Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Nativní SQL, aplikační vzory a propojení s UML

Návrhové vzory SQL

Diplomová práce

Autor: Bc. Roman Borkovec
Studijní obor: Informační management

Vedoucí práce: doc. RNDr. Jaroslava Mikulecká, CSc.

Hradec Králové duben 2016

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne..... .....

podpis studenta

Poděkování

Děkuji vedoucímu diplomové práce doc. RNDr. Jaroslavě Mikulecké, CSc. za ochotu při odborných konzultacích, metodickou, pedagogickou a odbornou pomoc při zpracování mé diplomové práce. Taktéž děkuji za odbornou pomoc i osobní podporu svojí paní, ing. Monice Borkovcové a váženému spolupracovníkovi ing. Jiřímu Kohoutovi jako vytrvalému korepetitorovi sql návrhů.

Anotace

Práce popisuje problematiku datových vzorů v rozsahu základních, aplikovaných i abstraktních. Vychází z historických zdrojů, důvodů jejich vzniku a možnosti jejich použití v oblasti výuky a abstrakce, datového modelování i v řešení konkrétních úloh v datové vrstvě.

Annotation

This thesis focuses on data design patterns: seed models, learning as well the abstract patterns. The idea of data model patterns is described considering the historical reasons of their development, their evolution and possibilities of using the patterns in learning methods, abstraction, data modeling and specific solutions, as well the possibility of their connectivity with the UML diagrams.

Obsah

1 Úvod

Tvorba počítačových informačních systémů a její metodiky jsou jedním ze základních kamenů vývoje software vůbec. Od okamžiku, kdy se člověk rozhodl svěřit svoje myšlenkové pochody počítačům, byl stvořen nejen programátor, jakožto dělník nového typu, ale také analytik, jehož cílem bylo zabránit hromadění nepřehledného kódu a zefektivnit práci podobně, jako v každém jiném, odborně i výkonnostně intenzifikovaném oboru.

V současné době již není příliš reálné si představit vývoj informačního systému (dále IS) bez důkladné analýzy a vlastně kompletního analytického modelování. Přesto, že základní realizace IS se v zásadě příliš neliší od prvotních řešení typu zajištění klientských požadavků, zajištění realizačních prostředků, postupná realizace řešení a nakonec uvedení do provozu, jednotlivé kroky těchto fází prošly obrovským vývojem. Od neřízeného vývoje se došlo až ke kompletní metodice realizace vícevrstvých (v případě IS téměř vždy třívrstvých) systémů, kde největší pozornost jak ze strany analytického návrhu, tak i ze strany koncové realizace dostává business vrstva. Zde se odehrává veškerá logika aplikace, včetně zajištění práv a přístupů dle požadavků nejen klienta, ale i logiky aplikace samotné. Otázkou je, jestli v rámci implementační architektury není chybou přenášet obecnost technologické architektury i do celkového řešení IS – viz Řepa [12].

Přenesením základní zátěže do této business vrstvy se na jedné straně efektivně soustřeďuje cílová realizace IS na myšlenkovou platformu – jinými slovy, samotné programování, resp. znalosti a dovednosti v konkrétních platformách a prostředích již nejsou podstatnou složkou ve srovnání s dokonalostí návrhu IS. Na straně druhé však často dochází k tomu, že veškeré schopnosti konkrétních databázových serverů, na nichž je realizována třetí vrstva (data layer), zůstává použita pouze jako datové úložiště. Vzhledem k několika desítkám let vývoje těchto serverů se pak stává, že potenciál a efektivní využití těchto serverů zůstává stranou. Dochází k tomu i z důvodu oné často zmiňované vlastnosti třívrstvých aplikací – nezávislosti na konkrétním řešení datové vrstvy.

Analytik se tak přestává zajímat o fyzické řešení – nezajímá ho výkon a vlastnosti jednotlivých serverů, očekává, že každý databázový server se bude v různých podmínkách užití chovat stejně. Ve výsledku, po velice kvalitní analýze a návrhu s použitím moderních metod a prostředků [1], může pak aplikace výkonově zklamat. Problematika tzv. degenerace databázových serverů, resp. jejich využití, je popsána jako častý dopad pojetí konceptuálního návrhu bez ohledu na reálné databázové úložiště tak, jak se o tom zmiňují Řepa [12] , Tropashko [18] a také i někteří vyučující dané problematiky ve svých materiálech[1].

[1] Zechmeister, Univerzita Pardubice, přednášky k předmětu IDAS1

Dalším důvodem, proč je datová vrstva poněkud opomíjena, je i její nevýrazný průnik do modelovacích nástrojů. Součástí návrhů bývají maximálně fyzické datové struktury, ale nikoliv modelové propojení včetně využití návrhových vzorů. Podobnou problematikou se zabývá i Palovská, jejíž práce [32] se stala jedním z mnoha zdrojů práce autora.

Autor ve své práci použil vývojové nástroje od firmy Microsoft – Visual Studio 2010 Express verzi a databázový server MSSQL 2014 [19], také v express verzi. Dále byly použity databázové servery Oracle XE11.2 od firmy Oracle [7], databáze DB2 v express verzi [17], univerzální databázová konzole firmy EMS a open-source produkty Firebird (databázový server) a MySQL. Pro účely praktického otestování kvantového emulátoru byl použit jazyk QCL[31].

2 Cíl práce a přehled postupů řešení

Hlavním cílem práce je popsat techniku tvorby návrhových vzorů SQL a navrhnout jejich přímé využití v modelovacím nástroji Enterprise Architect.

Celkově práce nabídla realizaci následujících dílčích cílů:

- Propojení analytického návrhu s možnostmi datového úložiště.
- Realizaci nestandardních aplikací za použití návrhových vzorů SQL.
- Pochopení vlastností konkrétního datového úložiště.
- Shrnutí všech zásadních typů návrhových vzorů umožňující pohled na datová úložiště nejen jako na uzavřenou a v důsledku funkcionálně nevyužitou entitu. Koncepčně tyto typy byly rozděleny do čtyř kategorií – aplikační, abstraktní, nativní a výukové.
- Shrnutí konkrétních nativních vzorů s cílem ukázat možnosti datových úložišť se zaměřením na efektivitu a nezávislost využití na konkrétním databázovém stroji.
- Návrh výukového modelu základních návrhových vzorů a celkový přístup k hlavním výukovým postupům v tématice SQL.
- Demonstraci praktického i budoucího využití návrhových vzorů – výběr vhodného úložiště a kvantové databáze.

Při realizaci cíle autor vychází z prací Silverstone [14][15][16], které již v padesátých letech minulého století popsaly nejen možnost, ale i nutnost tvorby datových modelů, čili v dnešním kontextu návrhových vzorů. Autor představuje základní aplikační vzory datové oblasti, a to nejen rešeršně shrnuté, ale i s vlastním přínosem z jeho vlastních praktických realizací.

V další části práce jsou shrnuty abstraktní datové vzory, jejichž přínos není přímo svázán s konkrétní aplikací. Pro čistotu konečného řešení je nutné tyto vzory akceptovat a v rámci vývojového týmu mít ujasněna a konkretizována jejich řešení.

Pro konkrétní využití nativních možností databázových serverů se autor rozhodl definovat základní, ale přesto netradiční databázové vzory. Tato návrhová knihovna umožňuje efektivní a standardizované řešení běžných problémů na databázové vrstvě, jako jsou např. odstranění duplicitních záznamů nebo rekurzivní SQL dotazy. Stejně jako implementace vzorů v Enterprise Architect je i tato část realizována jako vlastní výzkum.

Problémem, který nesouvisí přímo s vývojem IS, ale přesto je nedílnou složkou výuky vývojáře, jsou návrhové vzory pro výuku SQL. Z principu usnadňují pochopení a

urychlují poznání problematiky. Velmi důležitou složkou této části použití návrhových vzorů je profesionalizace užití SQL – výuka je díky tomu vedena na úrovni, která by pak měla odpovídat i profesionálnímu užití SQL v reálném světě. Nezahrnuje tak v sobě užití na konkrétním paradigmatu, ani to není cílem této práce.

V návaznosti na výuku SQL autor doplňuje zhodnocení práce dvěma konkrétními modely využití vzorů v praxi. Jedním z nich je testování a porovnávání databázového výkonu, resp. tvorby sad datových vzorů pro tento účel. Na rozdíl od tohoto problému směřuje druhé zhodnocení do možné budoucnosti – využití datových vzorů v kvantových databázích.

Posledním problémem, kterým se práce zabývá, je implementace návrhových vzorů do samotných schémat Enterprise Architect, jakožto modelovacího nástroje. Autor se pokusí navrhnout efektivní způsob propojení modelu s novými prvky a zároveň provázání stávajících návrhových schémat v součinnosti s datovou vrstvou. Tak, aby se bez ztráty zobecnění datové vrstvy tato stala podstatnou a nezbytnou i v rámci modelu IS. Zároveň v návaznosti na svoje práce předešlé [4][5] navrhne propojení s generickým frameworkem, a to za účelem zvýšení provázanosti samotné realizace IS s modelem.

3 Abstraktní vzory v aplikačním pojetí

Při návrhu informačního systému zkušený vývojář používá osvědčené aplikační vzory. Nejedná se o programátorské modely, jedná se o modelové vzory, které v součinnosti se zadavatelem předkládá jako osvědčené řešení. Technicky se tyto vzory začaly konstruovat již v době, kdy je ani nebylo možné aplikovat v počítačovém prostředí. Jejich účelem v prvopočátku bylo stanovit určitá pravidla pro oběhy dokladů ve firemních procesech tak, aby bylo dosaženo určité míry optimalizace a bezpečnosti těchto procesů. Z té doby pochází i první práce Lena Silverstona [14], na kterou pak navázal i konkrétními aplikačními vzory, realizovanými již v prostředí počítačových řešení [16].

3.1 Součinnost vývoje aplikací, aplikační vzory

Vyvinout aplikaci, respektive celý informační systém, není triviální záležitost. Je nutná souhra znalostí, zkušeností, komunikace s klientem a kvalita vývojových nástrojů. Nezbytným prvkem realizace IS je nejen schopnost takový systém vyrobit, ale zejména realizovat jeho budoucí doplnění, rozšíření a podporu vůbec. V praxi to znamená, že realizační tým musí řešit požadavky „do budoucna“, očekávat budoucí realizace tak, aby nedošlo k zásadním a nutným přestavbám jádra IS a uživatel mohl bez takových změn používat IS trvale při své práci.

Zároveň je třeba domýšlet určitou nezávislost na současných personálních zdrojích, vytvářet systém tak, aby byl technicky transparentní a dodržoval jednotné postupy technického řešení. To vyžaduje součinnost a vyrovnanost analytických prací s vývojářskými realizacemi. Jakmile dojde k tomu, že analytická schémata a podklady začnou být oproti technickým realizacím neaktualizována, začne být celá realizace IS nekonzistentní, což v důsledku přináší nejen zvýšení režie, ale také to obvykle má přímý dopad na funkčnost a efektivitu samotného IS.

Ve své bakalářské práci se autor zamýšlel nad jedním z řešení, jak efektivně zajistit propojení analytických prací s vývojem [4]. Součástí této práce je vytvoření návrhových vzorů na úrovni datové vrstvy tak, aby došlo nejen k zpřehlednění celého vývoje IS, ale zároveň i optimálního využití databázového prostředí, které bývá oproti business vrstvě upozaděno.

Již od konceptuálního počátku vývoje IS (jedná se zhruba o padesátá leta dvacátého století) se začalo diskutovat o základních vzorech, které je možno při vývoji IS použít. Tyto první vzory směřovaly přímo k aplikačním problémům – např. vystavení faktury, kontroly kvality výroby atp. V době, kdy ještě nebyla stanovena pravidla ani strukturovaného programování, takový model zpravidla zakresloval myšlenkový návrh přímo na konečných fyzických strukturách. Přesto jsou tyto aplikační vzory s jistou dávkou

abstrakce použitelné pro návrhy IS a díky určité normalizaci nejen na straně vývoje, ale i v samotných procesech, tvoří prvotní prostor pro budoucí kompletní definici řešeného IS.

Z pohledu aplikačních vzorů lze tak uvést tyto základní realizace:

- role a přístupy,
- hierarchie, agregace, vazby – rekurzivní vzory,
- stavové vzory,
- reporting.

3.1.1 Role a přístupy

Problematika rolí a přístupů k informačnímu systému je široká, a lze ji definovat v několika rovinách. V době, kdy vznikal první aplikační vzor v této oblasti (50. léta 20. století) nebylo třeba řešit příliš problematiku bezpečnosti – ta byla sama o sobě řešena odděleností tehdejších výpočetních sálů a jejich nekonektivitou s okolím. Zároveň je dobré připomenout, že neexistoval efekt domácích počítačových uživatelů, což znemožňovalo pokusy o neautorizované přístupy do systémů.

Vývojáři se v té době mohli soustředit na autorizaci procesů, tzn. zajistit, aby konkrétní procesy mohli realizovat konkrétní lidé. Jinými slovy, bylo třeba zajistit identitu uživatele proti odpovídající agendě. Odpovídající skupina uživatelů provozovala odpovídající skupinu aplikací, většinou v procesu zadání úloh (jobů) týmu operátorů výpočetního centra. Tento model nevyžadoval podrobnější řešení autorizace, a to i z důvodu, že prakticky neexistovalo uživatelské rozhraní. Operátoři nebyli uživatelé v pravém slova smyslu, byla to obsluha, která dozorovala průběh aplikace.

V okamžiku, kdy začaly vznikat počítačové sítě a zároveň operační systémy, které nabízely uživatelské rozhraní, bylo třeba změnit i principy aplikačních autorizací. Na níže uvedeném schématu je příklad poměrně sofistikovaného řešení, které umožňuje realizovat křížové využití rolí, včetně datumových omezení a dočasných pravidel:

Obrázek 3-1 Křížové využití rolí (zdroj [14])

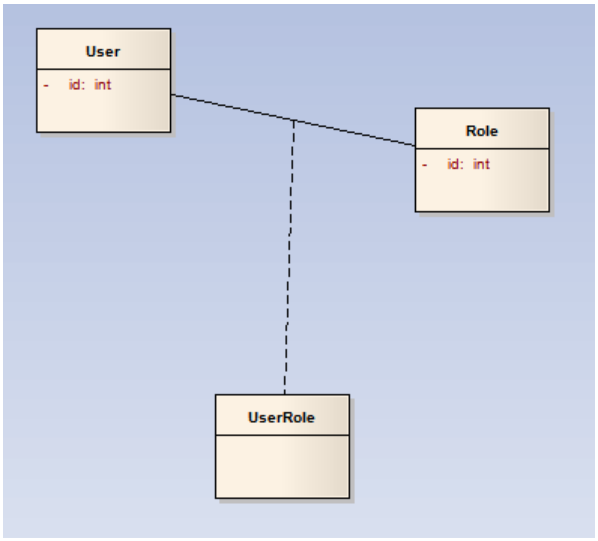
Celkově se tato problematika v současné době nazývá *membership* a zapouzdřuje nejen struktury uživatelů, rolí, skupin a jejich vazeb, ale také základní metody a omezení, které pak následně splňují i poměrně složité požadavky na celkovou autorizaci procesů.

Vzhledem k možné složitosti takových řešení se objevují i práce, které řeší optimalizaci rolí. Zahrnují v sobě nejen matematickou abstrakci, ale také metody operačního výzkumu [8]. Základním problémem přesto zůstává provozní adaptovatelnost v reálném čase, protože požadavky ve složitých systémech se mění rychleji, než je možné očekávat od běžného času analýzy, návrh a realizace takových optimalizací.

Vzory, zahrnuté v této práci, nesměřují na konkrétní typy samotných rolí (deklarativní, kontextové atp.), ale zcela ve smyslu tématu práce na fyzický návrh jejich vazeb vůči uživatelům v prostředí SQL.

3.1.1.1 Lineární model membershipu

Tento model je základní formou příslušnosti uživatelů a jejich rolí v systému. Existuje číselník rolí a seznam uživatelů systému. V asociační třídě *UserRole* je pak každému uživateli navázáno tolik rolí, kolik je třeba. Návrhový vzor je možno v rámci databázového serveru realizovat podle následujícího analytického modelu:



Obrázek 3-2 Lineární model membershipu (zdroj vlastní)

Výhodou tohoto vzoru je jeho transparentnost a jednoduchost, i když jeho maximální využití může pokrýt velké množství potřeb nad rámec původního vzoru. Je možno jej doplnit například o tyto možnosti:

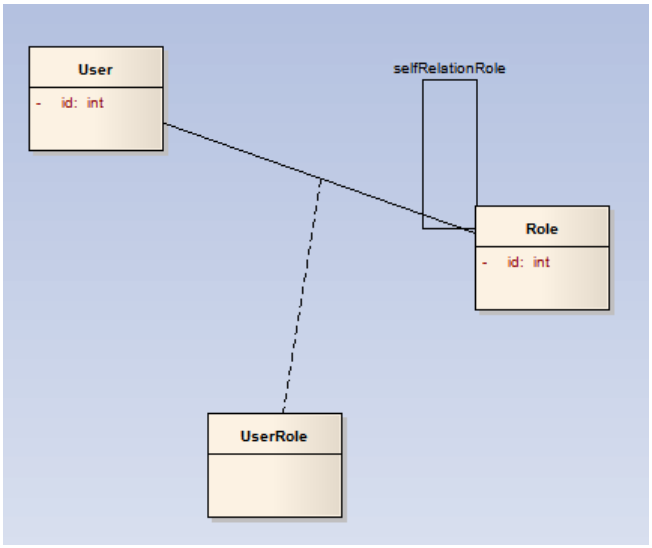
- Časové omezení rolí, uživatelů u vazby mezi nimi. Lze aplikovat např. pro dočasné zaměstnance nebo externisty.
- Skupinové role. Jedná se o substituci za množinu rolí tak, aby se zjednodušil nejen popis systému, ale také se usnadnila práce s uživateli, kteří mají společné větší množství rolí (nezaměňovat s dále popsaným rekurzivním modelem rolí.).

Pro vyjádření celkové funkcionality role pro jednotlivé entity a procesy aplikace pak stačí definovat metodu *isUserInRole*, která vyhodnotí autorizační požadavek proti uživatelským rolím a ve výsledku sdělí aplikaci true/false výsledek. Algoritmicky se jedná o křížové srovnání dvou řetězců, přičemž při první neshodě s požadavkem (tzn. že uživatel nemá požadovanou roli ve své množině rolí) tato metoda vrací false.

Z praktických realizací jednoduchého lineárního membershipu lze uvést např. hotelový systém pro Jerome Hotels s.r.o., kde se osvědčily jeho vlastnosti i při rozšíření systému z client/server aplikací na webové aplikace. Podrobnější popis systému je v příloze 3 této práce.

3.1.1.2 Rekurzivní model membershipu

Lineární slučování rolí dokáže transformovat jednoduché autorizační požadavky a jejich kombinace, které lze sloučit. V okamžiku, kdy přestávají stačit pouhé skupiny rolí, ale je třeba přenášet jejich atributy ve formě stromové struktury (dědění atributů), nelze se obejít bez rekurze.



Obrázek 3-3 Rekurzivní model membershipu (zdroj vlastní)

Ve vzoru je použita self-relace, což zajišťuje libovolnou požadovanou úroveň vnoření. Na konkrétních serverech typu MSQL nebo ORACLE lze použít pro fyzickou konstrukci vzoru dvě základní cesty:

- nativní datové typy *hierarchy:*
    - Výhodou je efektivnější využití databázových serverů, nevýhodou je jejich neexistence na některých serverech.
- standardní typy s následným rekurzivním vyhodnocením:
    - Výhodou je přímá realizace vzoru i řešení, větší transparentnost pro vývojáře. Nevýhodou je požadavek na větší schopnosti vývojářů při jejich realizace a s tím související možné komplikace – chybná algoritmizace, nedokonalá realizace, zavlečené chyby atp.

I zde je možno uvažovat metodu *isUserInRole*, je třeba však pamatovat na vnořenost uživatelských rolí proti autorizačnímu požadavku. Jedním ze způsobů, jak toto algoritmicky řešit, je rozdělit ověřování do dvou kroků:

- Poskládat z rekurzivního dotazu lineární vektor rolí uživatele.
- Následně použít stejný princip ověření jako u předchozího vzoru.

Tento rekurzivní vzor byl autorem použit v aplikacích pro ambulantní ordinace (Dyndal), kde při použití na víceordinačních zařízeních bylo zapotřebí zkombinovat pravidla rolí ordinací a celkového zařízení. Zároveň je použit jako základ membershipu univerzitního webu. V příloze 4 této práce se lze nalézt přesnější popis tohoto informačního systému.

3.1.1.3 Kombinační model

V některých aplikacích je požadavek na sloučení uživatelských účtů – znamená to, že uživatel, mající více jak jeden uživatelský účet (např. zaměstnanec, student, učitel), projde před samotnou autorizací procesem sloučení uživatelských účtů. Na straně rolí je jedno, jestli je použit vzor lineární nebo rekurzivní, to podstatné je poskládat možnosti rolí jeho jednotlivých účtu s preferencí silnějšího. Pokud jako student nemá přístup k určitým datům, tak v případě, že je zároveň učitel jej může dostat.

Výraz „může“ je vyjádření různých variant pro konkrétní realizaci kombinačního modelu v systému. Vždy se ale bude jednat o nejméně transparentní vzor, u kterého je vždy dobré promyslet skutečné důsledky jeho užití pro koncové uživatele.

Autor poukazuje na praktickou realizaci tohoto vzoru ve dvou aplikacích pro UHK, a to agendu *Přijímacích řízení* a agendu *Praxe*. I tyto systémy jsou popsány v příloze 5 této práce.

Při návrhu a použití vzorů, realizujících membership je dobré věnovat pozornost teoretickým modelům, jako je např. RBAC, popsaném v [13] případně jeho vícevztahová optimalizace tamtéž.

3.1.2 Rekurzivní vzory

Rekurze samotná je v principu základem objektové tvorby, ať již modelové nebo vývojářské. Dá se tedy očekávat, že bude k dispozici i v samotných aplikačních vzorech. V předešlé kapitole byl popsán vzor *Rekurzivní model membershipu*, který popisoval konkrétní situaci. Takových situací, které je dobře řešit jako návrhový vzor, je více.

Obecně u rekurzivního aplikačního vzoru vždy platí, že se v prostředí SQL bude jednat o self-relaci. Tato self-relaci musí vždy obsahovat ukončovací podmínku, jinak rekurzivní proces nebude ukončen a dle konkrétního prostředí to bude mít odpovídající, leč velmi nechtěné následky.

Rekurzivní vzor je možno nalézt při požadavku na vnořování informací, tzn. vývojář opouští standardní linearitu k dosažení informace a je nucen zanořovat se na požadovanou úroveň a teprve pak vracet informace.

V prostředí univerzitního webu byl za pomoci rekurzivního vzoru řešen *Dokumentový server*. Jeho princip zprostředkoval vývojářům a v rámci prezentační vrstvy i koncovým uživatelům emulaci souborového systému v databázi. Koncová fyzická realizace spočívala v jedné nosné tabulce *Documents* s následující nutnou strukturou:

Id	jednoznačný identifikátor záznamu (v koncové aplikaci použito pro specifikaci požadavku *download položky*.)

Nazev	název dokumentu nebo adresáře
Typ	0 – root, 1 – adresář, 2 – soubor (*Dokumentový server* je ve výsledku multiuser úložiště, proto je nutné uvažovat neomezené množství root-adresářů)
Content	binární obsah dokumentu
ContentType	typ dokumentu (jpg, doc, txt ….)
Access	definice práv a přístupů – využito v implementaci vzoru *Membership* proti autorizaci uživatele

Atributů tabulky může být podstatně více, záleží na požadovaných funkcích *Dokumentačního serveru.* Pokud by byl požadavek na verze dokumentů, je dobré připojit funkcionality dále uvedených stavových vzorů.

Rekurzivní kód není třeba řešit pouze na straně klienta – většina databázových serverů nabízí rekurzivní operace, ať již jako syntaktickou formulaci (with recursion), nebo implicitně při volání vnořených výběrů. Tato varianta umožňuje efektivně využít vlastnosti databázového serveru, bohužel některé ji stále ještě nenabízejí. V kapitole nativních návrhových vzorů SQL je přímo takový vývojářský vzor popsán.

3.1.3 Stavové vzory

V souladu s praktickými vývojáři uvádí Silverstone ve své publikaci [14] i formulaci *oběhu dokladu*. Jedná se o atributy záznamů, které určují, do jakých procesů mohou vstoupit. Ve většině užití je jejich schéma následující:

Vstupní hodnota atributu → proces → výstupní hodnota atributu.

Pořízený doklad má hodnotu stavového stributu dokladu takovou, aby bylo možno do jeho přijetí provádět úpravy. Jakmile jsou procesně zakázány tyto úpravy, mění se hodnota tohoto atributu. Pokud záznam prochází více procesy, obvykle dochází k odpovídajícím změnám atributu až do konečného stavu, což mohou být hodnoty *Vyřešen, Odstraněn* atp.

Tento základní stavový vzor je možno doplnit o další funkcionality:

- Evidence historie dokladu (systém je schopen vrátit informace o tom, kdy se doklad v jakém stavu nacházel, většinou za použití stavové tabulky, aby nebylo nutné provádět buď redundanci dokladů, nebo vytvářet specifické subtabulky pro konkrétní dokladovou entitu.

- Podobné rozšíření pro autorizaci změn stavů. Kromě informace o změně se eviduje informace o jejím autoru. Tento požadavek nemusí být triviální a může být provázán s vázanými faktory na membership, resp. správu uživatelů systému.

3.1.4 Reporting

Poměrně složitým vzorem může být i aplikační vzor Reporting. Zahrnuje v sobě správu tiskových výstupů, které informační systém nabízí.

Prvním požadavkem na vzor je autorizace uživatele, který má práva jej spustit. V návaznosti na vzor *membership* stačí dotaz na metodu *IsUserInRole* a bude tento požadavek splněn.

S reportingem těsně souvisí tiskové šablony a historie dat. Reportingový vzor tak musí uvažovat nejen možnost tisku aktuálních dat do aktuálních šablon, ale v podstatě všechny dostupné kombinace:

- Do stávající šablony vytisknout historická data (např. přehled studentů prvního ročníku před deseti lety).
- Do staré šablony vytisknout aktuální data (přehled dnešních studentů podle loňských požadavků).
- Do staré šablony vytisknout stará data (opakovaný tisk potřebných výstupů v historickém formátu).

Realizace reportingového vzoru je na úrovni SQL postavena na dvou úrovních:

- Verzování tiskových šablon s odpovídajícím obdobím platnosti.
- Verzování dat, s využitím stavových vzorů.

V případ použití MSSQL serveru lze vhodně použít z pohledu výkonu partitioning, kdy „stará“ data jsou odklizena do jiných prostor databáze a nezatěžují provoz aktuálních dat.

Jakousi přidanou hodnotou tohoto vzoru je i možná realizace jazykových verzí požadovaných výstupů, které cekovému řešení vzoru přidávají ve výsledku jeden rozměr navíc.

3.2 Vznikající aplikační návrhové vzory

V současné době je nutné realizovat vzory, zaměřené na budoucí aplikace. Rychlý posun v technologiích umožňuje a zároveň vyžaduje nové přístupy. Tyto se liší od původních myšlenkových konceptů, a jejich dopad pro budoucí aplikace je obrovský. Přínos budoucích aplikačních vzorů již není v tom, definovat business procesy, ale pokrýt obecnou škálu realizace celého business prostředí.

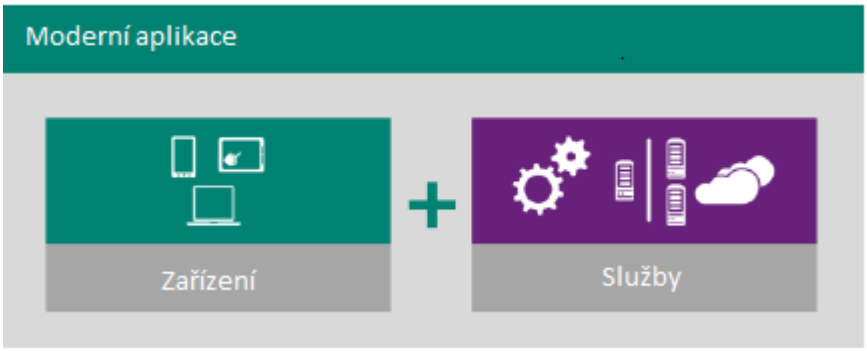
Zaměřují se dvěma hlavními směry:

- Tvorba vzorů přes rozhraní různých zařízení. V současné době již myšlenkové modely nemusí být vázány na konkretní fyzická zařízení, jejich dopad nemusí brát do úvahy cílový technický prostor, ve kterém budou realizovány.

- Tvorba standardních, „lightweigth“ služeb, nabízených jako rozšíření v cloudech. Tato vlastnost přestává omezovat dopad realizovaných vzorů, co do šíře saturovaných uživatelů. Stejně jako technicky není třeba řešit vlastnosti zařízení, není třeba ani řešit rozsah a dostupnost koncových klientů.

Tyto dva faktory ovlivňují stávající aplikační vzory a poskytují nové možnosti při návrhu budoucích vzorů.

Podle společnosti Microsoft ® je možno brzký výhled moderních aplikací znázornit tímto schématem:



Obrázek 3-4 Výhled moderních aplikací podle firmy Microsoft (zdroj [20])

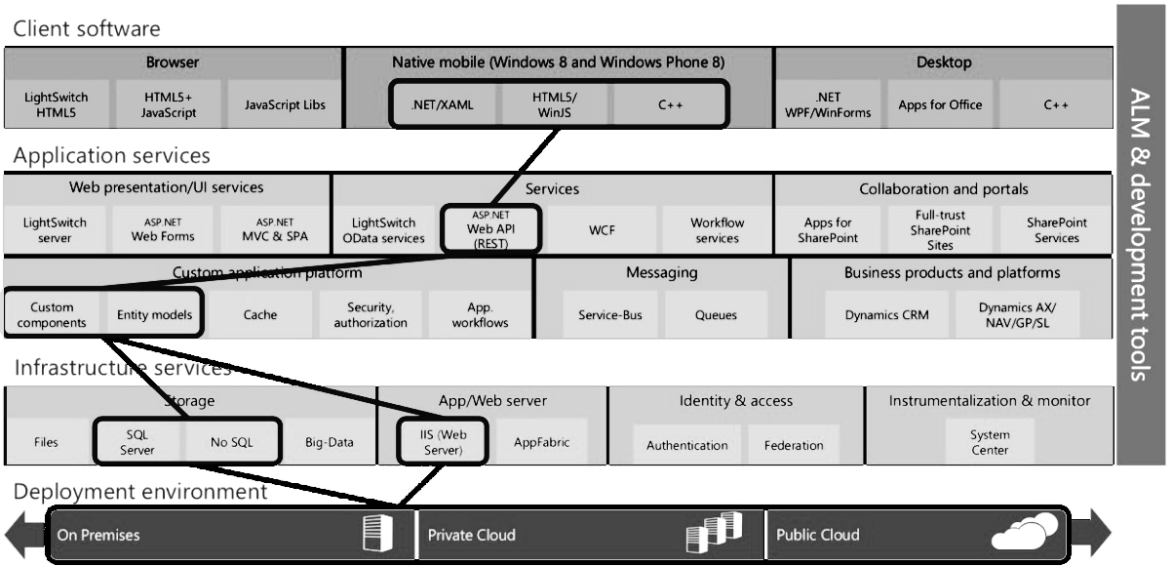
Z pohledu zařízení se pak směr vývoje orientuje dvěma směry – nativní a webové aplikace.

Nativní aplikace jsou aplikace běžící na konkrétním zařízení a využívající maximum jeho možností.

Webové aplikace jsou schopny pomocí technologií jako je HTML5, CSS3 dosáhnout toho, aby se i na klientských stanicích dosáhlo podobných efektů jako v případě nativních aplikací.

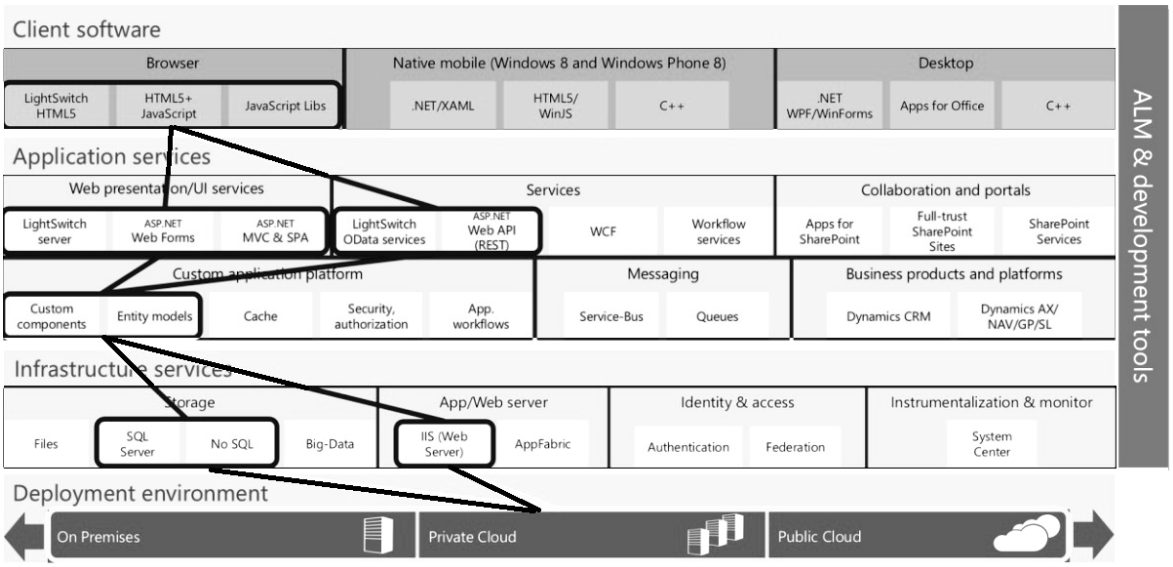
Vývojové nástroje nemají za cíl upřednostňovat jeden z obou směrů, ale naopak poskytnout vývojářům nástroje, které z obou směrů nabídnou nejkvalitnější realizace pro koncového klienta.

Základní scénáře se tak již nevěnují problematice aplikační, ale spíše realizační, ve smyslu prostředí a poskytovaných služeb. To se liší podle využívaných technologií, nebo přesněji celých vývojových ekosystémů. Pro budoucí, resp. vznikající aplikační vzory lze tak definovat v .NET ekoprostředí následující aplikační vzor (převzato z [20]):



Obrázek 3-5 Aplikační vzor v ekoprostředí .NET (zdroj [20])

Stejně tak i pro webové aplikace je možné v .NET prostředí vykreslit vzor, který splňuje představy o provázanosti zařízení a služeb v nedaleké budoucnosti:



Obrázek 3-6 Aplikační vzor pro webové aplikace v .NET (zdroj [20])

Na těchto schématech je vidět, že definování aplikačních vzorů se posunuje od hranic, které stanovovaly „CO“ je třeba řešit až k úrovni „JAK“. Probíhá podobný proces, jako v aplikačním vývoji samotném, kde se důraz překlápí ze samotné realizace na kvalitní model. Tendence, stanovit model na ověřených vzorech, tak povyšuje až nadaplikační uvažování a nabízí variantu konzistentních řešení bez omezení technologií a rozsahu služeb.

V souvislosti s budoucími vzory je třeba ve vztahu k datovým analýzám uvést i Machine Learning. Jde o novou technologii při práci s daty v prostředí Azure, která na rozdíl od data miningu používá prediktivní modely. Ty se učí z dat a měly by být schopny předvídat chování, trendy, výstupy. Vstupy mohou být brány z historických dat nebo přímo od uživatele. Je možné předpokládat, že i pro tyto modely budou definovány návrhové vzory, které by měly zajistit veškeré výhody plynoucí z jejich použití.

4 Abstraktní vzory v datové vrstvě

Moderní pojetí tvorby aplikací vychází ze souladu návrhu a samotného vývoje. Celkové členění aplikací do tří vrstev (prezentační, business a datové vrstvy) se ujalo jako obecně optimální model, který splňuje požadavky kladené na kvalitní návrh, čistotu kódu a propojení návrhových a realizačních postupů.

Třívrstvá architektura nemá jednoznačné a fixní řešení. Podle typu aplikace je možné základní schéma modifikovat, přesto stále implementační zaměnitelnost jednotlivých celků je základní premisou výhod této architektury. Jedno z mnohých uspořádání je zobrazeno na následujícím obrázku:

Obrázek 4-1 Třívrstvá architektura (zdroj vlastní)

Datová vrstva je dále rozčleněna na podvrstvu komponent logiky přístupu k datům, která zajišťuje modelovou otevřenost pro různá fyzická řešení. Zabaluje nejen konektivitu, ale také přístupy, a to jak z pohledu realizovaných optimalizací z pohledu výkonu daného úložiště, tak i z pohledu autentizačních procesů samotné aplikace. Tato podvrstva se nazývá Data Access Logic (DAL) a stejně jako ostatní vrstvy v návrhu je zcela autonomní – v praxi to znamená, že je možno na ní pracovat a zkvalitňovat jí v průběhu nejen vývoje, ale i provozu, aniž by bylo nutné zasahovat do aplikace samotné.

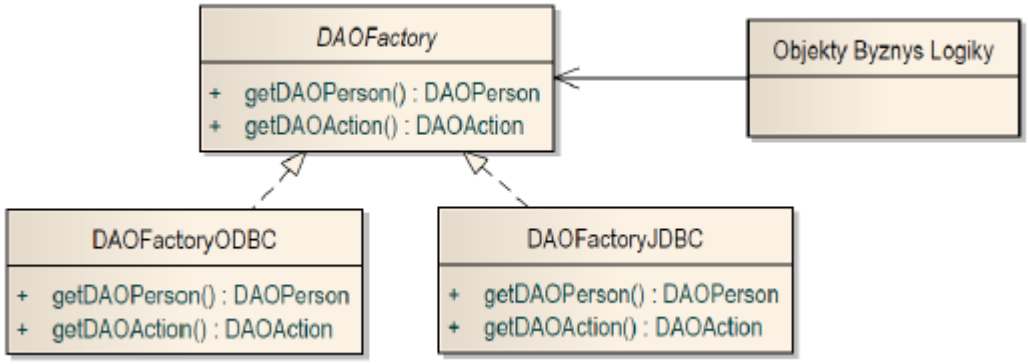
4.1 Abstract Factory

4.1.1 Popis vzoru

Vzor řeší použití konkrétních Factories, které řeší stejný problém různým způsobem. Klasickým způsobem využití je databázový provider. Ve výsledné realizaci nabízí stejné metody pro práci s libovolným datovým úložištěm.

4.1.2 Teoretická část

Obecné schéma vzoru lze zapsat několika způsoby. Uvedené schéma řeší základní objekty pro práci s metodami datové vrstvy a jejími uživateli.



Obrázek 4-2 Schéma vzoru Abstract Factory (zdroj vlastní)

Konkrétně je třeba řešit obecné metody typu *Connection, CreateParameter, SQLCommand aj.* tak, aby vývojář neměl důvod cokoliv ve svém kódu měnit v případě, že cílové úložiště změní svoji fyzickou charakteristiku. Klient, který potřebuje pracovat s daty, získá pouze instanci DAL komponenty. Podle načtení konfiguračních údajů tak vytváří instanci konkrétní továrny (factory).

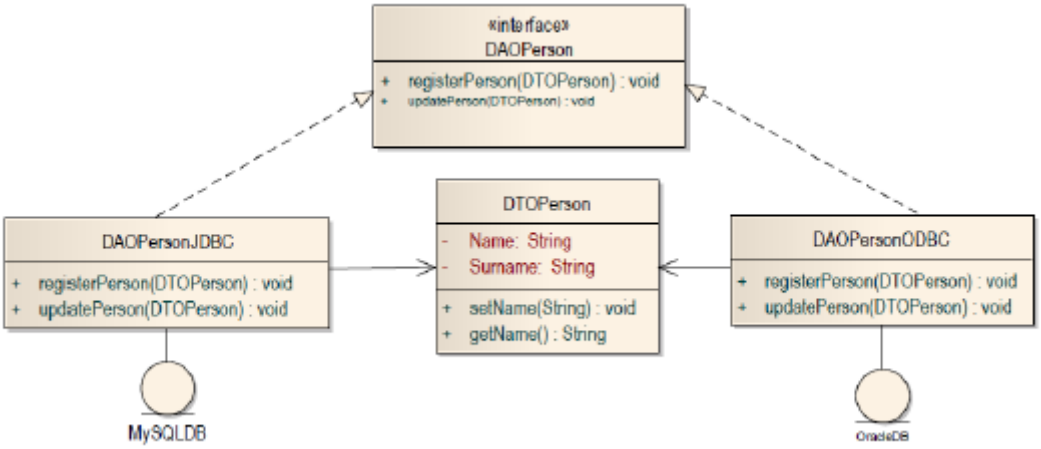
4.2 DataAccess Object

4.2.1 Popis vzoru

Tento vzor poskytuje data z databáze podle svých metod. Obaluje tak různé zdroje informací a zároveň poskytuje stejné metody pro získání stejných informací. Jeho mapování může být různé – viz vzor ACCESS RECORD.

4.2.2 Teoretická část

Vyjdeme-li z předchozího vzoru FACTORY, schéma vzoru DAO může vypadat, včetně použití FACTORY, následovně:



Obrázek 4-3 Schéma vzoru DataAccess Object (zdroj vlastní)

Principem vzoru je to, aby nabídl konkrétní metody pro business logiku, která se stává klientem vzoru. Klient je vlastníkem, implementaci přenechává datové vrstvě. Celkové řešení tak odstiňuje konkrétní implementaci na jedné straně vzorem FACTORY, na sraně druhé (k business vrstvě) vzorem DAO.

4.2.3 Praktická ukázka vzorů Factory a DAO

Předpokladem je, že informační systém je navržen tak, aby nebyl závislý na cílovém databázovém serveru. Důvodem může být realizační nezávislost na cílovém klientovi, nebo nutnost komunikovat s různými datovými zdroji. Používá obecnou Factory DbProvider, na které pak dle zadání ve web.config konkretizuje konkrétního providera (MSSQL, Oracle atd.). V příloze 7 je uveden konkrétní kód, který tento přístup zastřešuje. Byl použit ve věech autorem realizovaných IS nejen pro UHK a vychází z implementace DbProvider v prostředí .NET.

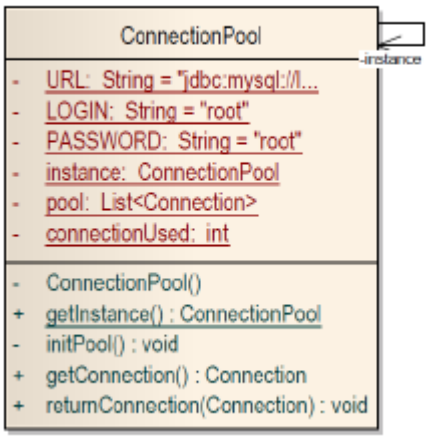
4.3 Pooling

4.3.1 Popis vzoru

Vzor POOLING nabízí možnost sdílet vytvořené instance v rámci systému (nebo jeho částí). Jeho hlavní funkcí je zajišťovat perzistenci objektů, resp. jejich instancí, jejich ditribuci a navracení zpět do úložiště.

4.3.2 Teoretická část

Vzor lze použít na vytvoření konečného počtu připojení, které pak třída ConnectionPool spravuje. Aplikace tyto připojení rozdává, což v důsledku zvyšuje rychlost vybavení klientských požadavků.



Obrázek 4-4 Třída Connection Pool (zdroj vlastní)

4.3.3 Praktická ukázka

Vzor POOLING není automaticky použitelný vzor. Je třeba zvážit, jakým způsobem se aplikace ke svým klientům chová a jaké jsou reálné nároky na konektivitu. To už bývá prakticky spojeno s konkrétním úložištěm a jeho vybaveností pro tvorbu spojení. Pro možnou realizaci vzoru je dobré se seznámit s možnostmi úložiště, pro účely této práce slouží příloha 15.1 – Srovnání databázových serverů.

4.4 Active Record

4.4.1 Popis vzoru

Vzor ACTIVE RECORD je způsob mapování DATA ACCESS OBJECT k databázi. Existuje několik přístupů k tomuto vzoru, jsou odvislé od míry podobnosti datové vrstvy a business logiky.

4.4.2 Teoretická část

Popis ukázkových vzorů implementace ACTIVE RECORD

Row Data Gateway

- Objekt, který svými atributy odpovídá právě jedné instanci v tabulce. Metody pak odpovídají základním databázovým příkazům typu insert, update a delete.

Table Data Gateway:

- Objekt je v principu celá tabulka. Nemá atributy, vstupní parametry se předávají jako parametry metod. Metody obvykle vracejí celé datové sety.

Active Record:

- Připomíná Row Data Gateway, ale navíc je spojen s business logikou. To znamená, že poskytuje širší informace, které odpovídají skutečným objektům business logiky.

Data Mapper:

- Ideální spojení mezi objekty datové vrstvy a business vrstvy, v podstatě na tomto vzoru jsou postaveny object-relation mappery. Některé z ORM si pak pomáhají dodatečnými sloupci, nebo vyžádaným pojmenováním některých sloupců – viz např. Entity Framework.

4.4.3 Praktická ukázka

Praktické použití vzor ACTIVE RECORD nabízí plné využití SQL daného úložiště. Výhodu je přesunutí výpočtu na databázový server, což zajišťuje nejen výkon, ale i transkační podporu, je-li třeba.

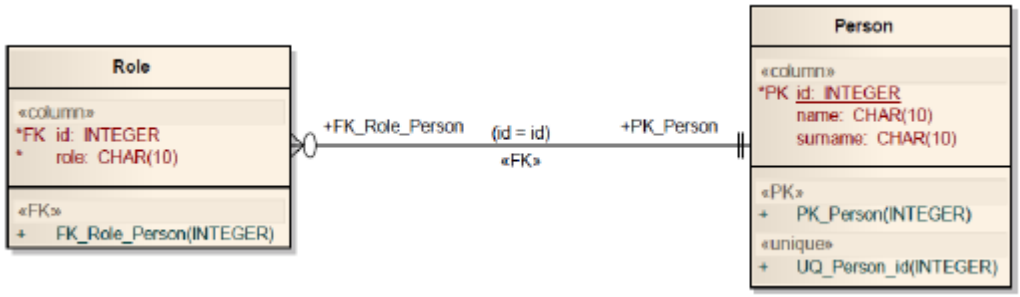
Nevýhodou je pak ztráta nezávislosti na konkrétním úložišti, ale to je nevýhoda pouze zdánlivá – ACTIVE RECORD se používá již v konkrétní vazbě na business logiku, pro realizaci by měl návazně použít odpovídající FACTORY a DAO a rozkládat tak zátěž na jednotlivé vrstvy.

Ve výsledku tak jsou třídy DATA ACCESS OBJECT mapovány na instance jednotlivých tabulek a mohou nabídnout komfortnější metody pro práci s daty.

4.5 Foreign Key Mapping

4.5.1 Popis vzoru

Princip foreign key je základní integritní omezení na úrovni databáze. Jeho využití je nejen zřejné, ale pro celkový výkon a ochranu dat i nezbytné. Tento vzor vychází z modelu tam, kde jsou definovány asociace různých charakteristik.



Obrázek 4-5 Schéma abstraktního vzoru ForeignKey Mapping (zdroj vlastní)

4.5.2 Teoretická část

Při použití tohoto vzoru je třeba respektovat následující pravidla:

- Vytvoření foreign key znamená vazbu mezi hlavní položkou a závislou. Není pak možno odstranit použitý klíč ze seznamu těchto klíčů.
- Nepoužití takové vazby výrazně zpomaluje datové procesy nad položkami.

4.5.3 Praktická ukázka

Zdálo by se, že neexistuje důvod, proč FOREIGN KEY nepoužít. Přesto takové situace existují:

- V době prvotního vývoje – a to nejen, pokud je realizován téměř souběžně s návrhem. V okamžiku, kdy je třeba nestandardně manipulovat s daty, cizí klíče mohou zbytečně zdržovat nutné operace. Pokud se cizí klíče doplní po stabilizaci struktury dat, má to ve výsledku vyšší efektivitu vývoje.
- Pokud je třeba pracovat s cizími daty z okolí aplikace. Pro hromadné importy, aktualizace atp. bývá výhodnější z hlediska rychlosti tato omezení nezávádět, popřípadě deaktivovat.

4.6 Serialized LOB

4.6.1 Popis vzoru

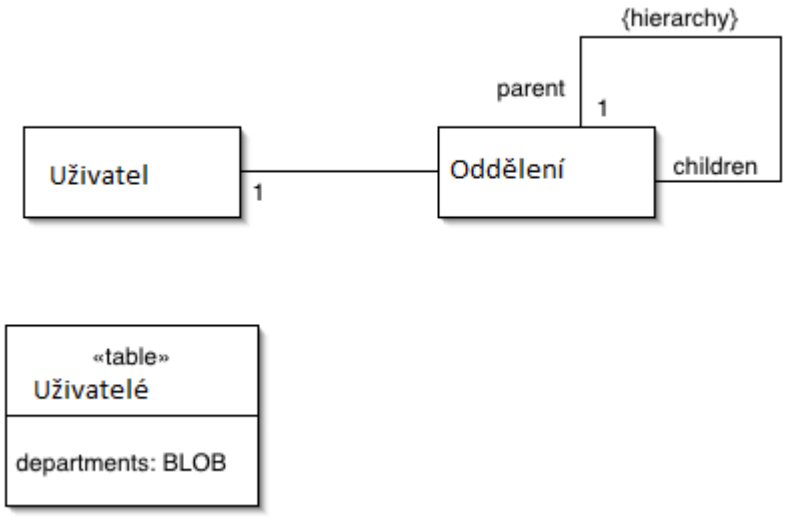
Tento vzor serializuje ve většině případů skupinu objektů, které nějak souvisí se zdrojovým. Zabaluje tak více struktur a v jednom streamu je posílá na datové úložiště. Toto zabalení může být implementováno jak vertikálně, tak i horizontálně, dokonce i hierarchicky.

4.6.2 Teoretická část

V případě hierarchie, jako nejzajímavější realizace vzoru, se nabízí návrh metod typu *getPredchudce, getNaslednik, getSourozenec* a mnoha dalších. Je třeba dbát zvýšené opatrnosti při fyzické realizaci na možnost špatného použití rekurzivních dotazů, nebo dalších *sql leaks* – tedy zbytečně zátěžových operací.

4.6.3 Praktická ukázka

Na schématu je ukázka vzoru pro hierarchickou skladbu dat.



Obrázek 4-6 Schéma abstraktního vzoru pro hierarchickou skladbu dat (zdroj vlastní)

Není možné nijak definovat rozsah, ten je specifikován až při konkrétním použití instance v závislosti na hloubce hierarchie.

Vzor tohoto typu by měl být realizován opatrně, jeho datová náročnost může být pro server velmi zatěžující. Příkladem je realizace na Centrálním registru vozidel, kde byl

modelově použit dobře, ale jeho praktická realizace na datové vrstvě zcela zbytečně předávala obrovské množství informací.

5 Realizace návrhových vzorů v nativním SQL

Nativní vzory jsou určeny pro řešení konkrétních situací, které je v rámci modelu řešit. Jejich definice v modelu usnadňuje nejen porozumění modelového záměru, ale také přináší veškeré výhody použití vzorů pro koncové vývojáře.

Nativním SQL se rozumí standardizovaný *Strcutured Query Language* tak, jak jej popisuje norma SQL92, resp SQL99. V případech, kde je třeba využít nebo poukázat na konkrétní rozšíření daného databázového serveru, je autorem na toto poukázáno.

Ačkoliv je přenositelnost SQL mezi databázovými servery omezená, navržené SQL vzory se drží výše uvedeného standardu bez omezení celkové efektivity řešení. Nedílnou součástí takového použití je i znalost vnitřních procesů konkrétního serveru tak, aby nedošlo ke kontraproduktivnímu užití.

5.1 COUNTING - četnosti

5.1.1 Popis vzoru

Základní návrhový vzor, jehož smyslem je zjistit přesný počet záznamů, vyhovujících zadání pro výběr. Řeší modelové rozdíly v případě použití DISTINCT, GROUP BY a možnosti využití CASE.

5.1.2 Teoretická část

Samotný COUNT(*) by asi nebylo potřeba v návrhu specifikovat. Pouze v okamžiku, kdy je potřeba využít jeho silných vlastností, vyplatí se tento návrhový vzor modelovat, aby nedošlo k nepřesnému pochopení definice konceptuálního modelu.

5.1.2.1 COUNT s argumentem

Prvním problémem je použití argumentu. Standardně se COUNT používá jako COUNT(*), přesto je možné použít syntaxi COUNT(column). Viz následující ukázka:

```
a)      SELECT COUNT(*) FROM zamestnanec

b)      SELECT COUNT(distinct jmeno) FROM zamestnanec
```

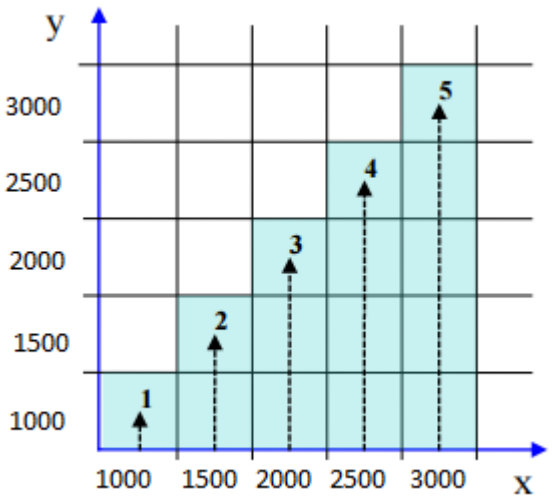
V prvním případě se vrátí hodnota 10 – počet záznamů v tabulce zaměstnanec. Ve druhém se vrátí hodnota 8, protože ve sloupci jmeno jsou dva zaměstnanci stejné hodnoty (Pavel) a jeden záznam nemá sloupec jmeno vyplněn.

5.1.2.2 COUNT ve složených příkazech

Druhou záležitostí jsou výběry, projekce, spojení a poddotazy. Užití COUNT je v takových případech poněkud nestandardní a při nedůsledné specifikaci návrháře může dojít k nesetkání se návrhu a výsledku. Příkladem může být řešení číslování řádků výsledného datového setu. Tuto problematiku lze řešit s konkrétními příkazy jednotlivých databázových serverů, ale jako ukázka využití cross-joinu v obecném SQL je velmi názorné. V tabulce *projekt* je atribut *sazba*. Pokud je třeba vypsat všechny sazby spolu se sloupcem, který označuje číslo řádku, lze využít právě cross-JOIN, resp. self-cross-JOIN:

```
SELECT t.sazba, COUNT(*) AS cRadku

FROM projekt t, projekt tt

WHERE tt.sazba <= t.sazba

GROUP BY t.sazba
```

Myšlenka vychází z následujícího schématu:



Obrázek 5-1 Schéma pro vzor COUNTING (zdroj vlastní)

Jeho výsledkem je následující set:

sazba	cRadku
1000	1
1500	2
2000	3
2500	4
3000	5

Obrázek 5-2 Výsledný set pro COUNTING (zdroj vlastní)

Dalším často užívaným vzorem pak může být počet podzáznamů hlavního záznamu. V ukázkové databázi je třeba zjistit počet členů týmů na jednotlivých projektech – jedná se tedy o využití COUNT v rámci poddotazu:

```
SELECT B.nazev, A.zkratka, (SELECT COUNT(*) FROM zamestnanectym

WHERE tymID = A.ID) AS pocetZamestnancu FROM tym A

INNER JOIN projekt B on B.id = A.projektID
```

Výsledný datový set pak vypadá takto:

nazev	zkratka	pocetZamestnancu
Web města Čáslav	WEBCS	5
Nemocnice Trutnov	NEMTR	5
Hotely Praha	HOTPR	0
Datové analýzy pro ABC. s.r.o.	DATAN_ABC	0

Obrázek 5-3 Výsledný set pro COUNTING s poddotazem (zdroj vlastní)

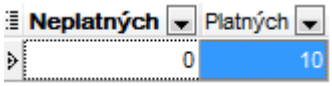
5.1.2.3 Podmíněný COUNT

V praxi jde o použití CASE přímo v příkazu COUNT, nikoliv v nějaké podmínce výběru nebo spojení. I tuto formulaci je dobré definovat vzorem v návrhu, její nepřesné zadání může způsobit odlišné výstupy.

Z tabulky *zamestnanecTym* je zapotřebí napočítat platné a neplatné členy týmů (atribut stav). Řešení podmíněného COUNT pak vypadá následovně:

```
SELECT

    COUNT(CASE WHEN stav = 0 THEN 1 ELSE NULL END) AS "Neplatných",

    COUNT(CASE WHEN stav = 1 THEN 1 ELSE NULL END) AS "Platných"

FROM zamestnanectym
```

Výsledný datový set dává tento výstup:



Neplatných	Platných
0	10

Obrázek 5-4 Výsledný set pro podmíněný COUNTING (zdroj vlastní)

5.2 SUMMARY - Podmíněná sumace

5.2.1 Popis vzoru

Podobně jako podmíněný COUNT, je možno tento vzor využít i pro další agregační funkce. Už na úrovni konceptuálního modelu ja tak možno doplnit závažné informace o zásadní představě výsledného objektu, resp. datového setu.

5.2.2 Teoretická část

Zatímco podmíněný COUNT je schopen zastoupit v obecném SQL funkci PIVOT, podmíněná sumace bude mít v praktickém užití patrně jiný účel. Jde o možnost smísit v jednom výsledném setu agregační informace podle typu konkrétního záznamu. Není to z pohledu teoretického návrhu příliš čisté řešení, ale v praxi je cílového klienta velmi často ideální. Nemusí provádět dvojí požadavek a získává dvojnásobnou informační hodnotu.

5.2.3 Praktická ukázka

Je třeba zachytit v jednom setu součet množství práce pro muže a finanční hodnotu tohoto množství pro ženy. Podobně by se dala taková dvojitá informace například zachytit u pracovníků THP a dělnických profesí, kde se výsledná hodnota počítá buď hodinově, nebo úkolově. V rámci příkladové databáze bylo vhodné jako ukázku použít první zadání.

Příkaz SQL:

```
SELECT A.id, A.jmeno, A.prijmeni, CASE A.pohlavi WHEN 0 THEN 'muž' else 'žena' END AS
Pohlavi,

   SUM(CASE A.pohlavi WHEN 0 THEN C.mnozstvi else C.mnozstvi * D.sazba END) AS
'M/hod,Z/Kc'

FROM zamestnanec A

INNER JOIN zamestnanectym B on B.zamestnanecID = A.id

INNER JOIN zamestnanectymprace C on C.zamestnanecTymID = B.id
```

```
INNER JOIN pozice D on D.id = B.poziceID

GROUP BY A.id, A.jmeno, A.prijmeni, CASE A.pohlavi WHEN 0 THEN 'muž' else 'žena' END
```

Výsledný datový set:

Tabulka 5-1 Výsledný set pro podmíněný vzor SUMMARY (zdroj vlastní)

id	jmeno	prijmeni	Pohlavi	M/hod,Z/Kc
1	Jan	Novák	muž	28
2	Karel	Novák	muž	3
4	Jiřina	Dvořáková	žena	840

Jak již bylo zmíněno, je možné rozporovat modelovou čistotu zdvojených informací, přesto mohou existovat situace, kde se podobný pattern uplatní.

5.3 SEQUENCE - Generátor sekvencí

5.3.1 Popis vzoru

Na rozdíl od Tropashka, který řeší tento vzor jako tvorbu jednoznačných identifikátorů číselného typu, tato práce rozšiřuje celou myšlenku o definiční a generované identifikátory. Účelem je nejen vyrobit abstraktní klíče, které nemají jinou datovou hodnotu kromě jednoznačné identifikace, ale také smysluplné definiční řady.

5.3.2 Teoretická část

Jako sekvenční identifikátory mohou sloužit buď autoinkrementační atributy (např. MSSQL), nebo sloupce provázané se sekvencemi (ORACLE, Firebird, od verze 2014 i MSSQL). Tyto hodnoty mají ve většině případů dvě funkce – zajišťují jednoznačnost záznamu v tabulce a také signalizují pořadí vytvoření záznamu. Následně se používají ve vazbách jako cizí klíče a celkově snižují náklady na výkonnost systému a orientaci v datech. Jednoznačný identifikátor typu GUID (UUID) má význam při distribuovaných databázích, kdy se mohou skládat různě vytvořené záznamy do jedné výsledné tabulky. Vzhledem k tomu, že GUID na rozdíl od celočíselného identifikátoru bude vždy jedinečný, není třeba doplňovat takový identifikátor nějakou verzí zdroje dat (databázového serveru). Nevýhoda GUID je v jeho velikosti (a v návazné velikosti případných foreign keys), nepřehlednosti a také v tom, že nenese žádnou další informaci – ani o pořadí záznamu, ani o jeho dalších inicializačních vlastnostech.

Tato práce si dovoluje rozšířit tématiku generování sekvencí i na zcela obecné číselné řady, a navazuje tak na předchozí bakalářskou práci autora. V ní byl naznačen generický přístup k tvorbě obecného typu sekvencí, který lze charakterizovat zhruba takto:

- Existuje tabulka, ve které jsou syntakticky popsány jednotlivé sekvence.
- Existuje uložená procedura (případně procedura na klientovi), která je schopna při požadavku na další hodnotu sekvence tuto hodnotu vyrobit a zároveň aktualizovat poslední použitou.

Tímto způsobem je možné zajistit např. čísla faktur, dokladů a dalších, již datově hodnotvých sloupců. Při dobrém syntaktickém návrhu se v těchto sekvencích mohou vyskytovat roční konstanty, střediskové hodnoty a všechny další, které jsou třeba.

5.3.3 Praktická ukázka

Jedním ze způsobů, jak takový generátor sekvencí vyrobit, je následující.

Mějme tabulku *generovaneRady* o následující struktuře:

Id	identifikátor řady
Nazev	název řady
Prefix	syntaktická maska kódu před iterační hodnotou
Value	iterační hodnota řady, definovaná jako char value – např. 000000 (určuje se tím zároveň zarovnání iterační hodnoty na počet číslic)
Suffix	syntaktická maska kódu za iterační hodnotou

Tímto způsobem lze například zajistit generování čísel faktur:

Nazev	Faktury	
Prefix	#YYYY	- bude přeloženo jako aktuální rok
Value	*000000*	
Suffix	#STRED	- bude přeloženo jako aktuální středisko

Výsledkem pak při volání procedury *generujRadu(nazev):varchar(20)* bude při prvním volání ze střediska ABC tato hodnota – 2015000001ABC. Na konci procedury se pak do sloupce *Value* uloží hodnota 000001, aby bylo možné iterovat řadu při dalším volání.

Masky lze definovat podle požadavků aplikace a schopností vývojáře, rozsah funkcionalit je obecně neomezený. Použití vzoru v návrhu pak zajišťuje přehledné uspořádání a definici nutných sekvencí, které jsou aplikací obsluhovány.

5.4 DECOMP - Dekompozice řetězců/kolekcí

5.4.1 Popis vzoru

Dle normovacích pravidel (Edgar F. Codd) by nemělo dojít k tomu, aby se ve sloupci vyskytovaly neatomické hodnoty. Existují situace, kdy v reálném prostředí takové záznamy vznikají – ať se jedná o import cizích dat, nebo vynucená data z důvodu jednoduššího kódování na straně klienta. Vzor dekompozice popisuje, jak z lineárně uspořádaných dat v jednom sloupci dostat data ve standardní datasetové formě.

sloupecA	-→	sloupecB
A,B,C		A
		B
		C

5.4.2 Teoretická část

Existují databázové servery, které umožňují tuto transformaci řešit přímo, mají to v rámci svých rozšiřujících funkcí. Pokud má být dekompozice nativní vzor, nezávislý na konkrétním serveru, bude třeba vyrobit funkci, která provede tzv. parse – rozdělení jednotlivých prvků podle předem zadaného oddělovače.

V každém případě není možné doporučit tento vzor jako zátěžový – tzn. jako trvalé užití nad velkými datovými sety. Je vhodný k přenosu dat, k uspořádání a ke zkvalitnění databázové struktury. Není vhodné ponechat data uložená tímto způsobem a využívat vzoru DECOMP jako standardního přístupu k datům.

5.4.3 Praktická ukázka

Pro realizaci dekompozice lze použít např. funkci fSplit s následujícím kódem:

```
CREATE FUNCTION [dbo].[fSplit] (     @dest nvarchar(max),
    @delim char(1),
```

```
    @where int
)
RETURNS nvarchar(max)
AS
BEGIN

          -- string = fSplit(vstup, oddelovac, poradi)
    -- pro prvni polozku se pouziva nula
    -- pokud je poradi vetsi nez pocet delimiteru, vrati null
    -- posledni znak nemusi byt delimeter

          declare @spom nvarchar(max)
    declare @spom1 nvarchar(max)
          declare @idx int
    declare @find int
    declare @length int
    declare @cnt int

    set @spom = @dest

    if (SUBSTRING(@spom,len(@spom),1)<>@delim) set @spom = @spom +@delim

    set @find = 0
    set @cnt = -1

    while ((@spom<>'') and (@find=0))
    begin
                    set @idx = CHARINDEX(@delim, @spom);
          if @idx>0
          begin
                    set @length = len(@spom)-@idx;
```

```
                set @spom1 = SUBSTRING(@spom,1,@idx-1 )

                set @spom = SUBSTRING(@spom,@idx+1,@length )

            set @cnt = @cnt + 1

            if (@cnt = @where)

            begin

                set @find = 1

            end

        end

        else set @find = -1

    end;


        if (@find<=0) set @spom1 = null

    return @spom1


END
```

Funkce umožňuje zadat řetězec k dekompozici, oddělovač a pořadí hledaného výrazu. Je použita v takřka všech realizovaných databázích autorem, s mírnými úpravami i pro různé servery (ORACLE, MSSQL, FireBird).

5.5 LISTAGGR - Agregace seznamů

5.5.1 Popis vzoru

Vzor Agregace seznamů funguje opačně, než vzor DECOMP. Jeho úkolem je do jednoho řádku výsledného setu poskládat více informací. Některé servery (Sybase, MySQL) mají tuto možnost přímo v základní kolekci příkazů, ale stejně jako u předchozího vzoru, neexistuje takový příkaz jako nativní SQL. Graficky lze princip vzoru zobrazit následovně:

Oddeleni	*Zamestnanec*	-→	*Oddeleni*	*Zamestnanci*
10	*Novák*		*10*	*Novák,Dvořák*
10	*Dvořák*		*20*	*Dryml*

20	*Dryml*	*30*	*Coufal, Lahoda*
30	*Coufal*		
30	*Lahoda*		

5.5.2 Teoretická část

Obecná realizace SQL není bez podpory konkrétní implementace možná. Pro ukázku byl použit MSSQL s tím, že ani tento vzor není vhodné použít na zátěžová data. Přesto je použit v několika aplikačních implementacích pro potřeby UHK jako jediné možné řešení. Jeho relativní neefektivnost je vyvážena omezujícím rozsahem požadovaných dat.

5.5.3 Praktická ukázka

Jako ukázku na přiložené databázi lze provést tento sql příkaz:

```
SELECT

zamestnanectymid,

 STUFF((SELECT  ','  +  CAST([id]  AS  nvarchar(max))  FROM  zamestnanectymprace  WHERE
(zamestnanectymid = Results.zamestnanectymid)

         FOR XML PATH (('')),1,2,'') AS NameValues

FROM zamestnanectymprace Results

GROUP BY zamestnanectymid
```

Jeho výsledkem je datový set, zobrazující lineárně přiřazené práce jednotlivých zaměstnanců v týmu:

Tabulka 5-2 Výsledný set pro LISTAGGR (zdroj vlastní)

zamestnanectymid	Prace
1	,2,3,4,5
2	
4	

5.6 RANDOM - Výběr náhodného záznamu

5.6.1 Popis vzoru

Výběr náhodného záznamu umožňuje z libovolného datasetu vybrat právě jeden náhodný záznam. Jeho použití je možné tam, kde je zapotřebí modelovat anebo provádět náhodné výběry z předem definovaných datových zdrojů libovolné složitosti. Byl použit i pro ukázku využití vzorů v kapitole 9.1.1.1

5.6.2 Teoretická část

Vzor využívá vlastností ORDER BY, které umožňují interpretovat libovolný, syntakticky správný výraz. Ačkoliv se tento příkaz používá většinou pouze na lineární řazení, lze ho použít i v tomto případě. Přidaná funkce newid() vrací náhodné GUID každému vybranému záznamu a následně pak vrátí pouze první záznam, jehož umístění v datasetu je zcela nepředvídatelné.

Zadání vzoru v diagramu obsahuje RANDOM, nebo RANDOM n, je-li požadován určitý počet náhodných záznamů.

5.6.3 Praktická ukázka

Zadání:

Vyberte dva náhodné záznamy ze seznamu zaměstnanců, kde datumNarozeni < 1.1.2000

Řešení:

```
SELECT TOP 2 * FROM zamestnanec WHERE datumNarozeni<1.1.2000

ORDER BY newid()
```

5.7 ORDERING - Podmíněné řazení

5.7.1 Popis vzoru

Vzor řeší situaci, kdy je třeba v jednom datovém setu realizovat více podmínek na jeho setřídění. Využívá většinou ne příliš užívaných možností příkazu ORDER BY.

5.7.2 Teoretická část

Jak již bylo naznačeno ve vzoru Výběr náhodného záznamu, příkaz ORDER BY má podstatně širší možnosti, než pouhé lineární řazení. Překládá dle vnitřního interpreteru v podstatě vše, co je mu dodáno, a to je mechanismus, kterým lze řešit i požadavek na podmíněné řazení.

Zadání vzoru v diagramu obsahuje definici CASE.

5.7.3 Praktická ukázka

V tomto případě autor zvolil ukázku v proceduře, pro ukázku širšího parametrického využití. Procedura *VyberZamestnance* vybere všechny záznamy z tabulky *zamestnanec*, a zároveň dojde k seřazení podle následujících hodnot parametrů procedury:

- Pokud je @OrderBy=‘Prijmeni‘ a zároveň @OrderByDirection= D, tyto záznamy seřaď podle *Prijmeni* a sestupně.
- Pokud je OrderBy=‘Prijmeni‘ a zároveň OrderByDirection != D, tyto záznamy seřaď podle *Prijmeni* a vzestupně.
- Pokud je OrderBy=‘Narozen‘ a zároveň OrderByDirection= D, tyto záznamy seřaď podle *Narozen* a sestupně.
- Pokud je OrderBy=‘Narozen‘ a zároveň OrderByDirection != D, tyto záznamy seřaď podle *Narozen* a vzestupně.

Řešení:

```
CREATE PROCEDURE dbo.VyberZamestnance (@OrderBy varchar(20), @OrderByDirection char(1))
AS

BEGIN

        SELECT * FROM zamestnanec

    ORDER BY

    CASE WHEN @OrderBy = 'Prijmeni'  AND @OrderByDirection = 'D'

                THEN Prijmeni END DESC,

        CASE WHEN @OrderBy = 'Prijmeni' AND @OrderByDirection != 'D'

                THEN Prijmeni END,

        CASE WHEN @OrderBy = 'Narozen' AND @OrderByDirection = 'D'

                THEN datumNarozeni END DESC,
```

```
        CASE WHEN @OrderBy = 'Narozen' AND @OrderByDirection != 'D'

                    THEN datumNarozeni END

END
```

Při volání procedury s parametry @OrderBy = Narozen a @OrderByDirection = D vrátí procedura tento datový set:

Tabulka 5-3 Výsledný set pro podmíněný ORDERING (zdroj vlastní)

id	jmeno	prijmeni	datumNarozeni	pohlavi
8	Marie	Dvorská	5.7.1998	1
9	Null	Dvorský	8.11.1996	0
10	Pavel	Dvořák	5.9.1991	0
6	Aleš	Horák	7.6.1990	0
3	Jana	Nováková	15.3.1988	1
4	Jiřina	Dvořáková	10.8.1986	1
7	Milena	Náprstková	30.7.1980	1
1	Jan	Novák	1.1.1980	0
5	Pavel	Dvořák	5.5.1975	0
2	Karel	Novák	22.2.1963	0

5.8 RECURSE - Rekurzivní dotazy

5.8.1 Popis vzoru

Vzor RECURSE zajišťuje rekurzivní získání hodnot z dat, aniž by bylo nutno programovat kód na klientovi. Výsledný objekt pak obsahuje záznamy, jež mají self-relation.

5.8.2 Teoretická část

Vzor souvisí s hierarchickým ukládáním dat, což je ne příliš časté užití v relačních databázích. Hierarchii lze využít nejen na datech typu adresářová struktura, ale také na přímých maperech objektů – tam se hierarchie přímo nabízí.

5.8.3 Praktická ukázka

V ukázkové databázi je tabulka *Document*, která naznačuje jednoduché úložiště dokumentů v podobné struktuře jako klasický souborový systém. Principem je pole

ParentID, které odkazuje na vlastnický záznam, a tím umožňuje neomezené zanoření a hierarchii.

Konkrétní ukázka SQL vrací počet prvků od dané úrovně níže.

```
WITH all_posts  AS
(        SELECT t1.id, t1.parentid AS parentid, t1.id AS root_id FROM document t1
--  WHERE t1.parentid is null      zde je možno určit počáteční úroveň
  WHERE t1.parentid = 1
  UNION ALL
  SELECT c1.id, c1.parentid AS parentid,p.root_id
  FROM document c1
    JOIN all_posts p on p.id = c1.parentid
)
SELECT root_id, COUNT(*) AS Pocet_prvku FROM all_posts
GROUP BY root_id
ORDER BY root_id;
```

Výsledný datový set:

Tabulka 5-4 Výsledný set pro RECURSE (zdroj vlastní)

root_id	Pocet_prvku
2	5
3	3

5.9 USERAGGR - Uživatelsky definovaná agregace

5.9.1 Popis vzoru

USERAGGR není standardním vzorem, pro který by stačilo čistě SQL řešení. Autor jej uvádí z důvodu jeho časté použitelnosti a na druhé straně jako ukázku propojení CLR .NET v prostředí MSSQL. Umožňuje definovat vlastní agregační funkci tam, kde nelze použít klasické SQL agregace. Důvodem může být použití agregace nad typem, který agregace v SQL nepodporuje, nebo sofistikovanější specifikace uživatele.

5.9.2 Teoretická část

Při tvorbě tohoto vzoru je třeba vytvořit vlastní agregační funkci (např. ve Visual Studiu), a tu následně implementovat jako rozšíření funkcionalit MSSQL. Toto je zajištěno použitím CLR, které od verze MSSQL 2005 umožnuje psát prvky serveru (uložené procedury, funkce a také vlastní agregace). Implementace je dvojího typu, buď pomocí příkazů sp_configure, nebo create aggregate a create assembly na straně MSSQL.

5.9.3 Praktická ukázka

V zadání je třeba agregovat datumové nebo časové položky, pro získání celkové časové režie. Prostý příkaz SELECT sum(datetimeColumn) skončí chybovou hláškou, která oznamuje, že tento datový typ nelze použít pro SUM.

Prvním krokem je definice agregační funkce.

```
[Serializable]
[SqlUserDefinedAggregate(Format.Native, Name = "SumTime")]
public struct SumTime
{
    private long ticks;
    private long minValueTicks;

    public void Init()
    {
        ticks = 0;
        minValueTicks = new DateTime(1900, 1, 1, 0, 0, 0).Ticks;
    }

    public void Accumulate(object Value)
    {
        if ((Value != DBNull.Value) && (Value is SqlDateTime))
        {
            DateTime add = ((SqlDateTime)Value).Value;
            ticks += add.Ticks - minValueTicks;
        }
```

```
    }


    public void Merge(SumTime Group)
    {
        ticks += Group.ticks;
    }


    public SqlDateTime Terminate()
    {
        return new SqlDateTime(new DateTime(1900, 1, 1, 0, 0, 0).AddTicks(ticks));
    }
}
```

Na straně serveru je třeba zajistit spuštění CLR:

```
EXEC sp_configure 'clr enabled', 1;
RECONFIGURE WITH OVERRIDE;
```

Po následném buildu se do MSSQL implementuje nová agregační funkce, kterou je možno použít jako jakoukoliv jinou.

5.10 PIVOT, UNPIVOT

5.10.1 Popis vzoru

Vzor PIVOT (resp. UNPIVOT) je určen k záměně řádků a sloupců výsledného datového setu. Lze ho realizovat pomocí podmíněných agregačních příkazů, nebo speciálním příkazem databázového stroje.

V případě vzoru UNPIVOT lze nadefinovat výběrový mechanismus tak, jak je uvedeno v kapitole 6.10.3.1

5.10.2 Teoretická část

Ve vzoru COUNTING je možno vypozorovat v jednom z jeho použití i možnost tvorby pivotovaného setu. Pomocí CASE lze vydefinovat hodnoty ve sloupcích a tak vytvořit sloupcově orientovaný set. (viz 6.1.2.3 včetně ukázky)

Některé servery nabízejí přímý příkaz PIVOT, který provede záměnu sloupců za řádky. Požadavek na pivotaci setu lze syntakticky definovat ve vzoru zadáním pevného a pivotovaných sloupců včetně podmínky.

5.10.3 Praktická ukázka

Tabulka *zamestnanectymprace* obsahuje kalkulovaný sloupec obdobi_COMP. Ten zařizuje zobrazení měsíčního intervalu, kdy byla práce realizována ve tvaru MM/YYYY ze sloupce *datumPrace*. Je třeba zobrazit součet množství práce pro každého zaměstnance týmu v požadovaných měsících:

```
SELECT * FROM

(SELECT zamestnanectymid, obdobi_COMP, mnozstvi FROM ZamestnanecTymPrace) AS tb1

 PIVOT (sum(mnozstvi) for obdobi_COMP in ([2014/10], [2014/11], [2014/12]) ) AS tb2
```

Výsledný set pak vypadá takto:

Tabulka 5-5 Výsledný set pro PIVOT (zdroj vlastní)

zamestnanectymid	2014/10	2014/11	2014/12
1	13	Null	5
2	Null	3	Null

5.10.3.1 Použití UNPIVOT

V tabulce *Projekt* jsou tři sloupce, označující nenavazující fáze projektu. Mohou být splněny v libovolném pořadí a je třeba zjistit, která proběhla jako poslední. Z dat, která jsou k dispozici v projektové databázi, je nutno obdržet tento výsledek:

Tabulka 5-6 Výsledný set pro UNPIVOT (zdroj vlastní)

ID	LastUpdateDat
1	20.7.2014
2	1.1.2015
3	4.12.2015
4	5.1.2015
5	5.3.2015

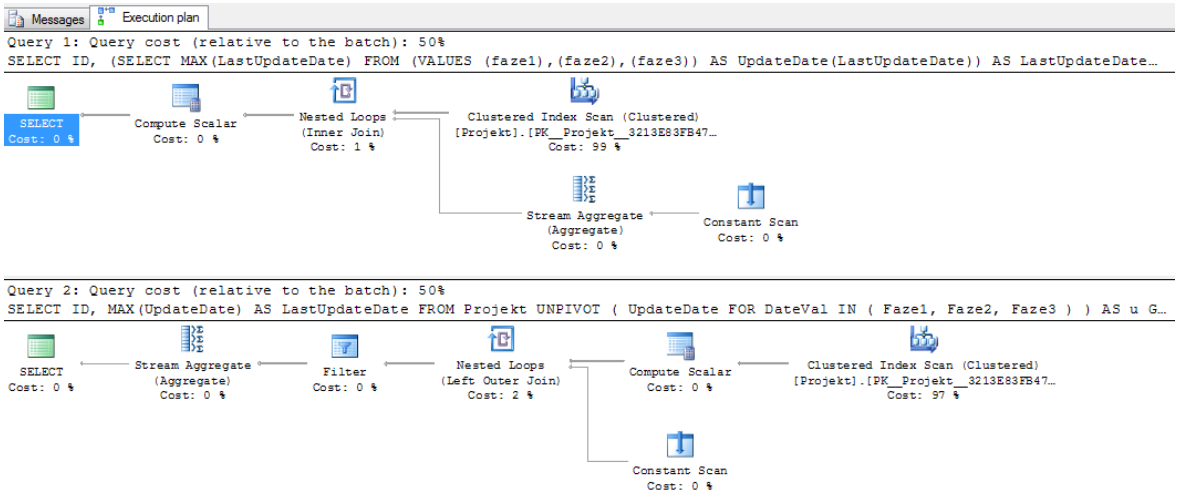
Standardní SQL toto může zajistit takto:

```
SELECT

   ID,

   (SELECT MAX(LastUpdateDate)

      FROM (VALUES (faze1),(faze2),(faze3)) AS UpdateDate(LastUpdateDate))

   AS LastUpdateDate

FROM Projekt
```

Zároveň je možno použít funkci MSSQL (podobně i v ORACLE), která přímo svým názvem definuje základní problém zadání – UNPIVOT:

```
SELECT ID, MAX(UpdateDate) AS LastUpdateDate

FROM Projekt

UNPIVOT ( UpdateDate FOR DateVal IN ( Faze1, Faze2, Faze3 ) ) AS u

GROUP BY ID, Nazev
```

Jen pro zajímavost autor uvádí srovnání obou příkazů, resp. jejich exekučních plánů. Kompletní analýza není účelem této práce a také je nutno výběr použité realizace uvažovat v konečných podmínkách databázového řešení:



Obrázek 5-5 Srovnání exekučních plánů pro různé varianty UNPIVOT (zdroj vlastní)

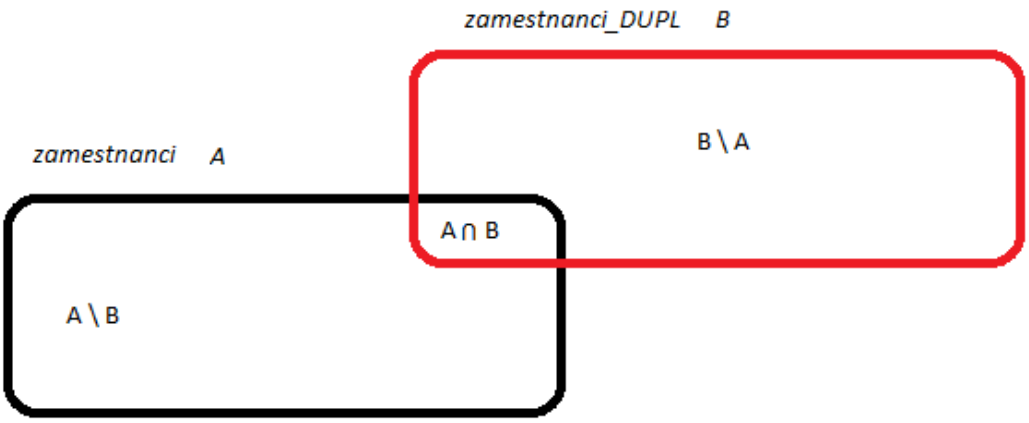
5.11 SYMDIFF - Symetrická diference

5.11.1 Popis vzoru

Tento vzor se myšlenkové blíží vzoru pro duplicitu záznamů, jen pracuje na srovnání dvou tabulek. Pro jeho aplikaci je v databázi tabulka *zamestnanec_DUPL*, která s vyjímkou jednoho záznamu obsahuje identická data jako tabulka *zamestnanec*.

5.11.2 Teoretická část

V praxi je nutnost srovnat obsahově dvě tabulky poměrně přirozená záležitost, ačkoliv se nebude týkat přímo modelové nebo vývojářské části. Přesto se tento pattern dá využít i v těchto oblastech, např. při realizaci aktualizací dat, přenosu atp. Základní myšlenkou je průnik rozdílů obou tabulek:



Obrázek 5-6 Průnik rozdílů dvou tabulek (zdroj vlastní)

Vzor lze řešit množinově nebo jako semi-JOIN, záleží na zvyklostech a možnostech konkrétního serveru.

5.11.3 Praktická ukázka

Pro MSSQL lze realizovat následující příkaz:

```
( SELECT * FROM zamestnanec

 except

 SELECT * FROM zamestnanec_DUPL

)
```

```
 union all (

 SELECT * FROM zamestnanec_DUPL

 except

 SELECT * FROM zamestnanec

)
```

Vrací následující set:

Tabulka 5-7 Výsledný set EXCEPT (zdroj vlastní)

id	jmeno	prijmeni	datumNarozeni	pohlavi
10	Pavel	Dvořák	5.9.1991	0
10	Pavel	Dvořák	5.9.1991	1

Z výsledku je vidět, že plní svoji funkci – vrací záznam z obou tabulek, který má alespoň jednu rozdílnou hodnotu. Pro ORACLE je nutno použít klíčové slovo *minus*, jinak je řešení stejné.

Semi-JOIN je funkční pouze na ORACLE, příkaz je pak možno definovat následovně, se stejnými výsledky. Jeho výhodou je možnost definovat sloupce, které jsou pro zachycení diference zajímavé.

```
SELECT * FROM zamestnanec

WHERE id, jmeno, prijmeni, datumNarozeni, pohlavi not in (SELECT id, jmeno, prijmeni,
datumNarozeni, pohlavi FROM zamestnanec_DUPL)

 union all

SELECT * FROM zamestnanec_DUPL

WHERE (id, jmeno, prijmeni, datumNarozeni, pohlavi) not in (SELECT id, jmeno, prijmeni,
datumNarozeni, pohlavi FROM zamestnanec)
```

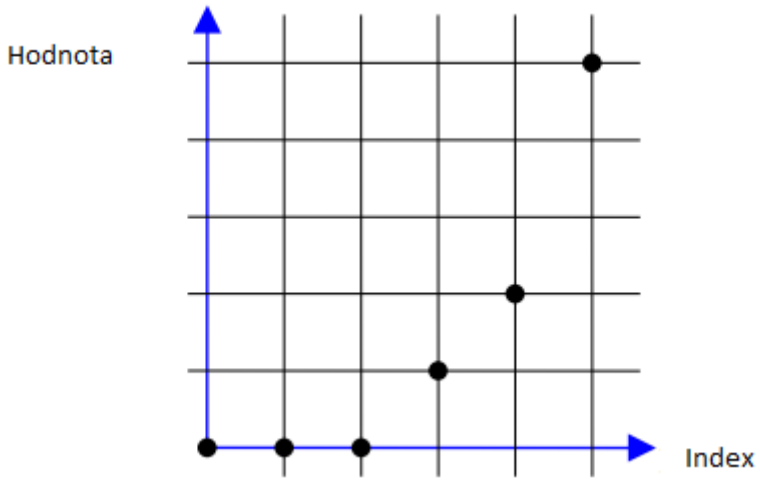
5.12 HISTOGRAM

5.12.1 Popis vzoru

Histogram je statistický pojem, který graficky vyjadřuje distribuci dat pomocí sloupcového grafu. Ani tento vzor nebude často aplikován, přesto je pro triviální zobrazení četností v některých případech vhodný. Umožní tak modelářům zadat požadavek na netradiční reprezentaci objektu v datovém setu.

5.12.2 Teoretická část

Vzor histogram se blíží pojetím vzoru Sloupec s pořadím – i zde je nutno připravit indexový sloupec, ze kterého následně vychází grafické zobrazení, čili histogram. Graficky lze vzor pojmout následovně:



Obrázek 5-7 Grafické vyjádření vzoru HISTOGRAM (zdroj vlastní)

Z uvedeného základního vzoru lze triviálně odvodit další typy histogramů, ať již zobrazení na výšku, nebo frekvenční.

5.12.3 Praktická ukázka

Pokud bude požadavek na histogramické zobrazení počtu zaměstnanců v týmech, je možné použít následující pattern:

```
SELECT zamestnanectymid AS ID, COUNT(zamestnanectymid) AS Pocet, right('**********…',
COUNT(zamestnanectymid)) AS 'Graf' FROM ZamestnanecTymPrace

GROUP BY zamestnanectymid
```

Jeho výsledkem bude tento set:

Tabulka 5-8 Jednoduchý výsledný set pro HISTOGRAM (zdroj vlastní)

ID	Pocet	Graf
1	5	*****
2	1	*

Pro větší hodnotu počtu jak 10 se vykreslí hvězdičky s tečkami, jako četnostně nezajímavá informace.

Pokud je třeba počet hvězdiček přímo svázat s hodnotou, stačí tento vzor upravit takto:

```
SELECT zamestnanectymid AS ID, COUNT(zamestnanectymid) AS Pocet,
right(replicate('*',COUNT(zamestnanectymid)),COUNT(zamestnanectymid)) AS 'Graf' FROM
ZamestnanecTymPrace

GROUP BY zamestnanectymid
```

Možnosti grafického vyjádření hodnot a vztahů v databázovém serveru ukazuje i kód procedury *getFK(tableName)* v příloze 8 – často požadované schéma uspořádání cizích klíčů, např. za účelem mazání vázaných dat.

Praktické použití vzoru v rámci UHK – podklady pro statistické analýzy z agENDy *Praxe*:

```
SELECT SUBSTRING(CONVERT(VARCHAR(13), log_stamp, 120),11,3) AS Hodina, COUNT(*) AS
Pocet,

right(replicate('*',COUNT(*)/100),COUNT(*)/100) AS 'Graf'

FROM mw_login

GROUP BY SUBSTRING(CONVERT(VARCHAR(13), log_stamp, 120),11,3)

ORDER BY 1
```

Vrací následující přehled:

Tabulka 5-9 Praktická ukázka použití vzoru HISTOGRAM (zdroj vlastní)

Hodina	Pocet	Graf
00	75	
01	26	
02	19	
03	13	
04	18	
05	67	
06	190	*
07	620	******
08	1 381	**********
09	1 739	**********
10	2 130	**********
11	1 727	**********
12	1 846	**********
13	1 822	**********
14	1 734	**********
15	1 360	**********
16	1 197	**********
17	1 270	**********
18	1 277	**********
19	1 438	**********
20	1 430	**********
21	1 098	**********
22	636	******
23	256	**

5.13 Dotazy typu Skyline

5.13.1 Popis vzoru

Vzory pro dotazy typu Skyline jsou založeny na nalezení optimálního řešení pro vícečetné podmínky. Z pohledu lineárního programování se jedná o vyhledávání dominátorů v zadaných stavech.

5.13.2 Teoretický popis

Vzor nelze realizovat jako standardní JOIN s podmínkami, ale je nutno předpokládat sloučení více variant.

5.13.3 Praktická ukázka

Je třeba nalézt projekty, které jsou pro zájemce zajímavé. V tabulce *projekt* existují atributy *typ* a *sazba*. *Typ* se pokouší nějakým rostoucím indexem popsat zajímavost projektu pro řešitele (např. jako vhodná položka do jeho profesního CV), *sazba* dává přímou finanční známku projektu. Řešitel se pokouší skloubit něco, co je pro něj zajímavé z obou těchto hledisek, podle srovnávacích podmínek. Výsledkem je tento příkaz:

```
SELECT * FROM Projekt c

WHERE not exists (

 SELECT * FROM Projekt cc

 WHERE cc.typ >= c.typ and cc.sazba > c.sazba

 or cc.typ > c.typ and cc.sazba >= c.sazba

);
```

Ve výsledné nabídce dostane řešitel tyto záznamy:

Tabulka 5-10 Výsledný set pro SKY LINE (zdroj vlastní)

id	nazev	typ	stav	sazba
4	Datové analýzy pro ABC. s.r.o.	3	1	2500
5	SONO centrum Mariánské Lázně	1	1	3000

Konečné rozhodnutí je na něm, tyto dva záznamy vracejí optimální variantu pro jimi nastavené podmínky – dominují.

5.14 DUPLICATE - Duplicita záznamů

5.14.1 Popis vzoru

Podobně jako vzor SYMDIFF je tento vzor připraven pro užití spíše v servisních nebo importních procesech. Podle definice zjišťuje a dohledá duplicitní záznamy buď na konkrétním sloupci, nebo na jejich požadovaném výčtu.

5.14.2 Teoretická část

Vzor používá vlastností agregačních funkcí a je ho možné použít ve standardu SQL. Existuje více možných fyzických řešení, a existuje i možnost získání více informací o duplicitních řádcích.

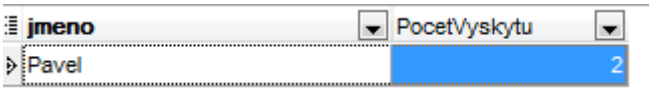
5.14.3 Praktická ukázka

Duplicita na konkretnim sloupci:

```
SELECT jmeno, COUNT(jmeno) AS PocetVyskytu

FROM zamestnanec

GROUP BY jmeno

HAVING (COUNT(jmeno) > 1)
```

Výsledný set:

Tabulka 5-11 Výsledný set pro DUPLICATE celého záznamu (zdroj vlastní)

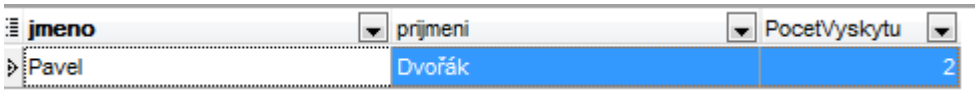


jmeno	PocetVyskytu
Pavel	2

Pro více požadovaných sloupců:

```
SELECT jmeno, prijmeni, COUNT(*) AS PocetVyskytu

FROM zamestnanec

GROUP BY jmeno, prijmeni

HAVING (COUNT(jmeno) > 1) AND (COUNT(prijmeni) > 1)
```

Tabulka 5-12 Výsledný set pro DUPLICATE dle vybraných atributů (zdroj vlastní)



jmeno	prijmeni	PocetVyskytu
Pavel	Dvořák	2

6 Použití návrhových vzorů při výuce SQL

Problematika výuky SQL je v současné době netriviální záležitostí. Vyplývá to i z předchozích kapitol a dá se to shrnout do následujících bodů:

- SQL jako jazyk přestává být využíván do maxima toho, co může nabídnout. Je nahrazen operacemi nad business vrstvou, která čím dál více obaluje samotné SQL dalšími rozhraními, jako je např. LINQ.
- Databázové servery jsou pro vývojáře nerozlišitelné. Nevěnují pozornost podstatným technickým rozdílům a vlastnostem, které konkrétní server nabízí a orientují se podle jiných podmínek. Komerční dostupnost serveru, cena.
- Vysoká orientace obecně na generické frameworky, mappery atp. Samotný vývojáře je pak odtržen od skutečného tvaru požadavku a není schopen ovlivnit jeho výkonnost.

V důsledku se pak vytváří situace, kdy vysoce výkonný databázový server je použit nejvýše jako datové úložiště a jeho silné stránky nejsou využity. Následně pak není ani ve výuce třeba trvat na konkrétních znalostech a potřebě znát o použitém nástroji jeho specifické vlastnosti.

U některých autorů (např. [27]) se v poslední době začíná prosazovat použití návrhových vzorů při výuce SQL tak, aby se předešlo výše shrnutým tendencím. Jejich hlavní smysl lze shrnout do následujících bodů:

- Pochopení základních operací nad databázovými servery.
- Transparentní popis rozdílu v přístupu jednotlivých serverů nad těmito operacemi.

Tato forma výuky pak může přinést ve výsledku odstranění výše uvedených nedostatků. Pokud by v této oblasti nedošlo k těmto změnám, může nastat situace, kde bude k dispozici obrovské množství návrhářů a modelerů, jen oni konkrétní kódéři už pak budou k dispozici jen v Indii, kde budou vyrábět kód na zakázku.

Z daných předpokladů pak vycházejí tři skupiny vzorů, které lze pro výuku SQL použít:

- Sub-query patterns.
- Katalog agregací.
- Joining katalog.

6.1 Sub-query patterns

Tato skupina vzorů obsahuje dva vzory – Virtual table pattern a Dynamic Filtering Criteria Pattern. Oba tyto vzory postihují základní požadavky na databázové servery – výstup kombinací dat bez nutnosti tvorby tabulek s redundantními daty a deklarativní výběr již z realizovaných datových setů.

6.1.1 Virtual Table Pattern

Vzor popisuje různé možnosti, jak získat propojená data. V rámci výuky je dobré intuitivně pochopit, že z existujících entit není třeba vytvářet další jen proto, že jsou samy o sobě autonomní. Pokud existuje v databázi tabulka *zamestnanci* a tabulka *detizamestnancu*, není třeba pro výstup informace o zaměstnanci a jeho dětech vytvářet další tabulku, ale provést spojení dvou existujících. Vytvoří se tak jakási virtuální tabulka, se kterou je pak z pohledu čtení informací možné pracovat jako s jednou konzistentní entitou.

6.1.1.1 SUBSELECT

Získání informací z poddotazu (subSELECT) je možno rozdělit na dva základní typy – skalární a multi-value.

- Skalární

Skalární, jednorozměrný poddotaz má typovou úlohu definovanou v dodání jednoznačné hodnoty pro hlavní dotaz. Tato dodaná hodnota může být např. poslední vložený záznam, nebo záznam s maximální hodnotou atp. Použití subSELECTu může obejít nutnost použití agregačních funkcí, nebo jinak v principu složitějších dotazů.

Příklad:

```
SELECT <sloupce>

FROM Zamestnanec, ZamestnanecTymPrace

WHERE ZamestnanecTymId = (SELECT top 1 tymid FROM ZamestnanecTym ORDER BY id DESC)
```

Tímto způsobem získáme přehled prací posledního zavedeného týmu do databáze.

- Multi-value

Vícehodnotový poddotaz vrací více než jednu hodnotu. Nelze jej tedy přiřadit jako skalární, ale je nutno použít klíčové slovo IN.

Příklad:

```
SELECT <sloupce>

FROM Zamestnanec, ZamestnanecTymPrace

WHERE ZamestnanecTymId IN (SELECT tymid FROM ZamestnanecTym WHERE tymid>1)
```

Pro zvýšení efektivity dotazů, resp. poddotazů s IN je dobré zvážit opačnou logiku výběru s použitím EXISTS, resp. NOT EXISTS. Tato varianta má podstatně menší režii, protože potřebuje najít pouze jeden prvek pod-setu.

6.1.1.2 APPLY

Pokud je třeba spojit záznamy podle pravidel nějaké UDF (user defined function), doporučují některé servery použít APPLY. Předpokládejme funkci, která vrací table-values, čili neskalární výsledky. Jediný způsob, jak tuto vrácenou tabulku spojit s jinou, je práve APPLY.

Příklad:

```
SELECT D.deptid, D.deptname, D.deptmgrid

    ,ST.empid, ST.empname, ST.mgrid

FROM Departments D

CROSS APPLY fn_getsubtree(D.deptmgrid) ST;
```

Funkce fn_getsubtree používá typický rekurzivní vzor (viz. Nativní vzory) a vrací zaměstnance pro každé středisko.

CROSS APPLY obecně přiřazuje každému záznamu levé tabulky všechny řádky pravé. OUTER APPLY přidává i ty z pravé, které mají hodnotu NULL.

6.1.2 Dynamic Filtering Criteria Pattern

Dynamické filtrování je výukový vzor, který ukazuje, jak navrhnout dotaz bez konkrétní podoby WHERE nebo JOIN. Připomíná vzor SUBSELECT z VIRTUAL TABLE, ale jeho účelem není vytvářet virtuální tabulku, ale získat informace, závislé na nekonstantních parametrech.

```
SELECT * FROM zamestnanec A

JOIN (SELECT tymID, zamestnanecID, FROM zamestnanectym ) B ON B.zamestnanecID = A.id
```

Uvedený příklad by se dal nahradit poměrně přirozeně klasickým spojením, pokud však je nutno použít složitější vnitřní příkaz, ušetří se přinejmenším použití pohledu:

```
SELECT B. Bor_id, B. Bor_name, F.Max(fine), F.numFine FROM Borrower B

JOIN (SELECT Bor-id, Max(Fine), COUNT(*) numFine FROM Fine GROUP BY Bor-id) F ON (F.Bor-
id =B.Bor-id)
```

6.2 Aggregation catalog

V případě získání kumulovaných informací, je třeba data vhodně seskupit. Toto seskupení se velice často váže s agregačními funkcemi, které zajišťují kumulativní operace nad daty. Zároveň lze tyto funkce použít i na lineárním datovém setu a jejich hodnoty získat přímo bez seskupení.

Agregační funkce vracejí vždy nějakou hodnotu, ale s jednou výjimkou – pokud se např. v subSELECTu nevrátí žádné řádky, provedená agregace v takovém případě NULL vrátí. Nezaměňovat s agregací řádků, které mají v daném sloupci NULL hodnotu, tam agregační funkce bude načítat nulu.

Příklad:

```
SELECT SUM(sazba) FROM Pozice WHERE id = 100

Vrátí NULL
```

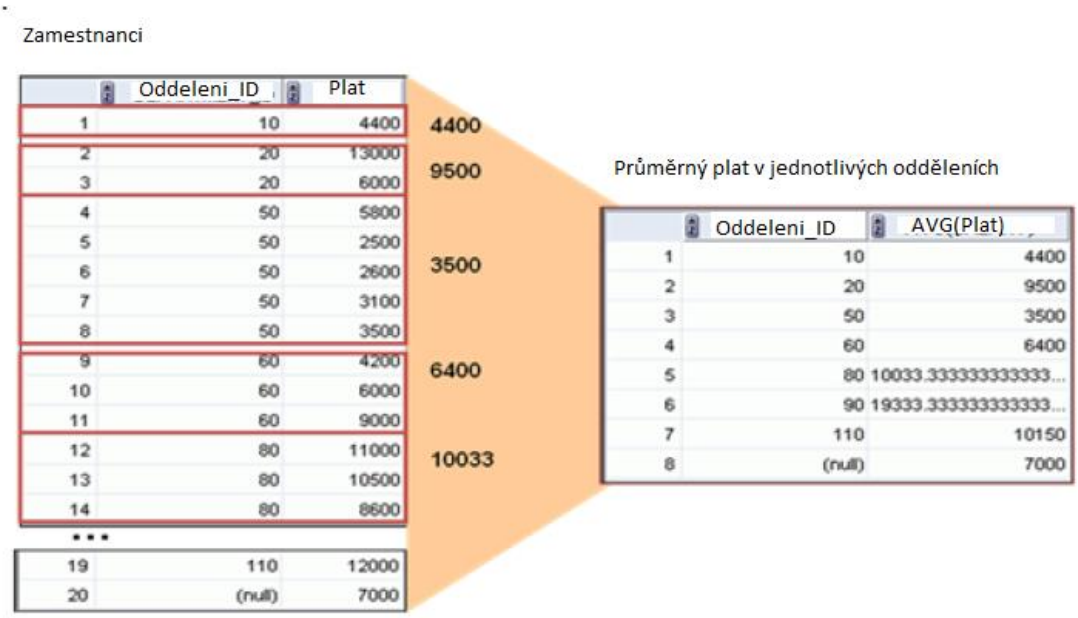
Řádek s ID = 100 neexistuje, proto je nutné při jejich použití převádět výsledky pomocí funkce ISNULL, nebo obdobné dle odpovídajícího serveru – tato funkce zajistí převod hodnoty NULL na nulu a tím neukončí případný výpočet s výsledkem NULL.

6.2.1 Grouping Result Pattern

Základní myšlenkou tohoto vzoru je správně charakterizovat jednotlivé sloupce požadovaného výstupu. Je třeba vědět, podle kterých sloupců seskupovat a které sloupce nesou výslednou skupinovou informaci. V předcházejícím případě nebylo třeba seskupení, protože se agregační funkce SUM použila na celou tabulku. Standardní použití je většinou postaveno na tom, že je třeba získat více agregačních hodnot pro několik skupin.

Grafická představa požadovaného seskupení může vypadat následovně:

Tabulka 6-1 Grafická představa požadovaného seskupení (zdroj [27])



Výsledná SQL realizace bude tato:

```
SELECT Oddeleni_ID, AVG(plat) FROM Zamestnanci

GROUP BY Oddeleni_ID
```

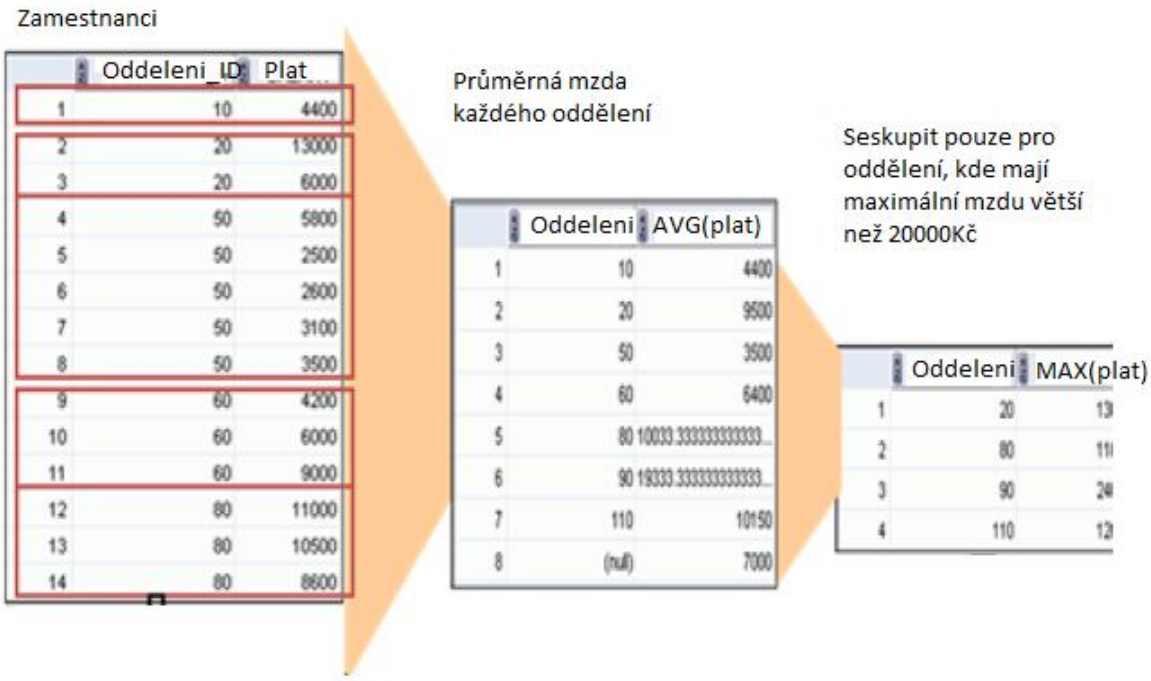
Při seskupování je třeba mít na paměti, že k němu dochází až po sestavení neseskupených dat. Znamená to, že případný WHERE se vkládá před GROUP BY a v žádném případě nemůže výsledné seskupení ovlivnit.

6.2.2 Restricting Grouped Result Pattern

Vzhledem k tomu, že v seskupení nelze použít WHERE a zároveň se z praktického pohledu nabízí potřeba nějak výsledné agregace podmiňovat (např. mě budou zajímat pouze ta oddělení, jejichž maximální plat je vyšší než 20000Kč), jeví se jako nezbytné nějak toto podmiňování agregací zařídit.

Opět grafická reprezentace požadovaného seskupení:

Tabulka 6-2 Seskupení s výběrem (zdroj [27])




```
SELECT Oddeleni_ID, AVG(plat) FROM Zamestnanci

GROUP BY Oddeleni_ID

HAVING MAX(plat)>20000;
```

Agregační funkce jsou silným nástrojem databázových serverů a jejich velkou předností je shodná funkcionalita ve všech databázových strojích. Další rozšíření těchto vzorů je možné dohledat v kapitole o nativních vzorech SQL.

6.3 Joining catalog

Příkazem JOIN se spojují dvě tabulky ve většině případů na základě primárního klíče hlavní tabulky a cizího klíče závislé tabulky. Je to jeden ze základů pochopení relačních databází a následně i jejich jazyka, SQL. Ve druhém patternu je popsán vzor, který provádí takové spojení i na jedné tabulce, čímž výsledný dotaz vrací stromovou strukturu.

6.3.1 Join pattern

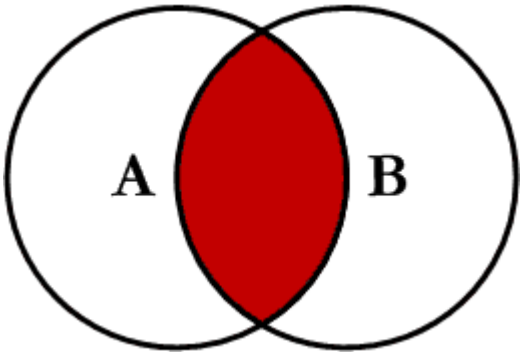
Variant, jak spojit dvě tabulky, je více. Z principu množinových diagramů lze jednoduše odvodit obecně platné vzory pro spojení.

Pro následující množinová schémata platí:

A – nosný sloupec hlavního datasetu (obvykle primary key).

B – odkazující sloupec závislého datasetu (obvykle foreign key).

INNER JOIN

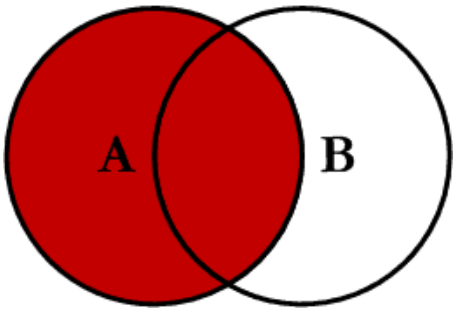


Obrázek 6-1 Množinový INNER JOIN (zdroj vlastní)

Nejjednodušší a také intuitivní, je spojení typu INNER. Spojuje všechny položky z A, které mají vazbu (položku) v B. Zápis v SQL je následující:

```
SELECT <sloupce>

FROM Tabulka_A A

INNER JOIN Tabulka_B B

ON A.Klic = B.CiziKlic
```

LEFT JOIN



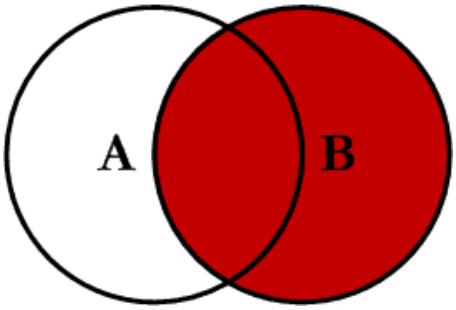
Obrázek 6-2 Množinový LEFT JOIN (zdroj vlastní)

Toto spojení vrátí všechny položky z A, včetně těch, které nemají žádnou vazbu v B.

```
SELECT <sloupce>

FROM Tabulka_A A

LEFT JOIN Tabulka_B B
```

```
ON A.Klic = B.CiziKlic
```

RIGHT JOIN

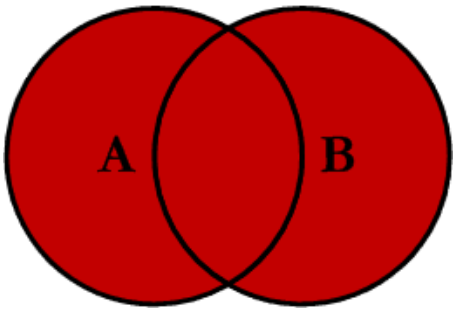


Obrázek 6-3 Množinový RIGHT JOIN (zdroj vlastní)

Pravý JOIN má podobný efekt jako levý, jenže obráceně. Vrátí všechny položky B, včetně těch, které nemají vazbu v A.

```
SELECT <sloupce>

FROM Tabulka_A A

RIGHT JOIN Tabulka_B B

ON A.Klic = B.CiziKlic
```

OUTER JOIN



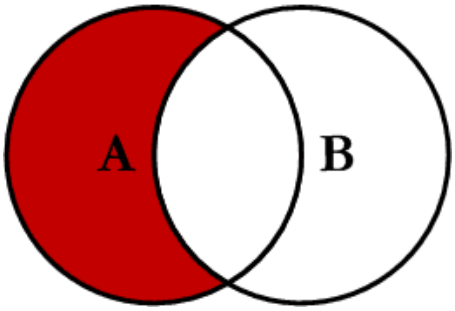
Obrázek 6-4 Množinový OUTER JOIN (zdroj vlastní)

Venkovní spojení, v některých SQL mutacích nazývané také FULL JOIN, vrací všechny položky A a všechny položky B bez ohledu na to, jestli mají mezi sebou vazby.

```
SELECT <sloupce>

FROM Tabulka_A A

FULL OUTER JOIN Tabulka_B B

ON A.Klic = B.CiziKlic
```

Pro některé požadavky lze použít EXCLUDING JOIN, což jsou doplněné JOINY o vyloučení položek, jejichž klíče jsou NULL. Z pohledu normalizace databáze by k těmto stavům dojít nemělo, přesto se dají použít např. v OLAP kostkách (decision cubes), které musí být v principu denormalizované.

LEFT EXCLUDING JOIN

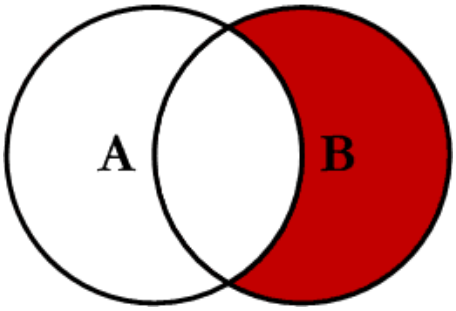


Obrázek 6-5 Množinový LEFT EXCLUDING JOIN (zdroj vlastní)

Tento vzor vyloučí s původního LEFT JOIN záznamy, které nemají hodnotu B.CiziKlic, resp. rovná se NULL.

```
SELECT <sloupce>

FROM Tabulka_A A

LEFT JOIN Tabulka_B B

ON A.Klic = B.CiziKlic

WHERE B.CiziKlic IS NULL
```

RIGHT EXCLUDING JOIN

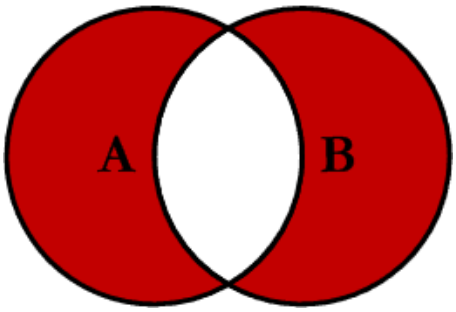


Obrázek 6-6 Množinový RIGHT EXCLUDING JOIN (zdroj vlastní)

Podobný případ, v obráceném gardu.

```
SELECT <sloupce>

FROM Tabulka_A A

RIGHT JOIN Tabulka_B B

ON A.Klic = B.CiziKlic

WHERE A.Klic IS NULL
```

OUTER EXCLUDING JOIN



Obrázek 6-7 Množinový OUTER EXCLUDING JOIN (zdroj vlastní)

Poslední ukázkou spojení je exkludovaný outer JOIN, kde ve výstupním setu zbydou pouze ty položky, které k sobě žádnou vazbu nemají.

```
SELECT <SELECT_list>

FROM Table_A A

FULL OUTER JOIN Table_B B

ON A.Key = B.Key

WHERE A.Key IS NULL OR B.Key IS NULL
```

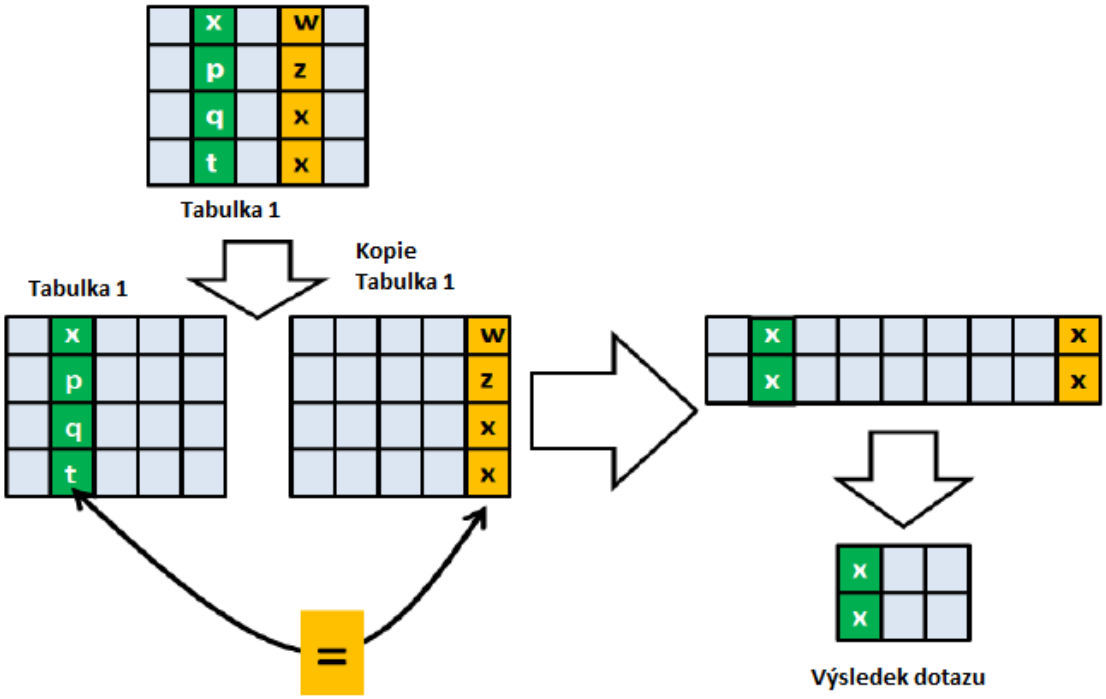
Vždy je dobré si v případě přechodu na jiný server odzkoušet všechny typy spojení tak, aby vracely jednoznačné hodnoty. Obzvláště netradiční rozšíření těchto příkazů může způsobit vracení odlišných hodnot, než na jaké byl vývojář dosud zvyklý. Použití připravených vzorových tabulek s takovými daty, které toto jednoznačné ověření umožní, je efektivní způsob, jak získat jistotu v novém prostředí.

6.3.2 Self-JOIN pattern

Dotaz na sebe-sama, tedy self-JOIN, je v množinovém vyjádření nepřehledný. Autor pro navrhnutí výukového vzoru použil standardní tabulkové vyjádření.

Základní myšlenkou vzoru je rekurzivní informace, kdy se v záznamu (řádku) vyskytuje nejen id řádku, ale také id řádku, na který je se třeba odvolat. Konkrétní použití

je v hierarchických strukturách modelujících podniková organizační schémata, nebo souborové systémy (adresář/soubor) atp.



Obrázek 6-8 Schéma rekurzivního SELF JOIN vzoru (zdroj [27])

Jak schéma ukazuje, principem je vytvoření kopie tabulky a následné spojení s ní samou. Vytvořit kopii neznamená ji skutečně fyzicky vytvořit, v tomto případě se jedná pouze o ukázkový příměr.

Při použití obecné SQL notace lze self-JOIN realizovat například takto:

```
SELECT distinct Bt_ID

FROM authorship ap1, authorship ap2

WHERE ap1.bt_id = ap2.bt_id

AND ap2.author_id <> ap1.author_id;
```

Dotaz vrátí seznam ID knih, které mají více než jednoho autora.

Tvorba self-JOIN dotazů patří mezi ty obtížnější – je třeba přestat myslet lineárně, sekvenčně, ale uvažovat stromově, vnořeně, jinými slovy rekurzivně. Obzvláště v případě aktivních rekurzivních případů typu update nebo delete je vhodné odzkoušet výsledný rekurzivní set výpisem před jeho aktivním spuštěním.

Jak je popsáno i v jiných částech této práce, konkrétní realizace self-JOIN není odkázáno pouze na serverově nezávislé SQL. Databázové servery právě pro tyto účely zavádějí vlastní datové typy (*hierarchy, hierarchyId)*, které umožňují nejen realizovat

samotnou hierarchii, ale i nabalovat celé objekty a vnořovat tak komplexní celky do jednotlivých úrovní. Zároveň mají i podpůrné prostředky jazyka SQL, které výsledné self-JOIN konstrukce zpřehledňují.

6.4 Shrnutí k návrhovým vzorům ve výuce

Při výuce SQL se uplatňují různé metody, relační algebrou počínaje a klasickým programováním při využití uložených procedur konče. Zajímavou alternativou, nebo spíše doplňkem výuky, pak mohou být právě návrhové vzory.

- Ověří znalosti získané v základních operacích typu spojení a výběru.
- Umožní zachycení hierarchie výuky, jinými slovy přehledovou následnost výkladu. Student se nemusí dostat do situace, kdy pochopí celou problematiku nespojitě.
- Celkově pak návrhové vzory rozšíří podmínky pro porozumění obsahu a v některých příkladech mohou ozřejmit látku studentům, kteří jsou na návrhové vzory uvyklí z jiných předmětů (programování, modelování atd.)
- Nabízí možnost tvorby dalších vzorů s tím, že jednak student fixuje svoje znalosti a zároveň je použije pro svůj další růst.

7 Návrh začlenění návrhových vzorů do UML metodiky a nástrojů

7.1 Syntaxe nativních vzorů

Pro využití nativních vzorů v modelovacím nástroji Enterprise Architect je třeba definovat jejich syntaxi – a to jak v případě přímého užití bez použití add-inu, nebo s jeho využitím. Navržená syntaxe by měla být srozumitelná jak návrháři, tak vývojáři a splňovat standardní podmínky jednoznačnosti a transparentnosti.

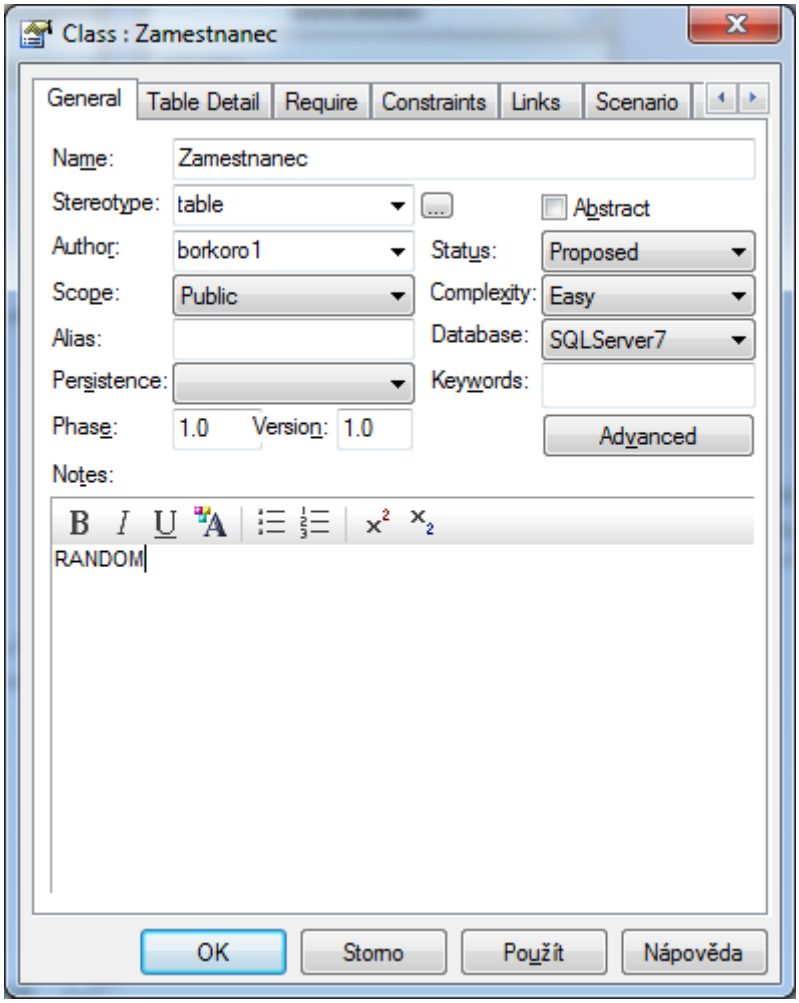
Návrh syntaxe vycházel z algebraické definice návrhových vzorů [26] s požadavkem na jednoduché a bezpečné syntaktické užití. Striktní překlad definic vzorů je v prvním případě realizován znalostní úrovní uživatelů vzorů (jak analytika, tak i vývojáře), ve druhém případě syntaktickou definiční tabulkou na straně nástroje EA.

Vzor	Syntaxe	Popis
COUNTING	COUNT describe	Četnosti – obecné, podmíněné, s argumentem
SUMMARY	CONDSUM	Podmíněná sumace
SEQUENCE	GENROW name	Tvorba sekvenčních šablon
DECOMP	DECOMP input > output1, output2,,,	Požadavek na dekompozici sloupce do atomizovaných sloupců
LISTAGGR	LISTAGGR input1,2,,> output	Složení více sloupců do jednoho (opačná transformace od DECOMP)
RANDOM	RANDOM source	Požadavek na náhodný záznam objektu (source)

ORDERING	CONDORDER condition	Požadavek na podmíněné setřídění kolekce dle zadané podmínky
RECURSE	RECURSE id, parentID	Definice atributů pro rekurzi – sloupec hlavní a sloupec dědičný
USERAGGR	USERAGGR name	Volání uživatelsky definované agregace
(UN)PIVOT	(UN)PIVOT columns	Transformace kolekce dle zadaných sloupců
SYMDIFF	SYMDIFF source1, source2	Požadavek na porovnání dvou zdrojů (tabulek)
HISTOGRAM	HISTOGRAM aggr, scale	Požadavek na triviální grafický histogram s definovanou agregací a výstupním měřítkem
SKYLINE	SKYLINE conditions	Definice srovnávacích podmínek pro nalezení optimálního výběru
DUPLREC	DUPLREC column1, column2 ….	Požadavek na zjištění duplicitních řádků dle definovaných sloupců

7.2 Využití Enterprise Architect bez nutnosti tvorby add-inu

Pro triviální zápis do modelu lze využít prostředků, které Enterprise Architect nabízí implicitně. Pomocí *Descriptions* lze do návrhu zavést alespoň pasivní informaci bez vazeb na okolní prvky modelu tak, aby vývojář tuto informaci obdržel. Takovou definici je možno zadat např. takto:



Obrázek 7-1 Ukázka přímé definice v Enterprise Architect (zdroj vlastní)

7.3 Komponentní prvek Design pattern SQL

Pro tvorbu modelových prvků je zapotřebí řešit problematiku jednoznačného vyjádření konkrétního požadavku. Ta se skládá ze dvou částí – grafického vyjádření v modelu a syntaktické definice upřesnění konkrétního návrhového vzoru.

Návrh grafického vyjádření je z pohledu této práce nepodstatný, podstatné je jeho funkcionální umístění – bude prvkem návrhového diagramu, jako závěrečné fáze konceptuálního návrhu. Základem bude kontejner s přidanými atributy tak, aby bylo možno konkrétní pattern upřesnit dle jejich popisu v kapitole 6.

Prvek Design pattern musí být provázán s odpovídající třídou a zároveň musí existovat plug-in, který bude schopen vygenerovat odpovídající SQL příkazy (resp. procedury, funkce atp.).

Pro fyzickou realizaci je možné využít datových vlastností EA – konkrétně faktu, že je vysteven nad databází MSAccess. Základní funkcionalitu zajistí transformační tabulka *plgSQLTransform*, jejíž struktura a vytvoření např. pomocí VBScriptu je následující:

(tabulka s projektem byla přejmenována na typ mdb)

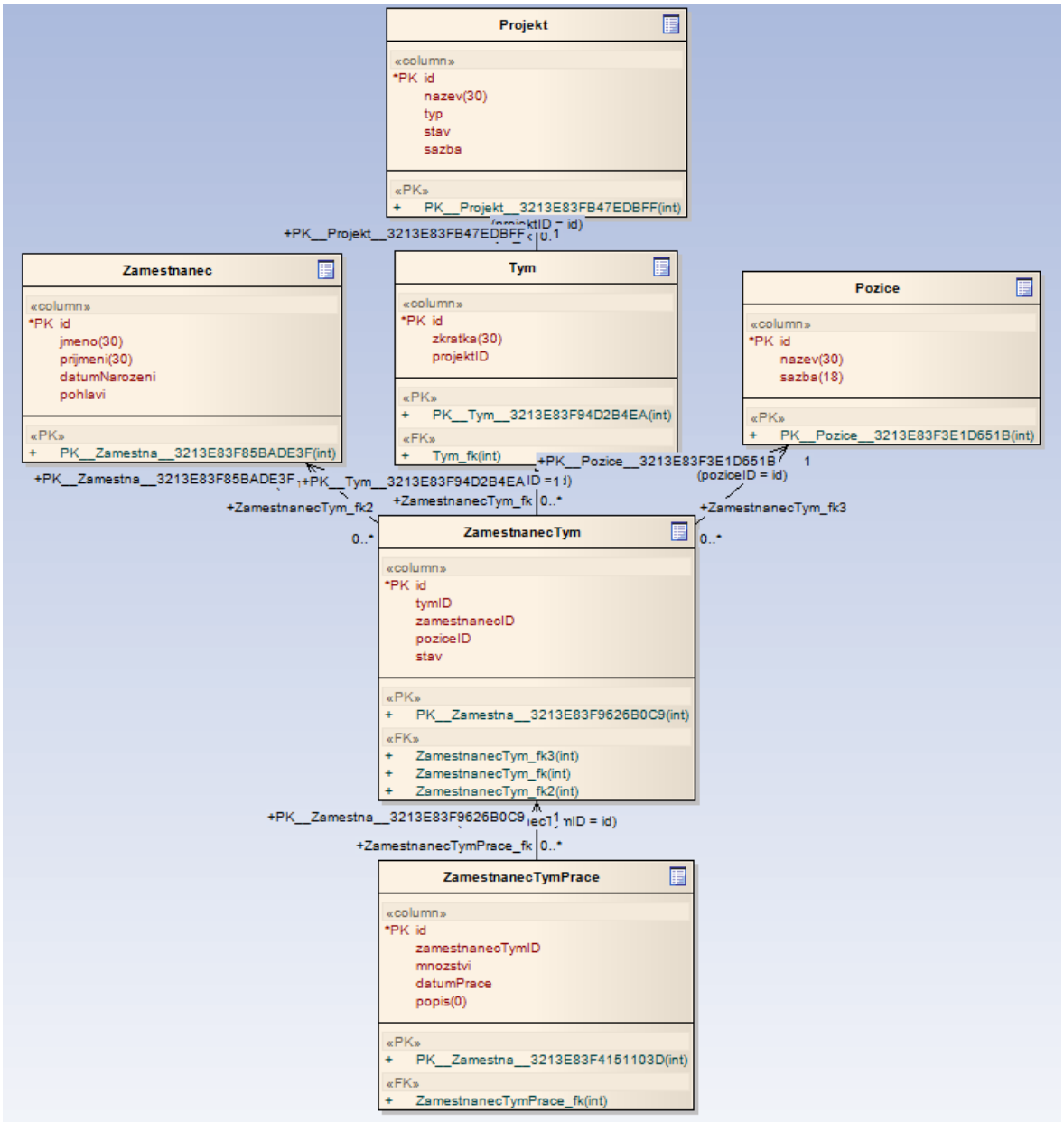
```
Sub CreateTableplgSQLTransform()

      Dim dbs As Database

     ' Zde je třeba doplnit konkretni connection string pro EA.mdb

    Set dbs = OpenDatabase("EA.mdb")

     dbs.Execute "CREATE TABLE plgSQLTransform " _

        & "(Code CHAR, Define CHAR );"

     dbs.Close

 End Sub
```

Pak je zapotřebí naplnit tabulku odpovídajícími pravidly, která jsou složena z kódového názvu vzoru a jeho realizační definice. V rozsahu, uvedeném v této práci, je možno použít tento importovací vzor:

```
INSERT INTO plgSQLTransform (code, define) VALUES

(

  Dim dbs As Database

     ' Zde je třeba doplnit konkretni connection string pro EA.mdb

    Set dbs = OpenDatabase("EA.mdb")

     dbs.Execute "CREATE TABLE plgSQLTransform " _

        & "(Code CHAR, Define CHAR );"

     dbs.Close
```

Realizace konkrétního add-inu je z programátorského hlediska triviální, technický postup je ukázán v příloze 6 této práce. Jakmile add-in najde v popisu EA objektu klíčové slovo návrhového vzoru, rozklíčuje jej podle zadaných pravidel a doplní do požadovaného výstupu.

Celkový efekt je pro práci nejen analytika, ale i koncového řešitele v tom, že jsou přesněji požadované datové vzory. Bylo by dobré tuto možnost implementovat do EA jako nadstavbu se zahrnutím vlastní třídy, bohužel tuto možnost EA neposkytuje.



Obrázek 7-2 Schéma ukázkové databáze (zdroj vlastní)

8 Zhodnocení použití návrhových vzorů

8.1 Zhodnocení z pohledu vývoje

Z pohledu samotného vývojáře umožňují návrhové vzory několik zásadních věcí. Jejich stručná charaketeristika je následující:

- Bezpečnost vývoje na základě jednoduchých, ověřených a transparentních vzorů.
- Čitelnost pro jiné pracovníky, než pouze autory vzorů a nebo pro pracovníky z jiných vývojářských úrovní. Zajištění komunikace mezi různými vrstvami vývoje.
- Snadnou nahraditelnost a rozšiřitelnost v případě získání nových zkušeností, platforem atp.

8.1.1 Zhodnocení design pattern setu pro optimální realizaci datového úložiště

Optimální realizace datového úložiště je v současné době velmi žádaná problematika. Jde o to objektivně posoudit, zda je aktuální řešení úložiště optimální, případně navrhnout jinou formu úložiště. Konkrétním problémem, před kterým stojí firmy, realizující provoz na velkých datech (nezaměňovat s termínem bigData), je doporučení klientovi, jakou formu datového úložiště zvolit - relační databázi, sloupcovou, bigData atd.

Kompletní řešení problematiky je úkol pro jinou práci, autor se v této kapitole soustředil pouze na možnost využití datových vzorů pro zhodnocení stávajícího úložiště ve smyslu jeho optimálního sestavení.

8.1.1.1 Teoretické podklady k záměru

Soustředit se na zhodnocení správného, resp. optimálního uložení dat znamená velmi odbornou práci s několikaletou zkušeností. Přesto lze alespoň obecné závěry stanovit z těchto několika bodů:

- Vytvořením vhodných sad příkazů a jejich implementací lze postihnout základní charakteristiky chování databáze

- Rozumně uspořádaná data se budou při opakované implementaci těchto sad chovat ve svých odezvách jako normálně rozložená. Tuto normalitu naruší data v provozu, v okamžiku, kdy je nutno zahrnout přístupy uživatelů a různorodost požadavků, není možné takové rozdělení předpokládat. Přesto je možné pro srovnání kvality uspořádání i volby cílového úložiště použít takové sady i na pasivní data (data mimo reálný provoz, pouze pod zátěží testovacích sad).

- Výsledkem budou přehledové tabulky, jejichž hodnoty budou porovnány s normálním rozdělením – tam, kde se budou hodnoty blížit více, je možno předpokládat kvalitnější uspořádání.

8.1.1.2 Návrh sady vzorů pro danou problematiku

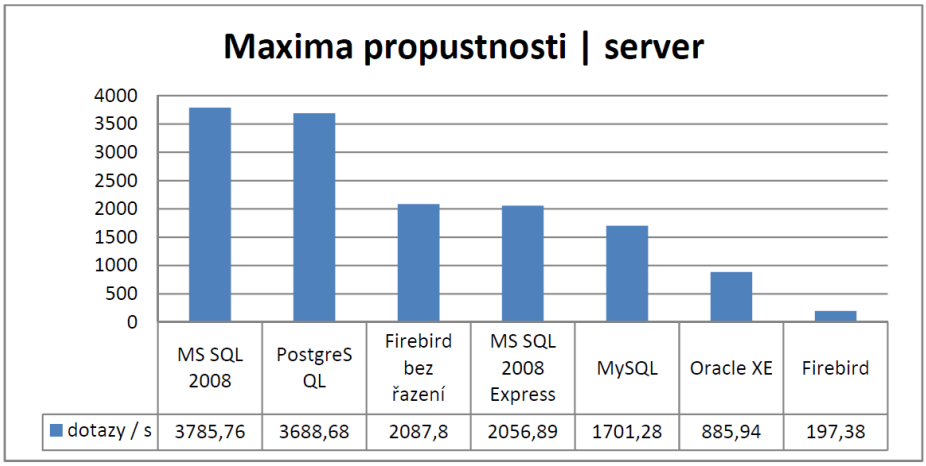
Situace, kdy je třeba porovnat výkonnost databázového řešení, je v reálném životě poměrně častá. Ať se jedná o rozhodnutí o typu datového úložiště nebo konkrétního databázového serveru. Pro takové záležitosti je dobré mít k dispozici testovací sadu příkazů, které často vycházejí z výše uvedených vzorů.

Zásadním rozdělením jsou sady pro získávání záznamů (realizované téměř vždy nějakou formou příkazů SELECT) a aktualizační sady. V každé skupině je pak nutné připravit lineární sady (sady, které jsou soustředěny na práce s jednou tabulkou) a kombinované, které budou mít složitější strukturu, pokrývající vazební realizace v aplikaci.

Pro řešení lineárních výběrových sad je dobré využít vzor RANDOM. Výběr náhodného záznamu, opakovaný v řádech několika tisíc volání je základním kriteriem pro hodnocení výkonnosti celé realizace. Jinými slovy řečeno, pokud nebudou fungovat lineární testy, nemá smysl pokoušet kombinované.

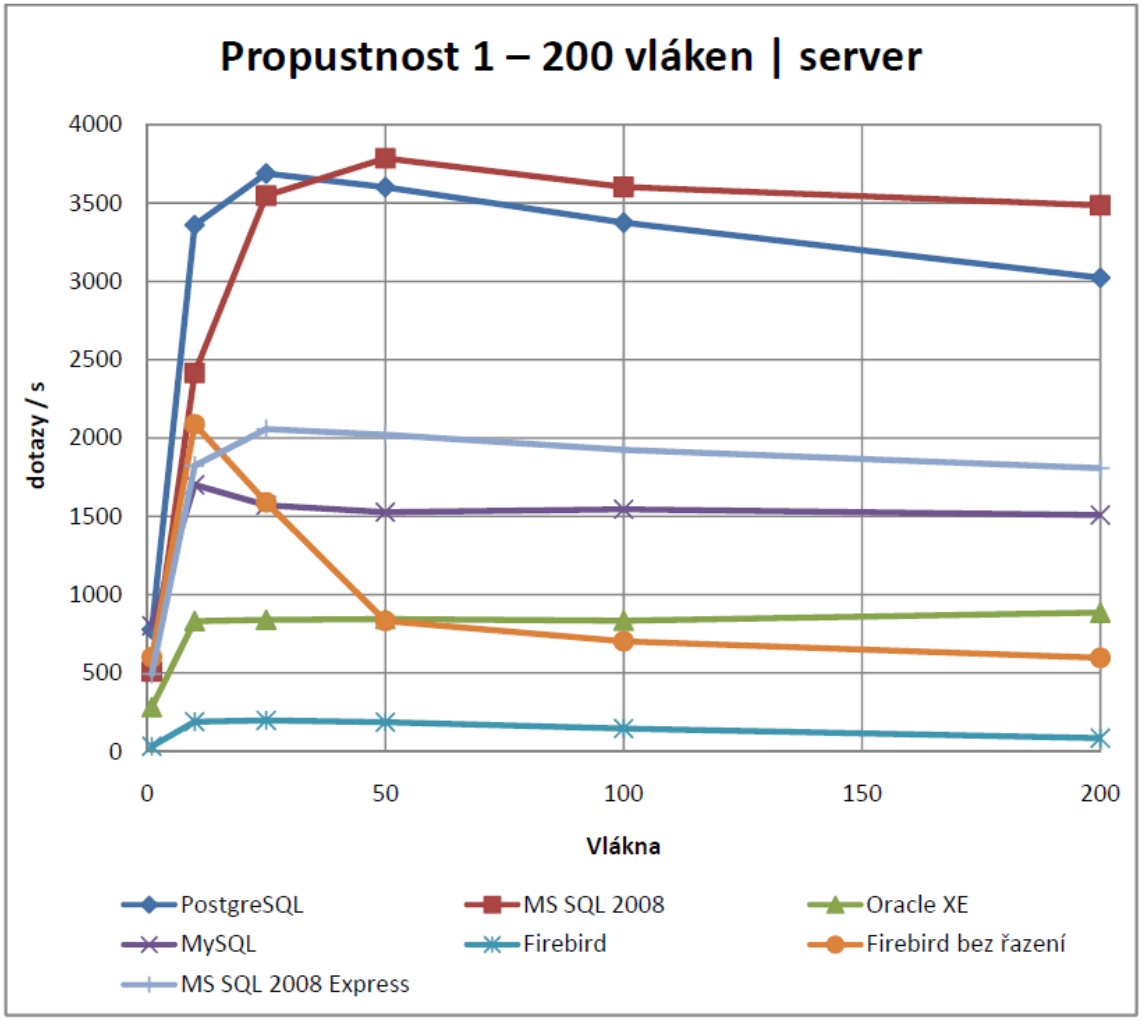
Základním záměrem lineárních testů je ověřit správnost volby databázového stroje. Každý má totiž jinou fyzickou realizaci a zásadní operace řeší jinak. Například uložení struktur, položek typu LOB (BLOB) a celkové řešení CRUD operací. Jen pro představu, co je výsledkem takového testování, je možné uvést výsledky srovnání z práce Košárka [28], který se problematikou porovnání včetně využití a definice testovacích sad zabýval podrobně.

Tabulka 8-1 Srovnávací tabulka maxim propustnosti DB serverů – zdroj [28]



Podobným způsobem lze připravit testovací sady s provázáním na počet užitých vláken, což vytváří reálnější provozní prostředí databáze.

Tabulka 8-2 Srovnávací tabulka s počtem použitých vláken – zdroj [28]



Při standardním postupu testování se nejdříve provedou lineární výběrové testy. Obecná syntaxe takového příkazu může vypadat takto:

SELECT top 1 cols FROM tab ORDER BY newid()

V testovací sadě bude použit pro páteřní tabulky systému a velké číselníky.

Druhým krokem bývají nelineární výběrové testy, což znamená přípravu takových příkazů, které se aplikačně budou používat nejčastěji. Může to být např. vyhledání faktury dle zadaných podmínek, nebo jakéhokoliv jiného páteřního dokladu aplikace. Ukázkový příkaz takové sady pak může vypadat takto:

```
SELECT top 1 cols FROM tab1

Inner JOIN tab2 on tab2.id = tab1.idtab1

…….

Order by newid()
```

Pro realizaci aktualizačních sad (lineárních i nelineárních) je možné narazit na problém nemožnosti měnit data v databázi. Důvodem nemusí být jen nepřístupnost z pozice testera (což by mělo být vyloučené, ale bohužel možné), ale také ztráta konzistence při náhodných aktualizacích. Pak je dobré doplnit do odpovídajících tabulek testovací aktualizační sloupce, jejichž použití není pro databázi významné a jejich následné odstranění není technický problém. Nevýhoda tohoto řešení je aktualizační neobsazenost podstatných atributů a tím pádem ne zcela objektivní výsledky testů – může dojít k neodhalení problematických zátěží.

Pro zátěžová srovnání lze použít i vzory, u kterých byla poznámka, že nejsou ideální pro běžné použití pro svoji náročnost. Takže například SYMETRICKÁ DIFERENCE, PIVOT, HISTOGRAM, DEKOMPOZICE, AGREGACE mohou být vhodně použity – cílová datová úložiště budou mít s jejich hromadným zpracováním velmi pravděpodobně výrázné výkonnostní diference.

8.1.2 Zhodnocení design pattern v budoucích směrech vývoje – kvantové databáze

8.1.2.1 Kvantové databáze

Problematika kvantových počítačů a z nich odvozovaných témat se jeví v současné době stále jako více teoretická, nežli přímo použitelná. Navzdory prototypům jako je D-Wave, panuje obecné povědomí, že o skutečně reálném kvantovém počítači nelze v současné době hovořit. Jedná se stále o kvantové simulátory, což má za následek, že skutečně efektivních kvantových algoritmů je k dispozici velmi malé množství. Navíc je

problém i v tom, že např. Shorův algoritmus D-Wave díky pouhé simulaci nedokáže v rozumném čase řešit.

Přesto existují práce (Paul Cockshot), které se již dnes zabývají tématy relačních databází v kvantovém pojetí. Pokud má autor ve své práci najít její dopad i v budoucnosti, bude dobré uvažovat i v tomto přesahu.

Definice kvantového počítače je primárně stavěná na využití tzv. qubitů – pro jednoduchou představu stačí prohlásit, že se jedná o bity, u kterých do doby pozorování není známo, jestli jsou ve stavu 0 nebo 1. Některé kvantové algoritmy dosahují exponenciálně vyšších výsledků než standardní postupy, proto se nabízí otázka, je-li možno uvažovat jejich využití i při zpracování dat. Pokud ano, má smysl přemýšlet o použití návrhových vzorů i v této oblasti.

V pracech Sudip Roy, Lucja Kot, Christoph Koch [36] jsou k dispozici matematické základy použití kvantových databází – od jejich definic, až po závěry z nich plynoucí. Cockshot již dokonce naznačuje implementace relačních databází. Jedná se o čistě matematické pojetí, jehož rozbor není účelem této práce. Přesto se dají některé poznatky zužitkovat i ve vztahu k návrhovým vzorům.

Autor tuto problematiku dále zpracovává ve svých zatím nepublikovaných pracech, protože z jeho pohledu se jedná o tématiku budoucího vývoje a je třeba se jí věnovat nejen v teoretické rovině. Zmiňované články popisují nejen problematiku návrhových vzorů, ale také představu kvantové databáze jako takové, včetně jejího interpretačního rozhraní a budoucích uživatelských aplikací.

8.1.2.2 Návrh realizace datových vzorů pro kvantové databáze

V současné době lze jako návrhové vzory při řešení databázové problematiky na úrovni kvantových počítačů stanovit na základě kvantových algoritmů. Splňují základní požadované operace pro práci s daty a pokud bude v dohledné době realizována nejen nutná technická základna, ale také odpovídající databázový stroj, budou mít tyto algoritmy smysl i pro analytiky – tím budou splňovat podmínku využití vzorů v etapě modelování výsledného návrhu.

Základními vzory jsou tak algoritmy, řešící vyhledávání v nestrukturované entitě, porovnání hledaného řetězce a kvazi-spojování dat do lineárních schémat – joining.

8.1.2.2.1 Groverův algoritmus

Kvantový Groverův algoritmus polynomiálně snižuje rychlost vyhledání prvku v nestrukturovaném poli. Na dohledání požadovaného prvku mu stačí z n prvků $\sqrt{n}$ kroků. Praktickou realizaci lze popsat jako hledání účastníka telefonní sítě podle jeho čísla. Při standardním vyhledávání je třeba projít v nejhorším případě všechny položky. V případě

realizace kvantového databázového stroje by tento algoritmus byl zcela jistě kandidátem na modelové zadání požadovaného procesu.

8.1.2.2.2 Porovnání řetězců

Jinými slovy lze tento algoritmus pojmenovat jako pattern matching, což lépe vystihuje jeho zaměření. Jde o to, nalézt v obecné entitě subentitu, která odpovídá hledanému patternu. Na rozdíl od klasických algoritmů je kvantový rychlejší dokonce superpolynomiálně – tzn. exponenciálně, případně ještě více rostoucí rychlost. Navazuje na Groverův algoritmus, takže v základních podmínkách má i stejný růst. V okamžiku, kdy budeme uvažovat entitní matice typu $m\times m\times\ldots\times m$ bloků s $n\times n\times\ldots\times n$ poli dané struktury, dojdeme právě k superpolynomiálnímu snížení celkové času na řešení. Tím je možno realizovat vyhledávání, které je v dnešních podmínkách časově nemožné. Využití tohoto vzoru je velmi široké, nejen v datařině, ale i v kryptografii a nebo v multimediálních a nestrukturovaných entitách. Montanaro zde http://arxiv.org/abs/1408.1816 uvádí příklad konkrétního algoritmu, který dosahuje průměrného hledání náhodného textu v textových entitách $O\tilde{}((n/m)^{d/2}2^{O(d^{3/2}\log m\sqrt{})})$. (n,m z předchozí definice, d je dimenze hledaného textu).

8.1.2.2.3 Lineární systémy

Poslední ukázkový vzor lze matematicky vyjádřit takto: mějme matici $n\times n$ A a popsaný vektor *b*. Hledá se obecná vlastnost *f*(A)b pro nějakou vyčíslitelnou funkci *f*. Předpokládejme, že A je matice Hermitovského typu (diagonálně sdružené prvky jsou komplexně sdružené) s *O* nenulovými prvky v každém řádku. Pokud tuto matematickou formulaci převedeme do databáze, jedná se o řešení požadovaného výběru s předpokladem neprázdných řádků. Funkcí *f* pak můžeme rozumět daný příkaz a výsledný vektor je řešení na matici A, resp. výsledkem funkce je jeho nalezení v matici. Zajímavostí tohoto algoritmu je to, že výsledný čas není závislý na rozsahu matice A, ale na počtu O – čili výsledných nenulových prvků.

Databázovou implementací by tak bylo možno řešit libovolná spojení v superpolynomiálním čase, jak dokazuje práce http://arxiv.org/abs/1010.4458. Jen jako poznámku lze uvést, že další prostory pro využití tohoto algoritmu jsou v oborech řešení lineárních diferenciálních rovnic, strojovém učení, stanovení odporu počítačových sítí apod.

8.2 Ekonomické zhodnocení

Ekonomických přínosů realizace návrhových vzorů obecně, nejen z pohledu SQL, je velké množství. Lze dokonce tvrdit, že i v ekonomice samotné se návrhové vzory

vyskytují, příkladem mohou být články [23]. Mají společná kritéria i zdůvodnění svojí existence, která lze shrnout do následujících bodů:

- Návrhový vzor snižuje při svém použití náklady na vývoj – zkracuje dobu realizace a zajišťuje použití optimálních řešení jak z pohledu výkonu, tak i spolehlivosti.

- Snižuje náklady na odbornost jejich uživatelů. Vývojář, který vzor použije, nemusí být nutně tak kvalifikovaný (rozuměj drahý), jako ten, kdo je vytváří.

- Nabízí vyšší nahraditelnost a snadnou mentální přenositelnost.

- Zvyšuje efektivitu zaškolení pracovníků, stejně tak umožňuje transparentnost ověření znalostí přijímaného pracovníka.

Zajímavou problematikou, která spojuje ekonomický přístup a vývojářská řešení, jsou tzv. side-effects, čili externality. Ať již v ekonomickém uvažování nebo v tvorbě IS jsou to supermodelové atributy, buď s pozitivním nebo častěji s negativním dopadem.

Použití návrhových vzorů umožní v důsledku v obou těchto prostorech side-effects zmírnit, nebo je dokonce eliminovat. Návrhový vzor nemá smysl použít s dovětkem, že navzdory svým kvalitám přináší i něco škodlivého – smysl a význam návrhových vzorů není v rychlosti jejich prvotní realizace, ale právě v jejich důsledném ověření a transparentním nasazení.

9 Testování návrhových vzorů

Vytvořené návrhové vzory je třeba důkladně otestovat. Jak již bylo řečeno, jsou základním kamenem a opěrným prvkem nejen modelu, ale celé aplikace, proto musí být navrženy a realizovány s maximální odpovědností. K ověření vzorů a jejich praktické implementace existuje několik typů testů.

Vždy je nutné, aby ověřené vzory obsahovaly kvalitní dokumentaci – nejednoznačnost výkladu a následného použití by mohla způsobit nečekané problémy. Ve výsledku je tak od prvotního nápadu ke konečné realizaci dlouhá a důsledná cesta, která se však při vývoji IS vyplatí.

9.1 Jednotkové testy

Ověření na úrovni základních jednotek – tříd, realizují jednotkové testy. Jejich metodika je stavěná na přímém použití vzoru ve třídě, která je následně prověřena jak do funkčnosti, tak i do výkonu.

V rámci této práce autor provedl ověření abstraktního vzor ABSTRACT FACTORY a výsledky shrnul v následujících tabulkách:

Test metody GetConnection

Vstup	Předpoklad	Test	OK?
<add name="connStr" connectionString="Data Source=cronus;Initial Catalog=ISU3;Trusted_Connection=Yes;" providerName="System.Data.SqlClient" />	OK	OK	OK
<add name="connStr" connectionString="Data Source=cronus;Initial Catalog=ISU3;Trusted_Connection=Yes;" providerName="System.Data.Sql" />	!OK	!OK	!OK
<add name="connStr" connectionString="Data Source=cronus;Initial Catalog=ISU3;Trusted_Connection=Yes;" providerName="FirebirdSQL.Data.SqlClient" />	OK	OK	OK

Test metody GetTable

Vstup	Předpoklad	Test	OK?
DataTable DT = GetTable(„SELECT * FROM zamestnanec“, null);	OK	OK	OK
DataTable DT = GetTable(„SELECT * FROM zamestnanec“, dp);	OK	OK	OK
DataTable DT = GetTable(„SELECT * FROM zamestnanec“, „@id = 3“);	!OK	!OK	!OK

9.2 Integrační testy

Následně je třeba ověřit postupnou konstrukci celého systému. K tomu slouží integrační testy, které lze rozdělit do dvou skupin:

- Strukturální testování – tzv. white-box testing. Na rozdíl od principu black-box, kde vývojář netuší, jakým způsobem se dostává od vstupu k požadovanému výstupu, je v tomto testování nutné ověřit všechny možné větve realizovaného kódu. Součástí těchto testů bývá testovací plán, který zajišťuje průchod celým kódem.

- Inkrementální testování – je v zásadě postupná kompletace výsledného systému. Ověří se základní moduly, k nim se přidávají další a tím se transparentně zachytí případné chyby v jejich návaznostech.

Tento způsob testování nebyl v práci uplatněn, je uveden pouze jako nezbytná součást implementace popisovaných vzorů při kompletaci řešení. Přesto je možné naznačit realizaci strukturálního testování za pomocí plánu:

Předpokládejme ověření funkčnosti a výkonnosti modulu pro autentizaci a autorizaci uživatele v IS. Plán strukturálního testování pak bude vypadat zhruba takto:

- Předpokládáme bezchybnou a ověřenou konektivitu k prostředkům.

- Autentizace otestována podle požadavků zadavatele – formulářová, podle OS, podle LDAP atp. V jejím rámci je třeba ověřit chování při zadání neexistujícího účtu, vadného hesla, v případě použití i časové omezení přístupu.

- Autorizační modul pak prověří všechny možnosti uživatelské autorizace tak, aby se zjistily případné nedostatky ve všech možných situacích:
    - Správné přiřazení funkčních složek IS (menu, reporty atd.).
    - Totéž pro ověření kombinovaných rolí.
    - Správnou funkčnost uživatelských formulářů – nejen jejich dostupnost, ale také odpovídající funkcionalitu jednotlivých komponent, závislých na správné autorizaci.
    - Ověření aktivních prvků – CRUD operace proti business vrstvě, tvorbu výstupů a přístup k externím zdrojům vůbec.

Inkrementální testy fungují principem shora dolů – od prezentační vrstvy, přes business až po data layer. Pokud jsou použity pouze na datové vrstvě, což odpovídá požadavku této práce, je třeba postupovat od tabulek, přes triggery až k uloženým procedurám.

9.3 Validační testy

Validační testy přicházejí na řadu po integračních testech. Fungují na principu black-box, kdy tester je odtržen od vnitřní implementace. Výsledkem těchto testů je ověření požadavků zákazníka.

9.4 Systémové testy

Systémové testy se soustřeďují na spolupráci vrstev, ať je jejich koncepce jakákoliv. Je třeba ověřit jejich stabilitu a interlayerovou spolehlivost. V případě testování prvků, které vycházejí z návrhových vzorů SQL, se bude jednat o business a data vrstvy.

Zajímavým případem, který vychází z této práce, je systémové testování sad pro ověření výkonnosti databázového řešení, které je popsáno v příkladech užití návrhových vzorů. Zde tyto dvě vrstvy budou tvořit sady vzorů typu RANDOM a pak následné aktualizační sady. Jejich srovnáním v testu bude možné odhalit výkonnostní problémy a zavést nutná opatření.

Vždy je třeba dodržet posloupnost kategorií testů – jejich princip je postaven na postupné spolehlivosti, jinak ztrácejí smysl. Druhým doporučením je, aby testovací tým nebyl tvořen autory kódu – většinou nemají schopnost nadhledu a ačkoliv testy projdou, zůstane nezachyceno to, co koncový uživatel zachytí leckdy při prvním použití aplikace.

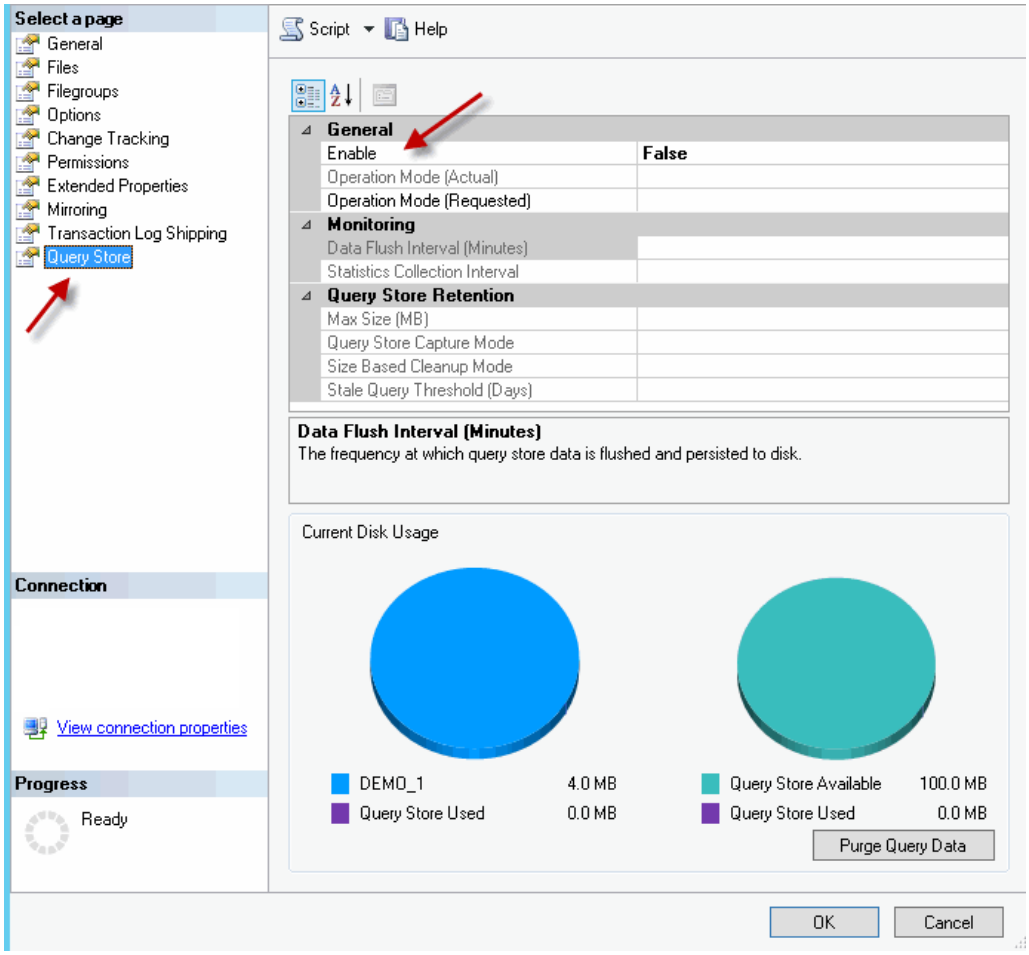
9.5 Query Store

Zajímavým řešením pro využití testování navržených vzorů je Query Store ve verzi 2016 MSSQL. Jedná se o nový nástroj databázové serveru firmy Microsoft, který slouží pro správu sql dotazů a plánů. Ve vztahu k návrhovým vzorům ho lze použít několika způsoby:

- Testování návrhových vzorů – jejich validitu, výkon a parametrizace.
- Realizaci plánů optimalizace a jejich evidenci.
- Komplexní sledování výkonů, včetně sledování v závislostech a v časových řadách při aplikačním použití návrhových vzorů.

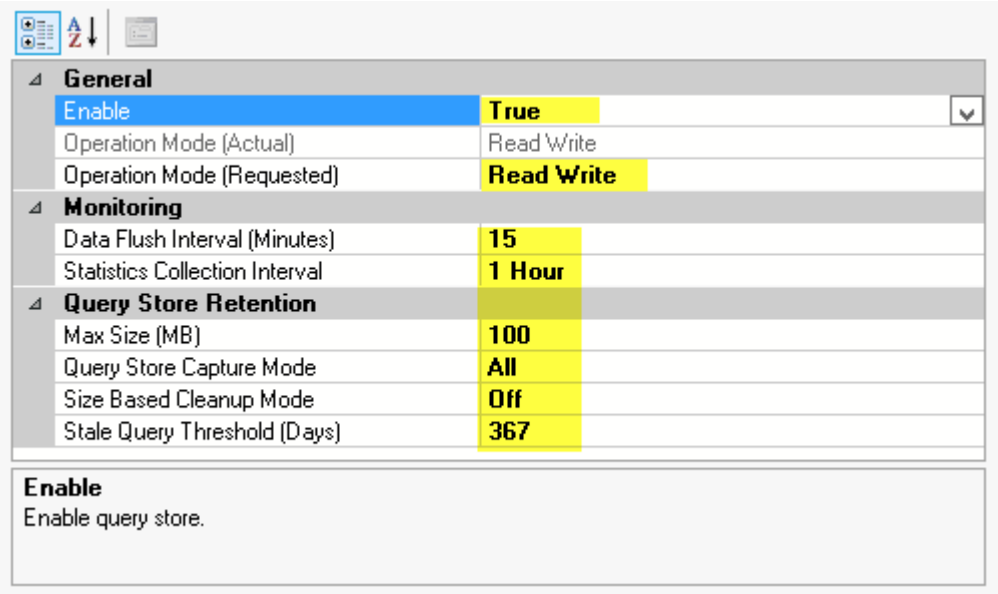
Query Store dokáže pokrýt všechny uvedené stupně testů, alespoň pokud lze vyjít z nabízených možností. MSSQL 2016 je v této době pouze CTP verze, existuje tedy možnost, že některé současné funkcionality nebudou v ostré verzi serveru uvolněny.

Query Store je třeba nad danou databází povolit:



Obrázek 9-1 Definice Query Store v Management Studiu (zdroj vlastní)

a následně nakonfigurovat:



Obrázek 9-2 Konfigurace Query Store (zdroj vlastní)

Veškeré činnosti lze provést i nevizuálně, s použitím následujících příkazů:

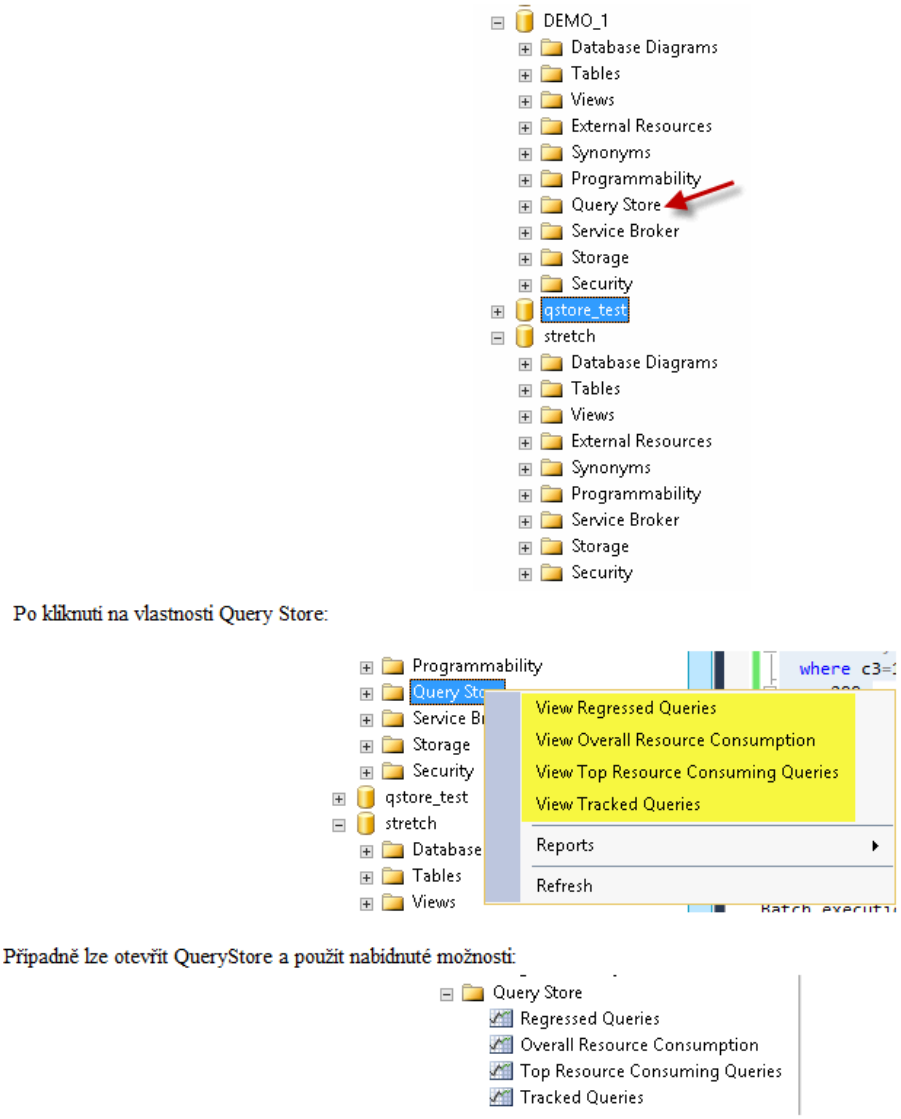
```
ALTER DATABASE [DEMO_1] SET QUERY_STORE = ON

GO

ALTER DATABASE [DEMO_1]

SET QUERY_STORE (OPERATION_MODE = READ_ONLY,

                                    CLEANUP_POLICY   =   (STALE_QUERY_THRESHOLD_DAYS   =
367),

                                    DATA_FLUSH_INTERVAL_SECONDS = 900,

                                    INTERVAL_LENGTH_MINUTES = 60,

                                    MAX_STORAGE_SIZE_MB = 100,

                                    QUERY_CAPTURE_MODE = AUTO,

                                    SIZE_BASED_CLEANUP_MODE = AUTO)

GO
```

Autor nechce v této práci popisovat konkrétní příkazy, jednak z důvodu jejich exaktní čitelnosti a jednak z důvodu, že se jedná o CTP verzi a v případě přesných formulací by bylo dobré počkat na verzi ostrou. Přesto je vidět myšlenkový záměr nastavení Query Store – definovat rozsah a podmínky ukládání, testování a vyhodnocování SQL dotazů.

Přehled všech doplněných událostí souvisejících s Query Store v CTP2:

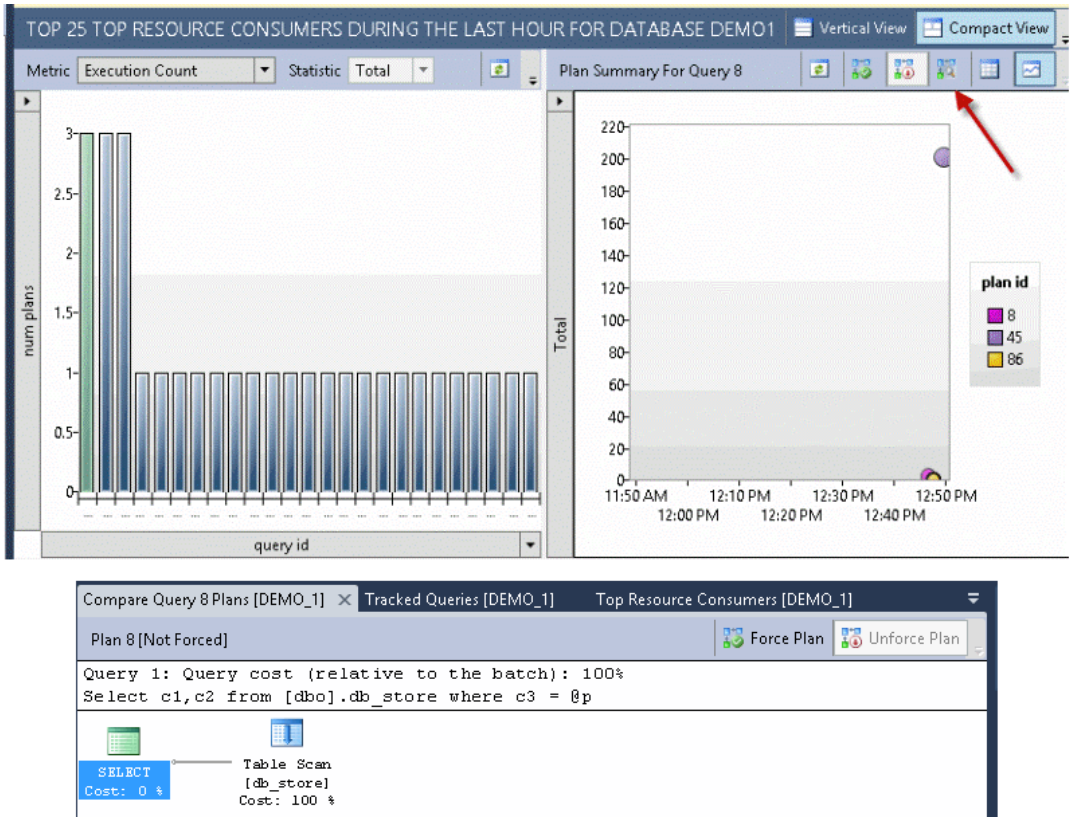
```
query_store_background_task_persist_started - Fired if the background task for Query Store data persistence started execution

query_store_background_task_persist_finished - Fired if the background task for Query Store data persistence is completed successfully

query_store_load_started - Fired WHEN query store load is started

query_store_db_data_structs_not_released - Fired if Query Store data structures are not released WHEN feature is turned OFF.

query_store_db_diagnostics - Periodically fired with Query Store diagnostics on database level.

query_store_db_settings_changed - Fired WHEN Query Store settings are changed.

query_store_db_whitelisting_changed - Fired WHEN Query Store database whitelisting state is changed.

query_store_global_mem_obj_size_kb - Periodically fired with Query Store global memory object size.

query_store_size_retention_cleanup_started - Fired WHEN size retention policy clean-up task is started.

query_store_size_retention_cleanup_finished - Fired WHEN size retention policy clean-up task is finished.

query_store_size_retention_cleanup_skipped - Fired WHEN starting of size retention policy clean-up task is skipped because its minimum repeating period did not pass yet.

query_store_size_retention_query_deleted - Fired WHEN size based retention policy deletes a query FROM Query Store.

query_store_size_retention_plan_cost - Fired WHEN eviction cost is calculated for the plan.

query_store_size_retention_query_cost - Fired WHEN query eviction cost is calculated for the query.

query_store_generate_showplan_failure - Fired WHEN Query Store failed to store a query plan because the showplan generation failed.

query_store_capture_policy_evaluate - Fired WHEN the capture policy is evaluated for a query.

query_store_capture_policy_start_capture - Fired WHEN an UNDECIDED query is transitioning to CAPTURED.

query_store_capture_policy_abort_capture - Fired WHEN an UNDECIDED query failed to transition to CAPTURED.

query_store_schema_consistency_check_failure - Fired WHEN the Query Store schema consistency check failed.
```

Tyto nové události jsou schopny zachycovat, vyhodnocovat a optimalizovat sledované SQL dotazy a procesy vůbec. Query Store je propojeno přímo s novým SQL Management Studiem:



Obrázek 9-3 Postup užití Query Store (zdroj vlastní)

Query Store podstatně rozšiřuje funkcionality SQL serveru a zároveň nabízí vývojářům širokou škálu prostředků nejen pro tvorbu kvalitních aplikací, ale také nosných návrhových vzorů pro další použití. Monitorovací prostředí bylo doplněno o funkce vyplývající z nových událostí a je propojeno se stávajícím Query Planningem tak, že společně mohou poskytovat oboustrannou spolupráci – jak při ladění existujících, tak i při návrhu předpokládaných řešení.



Obrázek 9-4 Ukázka ladění dotazů s využitím Query Planning (zdroj vlastní)

10 Shrnutí výsledků

Práce vycházela z premisy, že datová úložiště nabízejí podstatně více, než standardní vývojář dnes vůbec použije. Pro shrnutí výsledků je možno porovnat dosažení cílů práce:

- Propojení analytického návrhu s možnostmi datového úložiště. Návrhové vzory umožňují využít jejich možností bez ohledu na konkrétní specifikace. Nabízejí konkrétní operace, které je výkonově vhodné realizovat na datové vrstvě.

- Realizaci nestandardních aplikací za použití návrhových vzorů SQL. Vývojářská práce není jen o sestavení IS, ale také o ETL, servisních, aktualizačních a hromadných operacích, kde žádná paradigmata příliš nefungují. Příkladem mohou být hromadná plnění daty, kde veškeré vazební aplikační funkcionality (klíče, omezení atp.) dokáží velmi zkomplikovat celý proces plnění.

- Práce poukázala na nutnost pochopení vlastností konkrétního datového úložiště. V oblasti relačních dat je potřebné znát fyzické vlastnosti serverů, není možné spoléhat na generické a všeobjímající postupy business vrstvy a ORM. Standardní IS má stále ještě pořád daleko do NoSQL databází nebo problematiky bigData.

- Shrnutí všech zásadních typů návrhových vzorů umožnilo pohled na datová úložiště nejen jako uzavřenou a v důsledku funkcionálně nevyužitou entitu. Z hlediska vývoje práce ukazuje její význam i ve vzorech aplikačních, abstraktních a také propojení nativních vzorů nejen s výslednou aplikací, ale také s analytickými celky při vývoji.

- Práce poukázala na to, že v současné době se problematika konceptuálního modelu začíná projevovat i v metodice. Začínají se ukazovat nevýhody vývojářského komfortu, kterému se celé odvětví věnovalo posledních patnáct, dvacet let, a to na úkor výsledného výkonu a bohužel i na úkor toho, co klient původně požadoval.

- Práce se pokusila mj. zkompletovat konkrétní nativní vzory tak, aby ukázala možnosti datových úložišť a nabídla tak efektivnější výslednou realizaci. Obsahuje katalog nejen návrhových vzorů, ale také antivzorů, jako popis původně dobrých postupů, které se v praxi ukázaly jako nedobré.

Poslední bod závěru poukazuje na velmi důležitou a potřebnou vlastnost nejen v IT. Nikdy není žádná metodika kompletní a dokonalá. Vývojář (analytik), který ustrne v šabloně jakéhokoliv paradigmatu, se připraví o to nejpodstatnější – čerstvý rozum a zdravé uvažování.

11 Závěry a doporučení

Jak bylo již v práci zmíněno, obecně se lze s návrhovými vzory setkat patrně v jakémkoliv oboru. To, co v IT oblasti nehrozí, je problematika tzv. *sands*, čili písečků – místo kvalitních návrhových vzorů se vyrábí procesy postavené na vztahových schématech mezi účastníky. Jsou-li tyto postupy favorizovány před skutečně odbornými postupy, nelze optimálního řešení nikdy dosáhnout.

11.1 Problematika antipatterns – jiný pohled na návrhové vzory

V rámci doporučení se nelze v této práci vyhnout tématu antipatterns – návrhové antivzory. Neslouží k tomu, jak by se dalo podle názvu snad také dovodit, aby popisovaly situace, kterým je třeba se vyhnout. Zachycují se ve svém katalogu, jako původně dobře zamýšlené vzory, které však v následné praxi neobstály, anebo se jejich původní dobré vlastnosti obrátily v neprospěch.

Ukázkovým případem souvisejícím s databázovou vrstvou jsou OR mapery. Právě datová vrstva v jejich koncepci se stala důvodem, proč se do takového katalogu zařadily. Laurie Voss (dlouholetý technický vedoucí Yahoo!Gadgets, v současnosti awe.sm) je autorem článku, ve kterém zařazení ORM mezi antivzory zdůvodňuje.

Následující text je tedy určen jako doporučení pro realizátory vlastních návrhových vzorů. Ukazuje, jakým způsobem se ORM podařilo projít cestu od původních dobrých záměrů ke kritickým výsledkům. Patrně neexistuje rozsáhlejší uplatnění této problematiky, na které by bylo možno zjevně poukázat na nebezpečí celé záležitosti.

- Původní záměry ORM:
    - Odstranit nutnost použití SQL (jinými slovy: nezatěžovat „geniálního“ vývojáře takovou podružností, jako je databázový server a sql, ale nechat mu prostor pro jeho skvělou tvorbu business vrstvy).
    - Generování kódu (jinými slovy – scaffolding; náš geniální vývojář se už nemusí zdržovat ani tvorbou formulářů pro správu dat).
    - Efektivita je „good enough“ (jinými slovy, všechny ORM upřednostňují jednoduchý kód před výkonem – problém je v tom, že pokud výkon nestačí, geniální ORM vývojář už ani nic lepšího do kódu vložit neumí).

- Základní problémy:
    - Abstrakce – cílem abstrakce je zjednodušit. Abstraktní vrstva nad SQL předpokládá, že vývojář zná SQL a v důsledku zdvojnásobuje potřeby znalostí; chápat, jak položit dotaz a umět ho v daném ORM podat. NHibernate používá pro tyto účely HQL, což je jakési SQL, do kterého se nakonec stejně HQL přeloží.
    - Osvobození od datové vrstvy – problém, zmiňovaný v této práci na několika místech. V důsledku je pak databázový server použit pouze jako úložiště, což v relačních modelech je výkonnostně nevhodný přístup. Zapomenout na optimalizaci datového serveru je v mnoha případech základní problém celkové výkonnosti výsledného IS.
    - Naprostá neefektivita dotazů – chce-li si vývojář vybrat objekt z databáze, nemá ve většině ORM možnost vybrat jeho vlastnosti na relačním protějšku. Pokud tedy vývoj směřuje cestou své jednoduchosti, enormně narůstá zatíženost výkonu.
- Řešení:
    - Pokud vývojář pracuje s objekty, je třeba se rozloučit s relační databází. V nabídce je spousta key-values úložišť a neexistuje pravidlo, které by nařizovalo automaticky použít relační databázi. Obzvláště v případě, kdy její schopnosti vývojář neumí nebo ani nechce využívat.

Z konkrétního rozboru jednoho z největších antivzorů vyplývá, že výsledkem nemusí být jeho nepoužívání, ale spíše jeho přesná aplikace, v tomto případě nepoužití relační databáze jako cíle.

V obecném pojetí existuje katalog antivzorů, který zahrnuje ukázkové problémy nejen z oblasti IT. Je zahrnut jako příloha této práce s použitím různých zdrojů.

11.2 Závěry

Přínos návrhových vzorů na úrovni databázového úložiště tak, jak vyplývá z práce, by se dal rozdělit do několika bodů:

- Ignorování síly databázových serverů je chyba. Spoléhání na výkonnost generických nástrojů typu ORM je zcestné a velmi často přináší více problémů než usnadnění.
- Striktní respektování objektově-relačního modelu není zdaleka vždy ideální. Pokud je třeba navázat objektovou aplikaci na data, existují lepší způsoby, a to buď ve formě cílových objektových úložišť, nebo jiných nerelačních architektur.

- Existují jasně definované procesy (např. ETL), které vyžadují maximální znalost databázového úložiště a s tím i související pochopení dobrých návrhových vzorů. Pokud takové operace realizuje osoba, uzavřená pouze v objektovém světě, nikdy nedosáhne v případě ETL efektivních výkonů.

- Zahrnout návrhové vzory SQL do analytické části vývoje aplikace znamená vyhnout se všem třem výše popsaným chybám. Čím více je analytick schopný korespondovat s datovou vrstvou, tím lepší jeho analýza bude. Nejde o znalost konkrétního SQL, jde o znalost možností datového úložiště – návrhové vzory v tomto případě dokáží analytickou část obohatit a vyřešit leckdy kolizní body celého návrhu, jež mají tendenci se projevit nikoliv při vývoji, ale bohužel až při praktickém provozu.

Práce se pokusila splnit požadované cíle – autor věnoval její realizaci nejen třicet let předcházející praxe, ale také rok usilovného studia dostupných materiálů, konzultací s experty nejen v České republice. Vychází z realizací informačních systémů včetně těch pro Univerzitu Hradec Králové a zároveň se snaží nahlédnout i do blízké budoucnosti. Je velmi pravděpodobné, že na tuto práci budou navazovat další články a práce s podobnou tématikou, obzvláště zajímavým prostředím se autorovi jeví kvantové databáze.

Stejně jako ve všech oborech, tak i v datových úložištích platí, že povrchní přístup se v důsledku nevyplácí. Jakékoliv dogma, ať již objektové nebo relační, omezuje a může způsobit v konkrétní realizaci více problémů než užitku. Znalosti, které jsou spíše prezentačního charakteru a nikoliv hlubšího a souvztažného, nejsou schopny vytvářet pevné a kvalitní výtvory.

Závěrem by autor chtěl poděkovat všem, kteří se desítky let podílejí na tvorbě datových úložišť a také vyjádřit naději, že stále budou existovat vývojáři, pro něž nebude výsledek ukryt pouze ve schématech a analýzách, ale také v reálných, použitelných výstupech.

12 Literatura a zdroje

Literatura:

[1] ARLOW Jim, Neustadt Ila. *UML 2 a unifikovaný vývoj aplikací*. 1. vyd. Computer Press, 568 s. 2012 [cit. 2013-10-1]

[2] BARBARA PERNICI, Costantino Thanos (eds. *Advanced information systems engineering 10th international conference, CAiSE '98, Pisa, Italy, June 8-12, 1998: proceedings*. Berlin [etc.]: Springer, 1998. ISBN 9783540694342.

[3] BLAHA, M., 2010: *Patterns of Data Modeling*, 1st ed., CRC Press

[4] BORKOVEC, Roman. *Effective use of the UML-Language in small companies*. 2012 [cit. 2014-01-05], ISSN: 16609336 ISBN: 978-303785751-9 Dostupné z: 10.4028/www.scientific.net/AMM.336-338.2111

[5] BORKOVEC, Roman. *Design of methodology for connecting Enterprise Architect with development solutions and necessary application Framework* [online]. [cit. 2015-09-30]. ISBN 978-1-61804-320-7

[6] FREEMAN, Eric, Elisabeth ROBSON, Kathy SIERRA a Bert BATES. *Head First design patterns*. Sebastopol, CA: O'Reilly, c2004, xxxvi, 638 p. ISBN 0596007124.

[7] GAMMA, Erich. *Design patterns: elements of reusable object-oriented software. Reading*, Mass.: Addison-Wesley, 1995, xv, 395 p. ISBN 0201633612-.

[8] HASLER, Tony. *Expert Oracle SQL: optimization, deployment, and statistics*. California: Apress, 2014, xxxi, 585 pages. ISBN 9781430259770.

[9] JABLONSKÝ, Josef. *Operační výzkum: kvantitativní modely pro ekonomické rozhodování*. 3. vyd. Praha: Professional Publishing, 2007, 323 s. ISBN 978-80-86946-44-3.

[10] KARWIN, Bill. *SQL antipatterns: avoiding the pitfalls of database programming. Raleigh*, N.C.: Pragmatic Bookshelf, c2010. Pragmatic programmers. ISBN 1934356557.

[11] RIORDAN, Rebecca M. *Vytváříme relační databázové aplikace*. Vyd. 1. Praha: Computer Press, 2000, xiv, 280 s. Databáze. ISBN 80-7226-360-9.

[12] ŘEPA, Václav. *Podnikové procesy: procesní řízení a modelování*. 1. vyd. Praha: Grada, 2006, 265 s. Management v informační společnosti. ISBN 80-247-1281-4.

[13] SANDHU, R., FERRAIOLO, D.F. and KUHN, D.R. (July 2000). *"The NIST Model for Role Based Access Control: Toward a Unified Standard*" (PDF). *5th ACM Workshop Role-Based Access Control*. pp. 47–63.

[14] SILVERSTON, L., 2001a: *The Data Model Resource Book: Volume 1: A Library of Universal Data Models for All Enterprises*, rev. ed., Wiley

[15] SILVESTRON, L., 2001b: *The Data Model Resource Book: Volume 2: A Library of Data Models for Specific Industries*, rev. ed., Wiley

[16] SILVESTRON, L. & AGNEW, P., 2009: *The Data Model Resource Book: Volume 3: Universal Patterns for Data Modeling*, rev. ed., Wiley

[17] ŠULA Jiří, CHONG Raul F., HAKES Ivan, AHUJA Rav, DVORSKÝ Bohuslav, *Začínáme s DB2 Express-C*, ISBN-978-80-86948-24-9

[18] TROPASHENKO Vadim, *SQL Design Patterns*, ISBN-13:978-0-97767 15-4-0

[19] VIEIRA, Robert. *Professional Microsoft SQL server 2008 programming*. Indianapolis, IN: Wiley Pub., 2009, xxxix, 893 p.

Ostatní zdroje:

[20] *.NET Technology Guide for Business Applications* [online]. [cit. 2016-01-04]. Dostupné z: http://blogs.msdn.com/b/microsoft_press/archive/2013/11/13/free-ebook-net-technology-guide-for-business-applications.aspx

[21] CARLSON Carl : *Design patterns as first-class connectors*. RIIT '13: Proceedings of the 2nd annual conference on Research in information technology [cit. 2016-01-04]

[22] COCKSHOTT Paul, *Quantum Relational Databases*, http://arxiv.org/abs/quant-ph/9712025 [cit. 2016-03-24]

[23] Články na téma Návrhové vzory v ekonomii, http://theurbantechnologist.com/2013/04/03/a-design-pattern-for-digital-urbanism-city-centre-enterprise-incubation/ [cit. 2015-11-04]

[24] FOWLER Martin, http://martinfowler.com/eaaCatalog/index.html [cit. 2015-10-12]

[25] Gabriel Bender, Lucja Kot, Johannes Gehrke. *Explainable Security for Relational Databases.* SIGMOD 2014. (electronic appendix) (source code) [cit. 2016-02-04]

[26] HASSO Sargon, ZHU Hong, BAYLEY Ian: *An algebra of design patterns*, 2013, Transactions on Software Engineering and Methodology (TOSEM) , Volume 22 Issue 3 [cit. 2016-01-05]

[27] HUDA Al-Shuaily, *SQL Patterns: A new approach in learning SQL*, University of Glasgow - University Avenue, Glasgow G12 8 8RZ, United Kingdom huda@dcs.gla.ac.uk - http://www.dcs.gla.ac.uk/~huda [cit. 2016-03-11]

[28] KOŠÁREK, Lukáš. Výkonnostní srovnání relačních databází [online]. Brno, 2011 [cit. 2015-12-29]. Bakalářská práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Vlastislav Dohnal Dostupné z: <http://is.muni.cz/th/256433/fi_b_b1/>.

[29] LONG, John. *Software Reuse Antipatterns*. ACM SIGSOFT Software Engineering Notes, 2001, http://dl.acm.org/citation.cfm?id=505492 [cit. 2016-01-04]

[30] Materiály k Enterprise Architect *http://edu.uhk.cz/~svoboka1/* [online]. 2001 [cit. 2013-05-10]. Dostupné z: http://edu.uhk.cz/~svoboka1/ [cit. 2015-01-18]

[31] ÖMER B., *Quantum Programming in QCL*, Master thesis (technical physics), Technical University of Vienna, 2001. [cit. 2016-02-05]

[32] PALOVSKÁ Helena, *K čemu jsou dobré databázové návrhové vzory?* JEL Classification: M10, C88 [cit. 2015-08-01]

[33] Server objektových technologií *www.objects.cz* [online]. 2001 [cit. 2013-05-10]. Dostupné z: http://www.objects.cz/ [cit. 2015-08-01]

[34] Stein, René. b*log.renestein.cz* [online] 2010 [cit. 2014-01-05] Dostupné z: http://blog.renestein.cz [cit. 2015-01-02]

[35] ŠTĚPÁN Petr: *Design pattern solutions as explicit entities in component-based software development*, In: WCOP '11: Proceedings of the 16th international workshop on Component-oriented programming [cit. 2016-02-02]

[36] Sudip Roy, Lucja Kot, Christoph Koch. *Quantum Databases*. CIDR 2013 [cit. 2015-11-02]

13 Seznam obrázků

14 Seznam tabulek

15 Přílohy

1 Projektová databáze

Projektová databáze byla realizována na databázovém serveru MSSQL 2014. Její skript včetně dat byl vytvořen v produktu EMS SQL Manager. Před jeho spuštěním je třeba mít vytvořenou databázi dp2016. Kompletní skript včetně dat je na přiloženém CD.

2 Katalog antivzorů

Přehledový katalog antivzorů je zčásti sestaven z práce Longa [29] a následně doplněn obecně zažitými pojmy z reálného světa. Jedná se pouze o informační přehled, striktně definovaný katalog neexistuje. Z oblasti ekonomie a managementu autor vycházel z článků [23], z oblasti SQL z [9].

2.1 Antivzory při vývoji software

- Velká koule bahna

Projekt, nebo aplikace, jsou tvořeny hlavním subjektem (třídou), na který se nabaluje vše ostatní pouze jako doplněk. Tato hlavní třída se někdy nazývá Božská třída – The God Class. Vzniká špatnou nebo vůbec chybějící architekturou a bývá často užitným vzorem samotvůrců, kteří nemají ve zvyku pracovat ve skupině, v týmu.

- Proud lávy

V projektu zůstávají staré části kódu, jejichž autorství nebo dokonce i funkce jsou nejasné. V podstatě tento antivzor kooperuje s tvrzením, co funguje, nechte fungovat. Leckdy dochází i k tvorbě duplicitních kódů, protože se v projektu ztratila nosná architektura. Velice často se k tomuto antivzoru uchylují vývojáři v situaci, kdy přebírají existující, zastaralý a nekvalitně zdokumentovaný projekt a jsou nuceni udržovat jeho běh a zároveň doplňovat další funkcionality. Bývá důsledkem i tlaku zadavatele, který nechce obětovat prostředky na kompletní přestavbu projektu.

- Kotva

Nevyužitá část projektu, která původně měla být tou podstatnou částí. Příkladem z praxe je např. realizace připojení na RFID čtečky, které následně nebyly nikdy zadavatelem pořízeny. Taková situace vzniká, pokud zadavatel nekomunikuje dostatečně s řešitelem, resp. ignoruje jeho technické a provozní připomínky.

- Zlaté kladivo

Velmi častý antivzor, pokud vývojář není schopen poskytnout nic kromě toho, co sám zvládá. V principu jde o prosazování postupů, metod, frameworků, které v konkrétní situaci nepřinášejí výhody, existuje lepší alternativní řešení. Příkladem mohou být třívrstvé aplikace i tam, kde je třeba se promptně zaměřit na výkon a celou architekturu postavit na zcela jiných základech.

2.2 Antivzory v softwareové architektuře

- Švýcarský nožík (Swiss Army knife)

Antivzor, který se vyskytuje tam, kde teoretické znalosti předstihují praktické požadavky. Vývojář přemýšlí a následně realizuje projektové třídy příliš obecně, zeširoka. V důsledku je většina funkcionalit nepoužitá a leckdy ani běžný vývojář není schopen pochopit, co mu vlastně všechno tyto třídy nabízejí. Vzniká problém nejen s využitím, ale i s udržováním, dokumentací a nutnými opravami.

- Závislost na dodavateli

Standardní, ale někdy bohužel nutné vyvolaná chyba při vývoji. Pokud existuje jediný dodavatel dané technologie (software/hardware) vzniká závislost, která může být závažím pro další rozvoj projektu. Je nutná údržba, zajištění kompatibility s prostředím, což v důsledku může prodlužovat celkový vývoj a zvyšovat jeho náklady.

- Vynalézání kola

Tento antivzor nerespektuje odborný a znalostní nadhled architekta. Nedochází k využití existujících možností, vše se tvoří na zelené louce. Ukázkovým příkladem je samotný obor webových aplikací, který dodává myšlenky, jejichž realizace byla řešena již v době mainframe systémů. Takové systémy jsou od počátku odsouzeny k nekvalitní architektuře, protože tento přístup sám o sobě naznačuje neochotu získat profesní nadhled.

2.3 Antivzory v projektovém managementu

- Paralýza analýzou (Analysis paralysis)

Analýza před realizací je nezbytnou součástí vývoje projektu. Je nutné si uvědomit, že nikdy nelze analýzu provést dokonale, protože v reálném čase se mění vše, co se měnit může. Pokud analytik směřuje k tomu, aby jeho analýza byla naprosto dokonalá a všepostihující, realizuje právě tento návrhový antivzor – někdy je k tomu dotlačen zadavatelem, který si v této fázi přijde jistější, než při samotné realizaci projektu.

- Inženýrství grafů (Viewgraph engineering)

Přehnané modelování souvisí i s předchozím antivzorem. Výsledkem projektového managementu má být realizovaný projekt, nikoliv nadměrné množství grafů, schémat a prezentací.

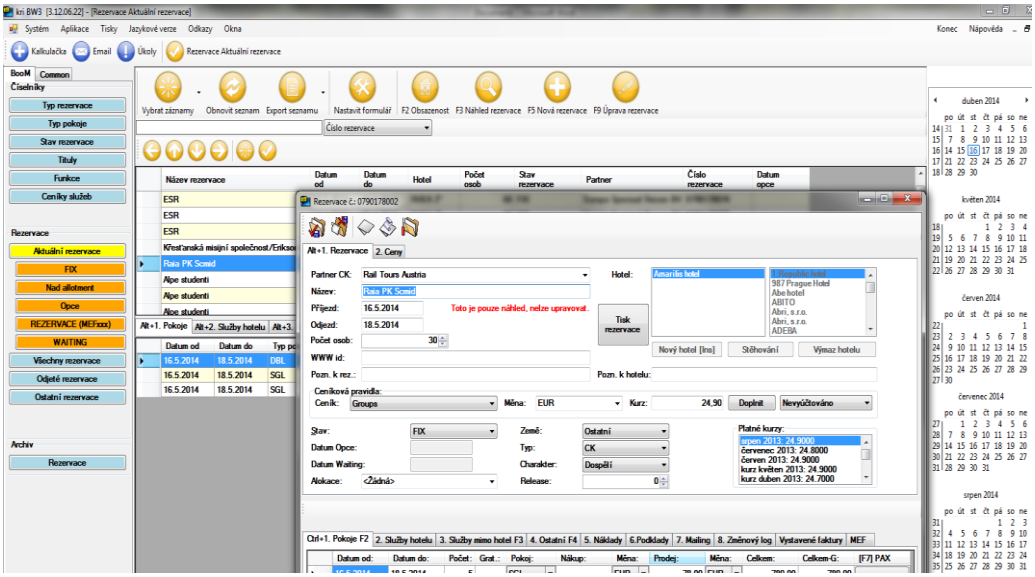
- Strach z úspěchu (Fear of success)

V posledních etapách realizace se může vyskytnout antivzor Strach z úspěchu. Přináší ho nejistota z možných chyb v samotném provozu, a vznikají oddalovací termíny pro spuštění, případně i další iracionální rozhodnutí. V celkovém výsledku tento přístup může zničit i kvalitně vedený a realizovaný projekt.

3 Stručný popis aplikace Jerome Hotels

Aplikace byla realizována a udržována autorem v letech 2006 – 2014 pro pražskou firmu Jerome Hotels s.r.o. Z pohledu této práce je na ní zajímavý dynamický model membershipu, který musel postupně zajistit následující požadavky:

- standardní lineární model Uživatel – Role,
- časové přiřazení k právům a rolím,
- napojení na vnitřní podnikový systém LDAP,
- realizace kontroly webového přístupu,
- kontrolované připojení z cizích systémů jiných poskytovatelů (Rusko, Evropa, USA).



Obrázek 5 Obrazovka aplikace Jerome (zdroj vlastní)

4 Stručný popis aplikace Dyndal – ambulantní systém

Aplikace byla autorem realizována a provozována v letech 1994 – 2016 na různých platformách tak, jak bylo zapotřebí reagovat na požadavky cílových klientů. Vzhledem k jejímu modulovému nasazení byla používána v různých prostředích. Od lokální stanice až po propojené sítě (např. pracoviště polikliniky Galenia Cheb, které propojovalo různá pracoviště v republice a zájmově různé klienty – od standardních sesterských a lékařských stanic po vědecké, rešeršní a monitorovací stanice).

5 Aplikace pro UHK

Autor spolupracoval na vývoji *Přijímacích řízení* v letech 2010 – 2013 a *Praxe* v letech 2013 až do současnosti. V obou případech bylo nutno řešit připojování uživatelů s přihlédnutím k několika variant při současném zajištění bezpečného provozu. Uživatelé mohli využít:

- automatického připojení v doméně,
- zástupného připojení,
- připojení mimo doménu,
- připojení a kontrola externích uživatelů mimo LDAP UHK.

6 Tvorba add-inu v prostředí Enteprise Architect

V okamžiku spuštění si EA načte informaci z registru [HKEY_CURRENT_USER\Software\SparxSystems\EAAddins]. Každý klíč nalezený v tomto registru reprezentuje jeden add-in. Hodnota klíče je defaultně rovna názvu assembly a názvu add-inu, oddělenými tečkou.

Tím EA získá od Windows informaci, kde je odpovídající assembly uložená a po jejím spuštění jsou k dispozici všechny její public metody.

K samotnému vytvořeni add-inu je tedy třeba:

- Vytvořit dynamickou knihovnu (dll) add-inu, obsahující volanou třídu.
- Přidat do registru klíč s hodnotou assembly a add-inu.
- Zaregistrovat standardním způsobem knihovnu do COM codebase.

Další postup je uveden na přiloženém DVD.

7 Realizace databázového providera

Jedná se o praktickou ukázku využití abstraktního vzoru FACTORY a SINGLETON. Při využití tohoto providera přestává být vnitřní kód aplikace závislý na konkrétním datovém úložišti s tím, že nijak neomezuje využití specifických vlastností tohoto úložiště. Všechny operace respektují konkrétního providera, který je volně modifikovatelný a není třeba na něj v samotném kódu aplikace brát ohled.

Kód je uveden na DVD, soubor provider.sql.

8 Vytvoření schématu cizích klíčů (použití vzoru HISTORY)

Využití grafických možností návrhového vzoru HISTOGRAM při nutnosti dohladat závislosti dle cizích klíčů dané tabulky ukazuje následující procedura *getFK(tableName)*:

```
create proc dbo.getFK (

   @table varchar(256),

   @lvl int=0,

   @ParentTable varchar(256)='',

   @debug bit = 1)

AS begin
```

```
	set noCOUNT on;

	declare @dbg bit;

	set @dbg=@debug;

	if object_id('tempdb..#tbl', 'U') is null

		create table  #tbl  (id int identity, tablename varchar(256), lvl int,
ParentTable varchar(256));

	declare @curS cursor;

	if @lvl = 0

		insert into #tbl (tablename, lvl, ParentTable)

		SELECT @table, @lvl, Null;

	else

		insert into #tbl (tablename, lvl, ParentTable)

		SELECT @table, @lvl,@ParentTable;

	if @dbg=1

		print replicate('----', @lvl) + 'lvl ' + cast(@lvl AS varchar(10)) + '

= ' + @table;



	if not exists (SELECT * FROM sys.foreign_keys WHERE referenced_object_id =
object_id(@table))

		return;

	else

	begin -- else

		set @ParentTable = @table;

		set @curS = cursor for

		SELECT

tablename=object_schema_name(parent_object_id)+'.'+object_name(parent_object_id)

		FROM sys.foreign_keys

		WHERE referenced_object_id = object_id(@table)

		and parent_object_id <> referenced_object_id;



		open @curS;

		fetch next FROM @curS into @table;
```

```
                while @@fetch_status = 0
                begin
                        set @lvl = @lvl+1;
                        -- recursive call
                        exec dbo.getFK @table, @lvl, @ParentTable, @dbg;
                        set @lvl = @lvl-1;
                        fetch next FROM @curS into @table;
                END
                close @curS;
                deallocate @curS;
        END
        if @lvl = 0
                SELECT * FROM #tbl;
        return;
END
```

Ve výsledku vrací tato procedura kromě datového setu i tento grafický záznam:

```
3 row(s) returned
lvl 0 = Zamestnanec
----lvl 1 = dbo.ZamestnanecTym
--------lvl 2 = dbo.ZamestnanecTymPrace
(execution time: 344 ms; total time: 546 ms)
Return Code: 0
```

Při odzkoušení této procedury na databázi KenticoCMS (redakční systém UHK) autor obdržel 119 záznamů, které popisovaly devět úrovní.

16 Zadání závěrečné práce

Univerzita Hradec Králové
Faculty of Informatics and Management
Akademický rok: 2014/2015

Studijní program: Systems Engineering and Informatic
Forma: Combined
Obor/komb.: Informační management (im2-k)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Borkovec Roman	Břetislavova 1219, Hradec Králové - Pražské Předměstí	I14253

TÉMA ČESKY:

Návrhové vzory SQL

TÉMA ANGLICKY:

SQL Design Patterns

VEDOUCÍ PRÁCE:

doc. RNDr. Jaroslava Mikulecká, CSc. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Historie a vývoj datového modelování
Abstraktní návrhové vzory v datové vrstvě
Aplikační návrhové vzory
Nativní návrhové vzory v DL
Použití návrhových vzorů při výuce
Testování návrhových vzorů
Zhodnocení použití návrhových vzorů
Antipatterns - jiný pohled na návrhové vzory

SEZNAM DOPORUČENÉ LITERATURY:

Vadim Tropashko, Donald Burleson : SQL Design Patterns: Expert Guide to SQL Programming, 2007, ISBN:0977671542 9780977671540
Petr Štěpán: Design pattern solutions as explicit entities in component-based software development, In: WCOP '11: Proceedings of the 16th international workshop on Component-oriented programming
Sargon Hasso, Carl Carlson : Design patterns as first-class connectors. RIIT '13: Proceedings of the 2nd annual conference on Research in information technology
Hong Zhu, Ian Bayley: An algebra of design patterns, 2013, Transactions on Software Engineering and Methodology (TOSEM) , Volume 22 Issue 3

Podpis studenta: ..... **Datum:** .....

Podpis vedoucího práce: ..... **Datum:** .....