VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNCHRONIZACE DATABÁZÍ MYSQL A MS SQL

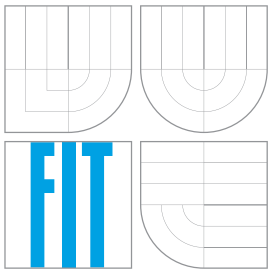
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

FILIP ŠIMANOVSKÝ

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNCHRONIZACE DATABÁZÍ MYSQL A MS SQL

MYSQL AND MS SQL DATABASE SYNCHRONIZATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

FILIP ŠIMANOVSKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Mgr. ROMAN TRCHALÍK, Ph.D.

BRNO 2014

Abstrakt

Tato bakalářská práce se zabývá synchronizací databází MySQL a MS SQL. Zkoumá a popisuje rozdíly mezi těmito dvěma databázovými servery a dále navrhuje a implementuje algoritmy pro jejich bezpečnou a ekonomicky bezztrátovou synchronizaci. Navržené postupy jsou následně implementovány do konzolové aplikace, jejíž práce je demonstrována na dvou případových studií. Výsledkem bakalářské práce je tedy návod k úspěšnému převodu databáze MySQL na MS SQL (a zpět) a samotný program, který tento převod realizuje.

Abstract

This thesis describes the synchronization of MySQL and MS SQL databases. It explores and describes differences between these two database servers and further designs and implements alghorithms for their safety and economicaly favourable synchronization. Designed methods are implemented in a console application afterwards. Two case studies are used as an illustrative of its work. The result of this thesis is a manual on how to successfully convert MySQL database to MS SQL database (and vice versa) and the convert application itself.

Klíčová slova

Databáze, MySQL, MS SQL, SQL Server, synchronizace, emulace, převod, SQL, SQL-92, T-SQL, spouštěč, pohled, procedura, index, klíč, primární klíč

Keywords

Databases, MySQL, MS SQL, SQL Server, synchronization, emulation, conversion, SQL, SQL-92, T-SQL, trigger, view, procedure, index, key, primary key

Citace

Filip Šimanovský: Synchronizace databází MySQL a MS SQL, bakalářská práce, Brno, FIT VUT v Brně, 2014

Synchronizace databází MySQL a MS SQL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Mgr. Romana Trchalíka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Filip Šimanovský
20. května 2014

Poděkování

Děkuji panu R. Trchalíkovi za trpělivost, odborné rady a čas, který mi poskytl při práci na této bakalářské práci.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Tato práce je zaměřena na oblast praktického využití databází. Téma dává tušit, že se zde budu zabývat problémem konverze jednoho typu databáze na jiný.

Bez databází si nelze v dnešní době život představit. Neexistuje snad jediný obor, jediná instituce či jediný člověk, který by třeba jen nepřímo nikdy žádnou databázi nevyužil. Umožňují nám třídit, porovnávat a skladovat informace na takové úrovni, jako žádný jiný prostředek na světě. Jako takovým je potřeba databázím věnovat značné úsilí nejen při jejich používání, ale především při jejich vývoji a správě.

Bylo vyvinuto nepřeberné množství typů databází, které se liší mnoha parametry a tak se někdy dostaneme do situace, kdy zjistíme, že nám zvolený typ databáze nevyhovuje a potřeboval bychom jej změnit. V tom však narazíme na problém, jak současná data bezpečně, rychle a také ekonomicky přesunout do nového úložiště.

A právě tento problém se pokusí práce, kterou držíte v ruce, vyřešit. Ačkoliv byly vyčleněny konkrétní dva typy databází, mezi kterými bude synchronizace probíhat, dají se zde nalézt body aplikovatelné na synchronizaci databází libovolných.

Toto se týká především první poloviny úvodní kapitoly, která pojednává o synchronizacích v širším spektru souvislostí. Pokusím se zde odpovědět na otázky jako například: Co je to vlastně synchronizace? Proč je potřeba? Jaké typy synchronizací lze provádět? Druhá polovina úvodní kapitoly pak již pojednává konkrétně o dvou vybraných databázích, a to MySQL a MS SQL. MySQL je v současnosti druhá nejčastěji nasazovaná databáze na světě[1], následována na třetím místě databází MS SQL[1], často také zvanou SQL Server či Microsoft SQL Server. Podíváme se tedy, čím se od sebe odlišují a co bude nutné při jejich synchronizaci vyřešit.

Následně se pokusím na základě zjištěných informací navrhnout obecný algoritmus pro obousměrnou synchronizaci libovolných dvou tabulek. Algoritmus pak rozšířím tak, aby jej šlo aplikovat na synchronizaci celých databází.

Dalším cílem této práce je navržený algoritmus implementovat. Výslednou aplikaci lze shlédnout na přiloženém CD a její práce je pak demonstrována na dvou vybraných případových studiích.

V samotném závěru pak budou zmíněny možnosti, jak tuto práci dále rozvíjet.

Kapitola 2

Seznámení

2.1 Synchronizace databází

V této části práce se čtenář dozví odpovědi na otázky: Co je to synchronizace databází? Kdy je potřeba? Jaké typy synchronizací známe a kdy zvolit který? Jak probíhá samotná synchronizace?

Pro zjednodušení a zkrácení zápisu se v dalším textu předpokládá následující značení:

- Písmeno S označuje databázi, která obsahuje nejaktuálnější data (S podle anglického „source“ — zdroj).
- Písmeno T označuje databázi, ve které je potřeba data upravit tak, aby odpovídala datům v S (T podle anglického „target“ — cíl).

2.1.1 Definice

Synchronizace je dle [3] uvedení nějaké činnosti v časovou jednotu, v časový soulad s jinou činností. V této práci je však potřeba definovat spíše pojem „synchronizace dat“. Avšak tento termín — zřejmě díky jeho relativní modernosti — nelze nalézt v žádném českém renomovaném slovníku cizích slov, a proto jsme nuceni zde použít definici dle [2]:

„Synchronizace dat má za úkol neustále udržovat přesné kopie zdrojových dat ve všech synchronizovaných úložištích.“

Za synchronizovaná úložiště jsou v této práci považovány databáze uložené na MS SQL databázovém serveru a na MySQL databázovém serveru. Cílem výsledné aplikace je tedy vytvoření prostředku, který zajistí, že každá změna v databázi S se adekvátně promítne do databáze T.

V dalším textu budeme pod pojmem „synchronizace“ vždy rozumět význam termínu „synchronizace dat“.

2.1.2 Proč je synchronizace databází třeba

Synchronizovat databáze je obecně potřeba ve třech případech:

1. Administrátor či programátor se z určitého důvodu rozhodne změnit databázovou vrstvu aplikace a začne využívat jiný typ databáze.

2. S databází potřebuje pracovat více aplikací, z nichž některá(é) neumí nebo nemůže obsloužit daný typ databáze.

3. Vedení záložní databáze založené na jiném typu serveru.

Tyto situace si následně podrobněji rozebereme.

Případ 1: Změna databáze, se kterou aplikace pracuje

V tomto prvním případě se jedná o **synchronizaci jednorázovou** — je potřeba v jednom momentě přenést celou databázi S do databáze T. Jedná se rovněž o synchronizaci jednosměrnou. Zpravidla je po dobu této synchronizace aplikace odstavena či je zakázáno mazání, aktualizace a vkládání nových záznamů do databáze. K rozhodnutí změnit typ databáze vede administrátory většinou některý z následujících důvodů:

- Přechod na novější verzi softwaru třetí strany, který pracuje s jiným typem databáze.
- Současná databáze nedostačuje požadavkům aplikace. Tyto požadavky mohou být výkonnostního, bezpečnostního či programového charakteru. Programovým charakterem je v tomto případě myšlen souhrn přístupového rozhraní (konzole, skriptovací jazyk) a funkcí databáze (například neexistující číselný datový typ s dostatečnou přesností u bankovní aplikace).
- Ukončení podpory stávající databáze ze strany vývojářů a s tím spojené bezpečnostní hrozby.
- Potřeba snížení nákladů na provoz databáze (u komerčních databázových serverů).
- Viz druhý bod z úvodního výčtu. Změna užívaného typu databáze je jedno ze dvou možných řešení.

Případ 2: Potřeba přístupu k databázi od více aplikací

Ve druhém případě — tedy tehdy, kdy k databází přistupuje více aplikací, ale každá by chtěla pracovat s jiným typem databáze — je potřeba provádět synchronizaci pravidelně, nikoliv jednorázově. Rovněž je potřeba provádět synchronizaci obousměrně. K této situaci dochází tehdy, kdy do již zaběhlého systému přidáváme nové aplikace (například když s Registrem osob, jenž spravuje Ministerstvo vnitra, potřebuje najednou — díky novému zákonu — pracovat i Ministerstvo obrany, které ale běžně pro své účely využívá jiný typ databáze). Je zde potřeba zajistit dostatečnou aktuálnost záznamů a co nejpřesnější replikaci dat.

Případ 3: Vedení záložní databáze na serveru jiného typu

Poslední případ — vedení záložní databáze na jiném typu databázového serveru — je spíše hypotetický příklad. Proto si jen řekněme, že je zde potřeba volit takové typy databází, které jsou tam i zpět téměř 100% kompatibilní, aby v případě havárie nedošlo ke ztrátě dat vinou zkreslení informací při samotné synchronizaci.

Důsledky automatické synchronizace

Pokud by synchronizace neprobíhala automaticky nebo alespoň poloautomaticky za pomoci synchronizačních programů, bylo by třeba všechny databáze převádět ručně. To zahrnuje opětovné vytvoření tabulek, převedení všech řádků, přepsání všech pohledů, spouštěčů, procedur a podobně. A to vše s ohledem na jiný typ databáze.

Pokud navíc přihlédneme k tomu, že mnohdy se databáze skládají ze stovek či tisíců tabulek, které obsahují milióny řádků, zjistíme, že by ruční převod mezi databázemi zaměstnal velké množství lidí, kteří by nad tímto úkolem strávili mnoho času. Všichni tito lidé by navíc museli mít nejvyšší administrátorská oprávnění a také oprávnění zobrazovat veškerá — tedy i osobní — data v databázi uložená.

Pokud by navíc bylo potřeba provádět synchronizaci v reálném čase, tedy tak, jak se záznamy v databázích mění, shledáme, že je manuální synchronizace zcela vyloučena.

Užitím k tomu určeného programu tak vede k výrazným ekonomickým důsledkům, kdy jsou ušetřeny nejen náklady na lidské a finanční zdroje, ale rovněž čas. Eliminuje se také možnost zanesení chyby vinou lidského faktoru. Naopak se nevylučuje individuální úprava výsledné databáze po skončení synchronizace.

2.1.3 Typy synchronizace databází

Abychom mohli o synchronizacích mluvit dále, musíme si určit jejich vlastnosti a jejich rozdělení.

Klasifikace podle hloubky synchronizace

Hloubkou se v tomto případě myslí, na jaké úrovni v hierarchii databáze synchronizace probíhá. Standardně tedy synchronizace může probíhat na těchto úrovních (seřazeno dle obecnosti vzestupně):

1. Sloupec
2. Tabulka
3. Schéma
4. Databáze
5. Celý databázový server

Dle typu databáze se někdy vynechává úroveň schéma (viz kapitola 2.2.6). Zároveň platí, že při synchronizaci na obecnější úrovni musíme vždy provést synchronizaci specifičtějších úrovní. Tzn. pokud synchronizujeme tabulku, musíme sesynchronizovat i její sloupce. Synchronizace celých databázových serverů probíhá zřídka a přináší především problém synchronizace tabulek systémových, které jsou zpravidla velmi odlišné, a je tak velmi obtížné je věrohodně emulovat. Pro příklad si uveďme, že MySQL 5 pracuje s 28 systémovými tabulkami, zatímco SQL Server 2012 s 281[1][4],[7]!

Specifikací úrovně synchronizace pak lze ušetřit strojový čas, který by byl spotřebován na synchronizaci nepotřebných částí databáze. Rovněž je výhodná v případě, že v T se k

[1]Přesnější by u obou zmíněných databází bylo používání termínu systémové pohledy místo systémové tabulky. Mezi vývojáři se však ustálilo používat pojem systémové tabulky a dokonce i dokumentace k oběma databázím hovoří o systémových tabulkách namísto systémových pohledů.

databázi S standardně přičleňují ještě nějaké tabulky navíc, které by jinak byly smazány. V dalším textu se pak — pokud není uvedeno jinak — předpokládá synchronizace na úrovni databáze.

Klasifikace podle přístupu k datům

Samotná synchronizace pak může aktualitu údajů zajišťovat dvěma způsoby:

1. Aktualizací existujících záznamů.
2. Odstraněním všech existujících záznamů a opětovným vložení všech dat (tentokráte již aktuálních) do databáze.

Ačkoliv se druhý způsob může na první pohled jevit jako zbytečně náročný či dokonce nesmyslný, jsou situace, kdy je nutné jej použít. První způsob je totiž použitelný pouze pro tabulky s existujícím primárním klíčem nebo s definovaným unikátním indexem. Pokud ani jedno z těchto dvou omezení není implementováno, nelze určit zda je řádek aktualizován, či se jedná o zcela nový záznam. Vysvětleno na příkladu:

Mějme tabulku v databázi T s následující strukturou a uveďme jeden řádek dat:

Jmeno	**Prijmeni**	**Bydliste**
Martin	Máčel	Praha

Tabulka 2.1: Ukázka struktury tabulky, u níž nelze s jistotou rozlišit řádky.

V databázi S však nalezneme tento řádek:

Jmeno	**Prijmeni**	**Bydliste**
Martin	Máčel	Kolín

Tabulka 2.2: Ukázka řádku tabulky, který působí potíže při určení zda jde o nový záznam či aktualizaci existujícího.

Otázka, před kterou nyní stojíme zní: Přibyl do databáze nový uživatel s náhodou shodným jménem s některým z již existujících uživatelů, nebo se existující uživatel Martin Máčel přestěhoval? Na tuto otázku nelze bez dalších informací odpovědět. Dokud není v tabulce definován žádný sloupec, dle kterého by šlo identitu řádků určit. Při zavedení standardního ID sloupce, je již odpověď na otázku jasná:

ID	**Jmeno**	**Prijmeni**	**Bydliste**
1	Martin	Máčel	Praha
2	Martin	Máčel	Kolín

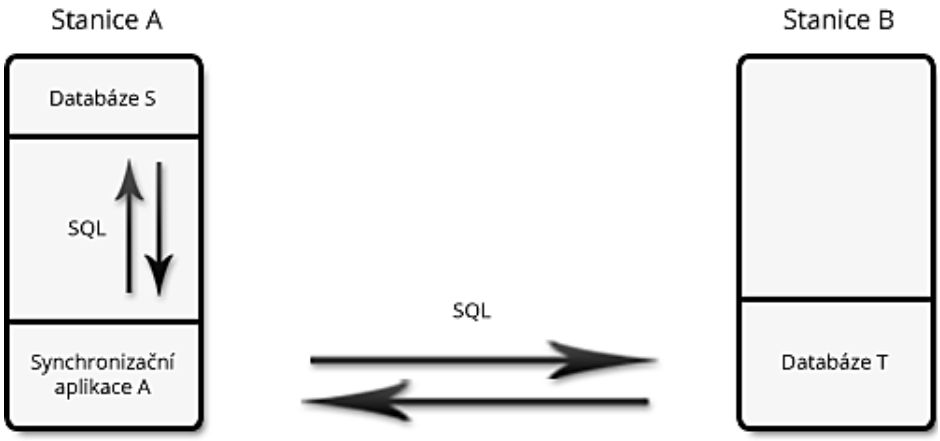
Tabulka 2.3: Ukázka struktury tabulky, u které lze jednoznačně rozlišit každý řádek.

Pokud tedy není v tabulce definován primární klíč či unikátní index, je nutno celou tabulku vyprázdnit a všechna data opětovně vložit. Jedině tak lze zajistit správné vyřešení problému nastíněného výše.

Klasifikace podle architektury synchronizace

Podle toho, jaké aplikace a servery do synchronizace vstupují, můžeme rozlišit dvě základní architektury synchronizačního procesu.

Na obrázku 2.1 je zobrazeno schéma první architektury. Vystupují zde pouze dvě stanice: Stanice A se zdrojovou databází S a aplikací, která celou synchronizaci řídí, dále pak stanice B, na které se nachází cílová databáze T. Princip synchronizace je poté následující: Někdo, patrně administrátor, iniciuje synchronizaci spuštěním synchronizační aplikace s patřičnými parametry. Aplikace poté pokládá téměř výhradně výběrové dotazy na databázi S. Tím zjistí její strukturu a obsah. Dotazy na databázi T poté aplikace přímo upravuje její strukturu dle informací získaných z S.

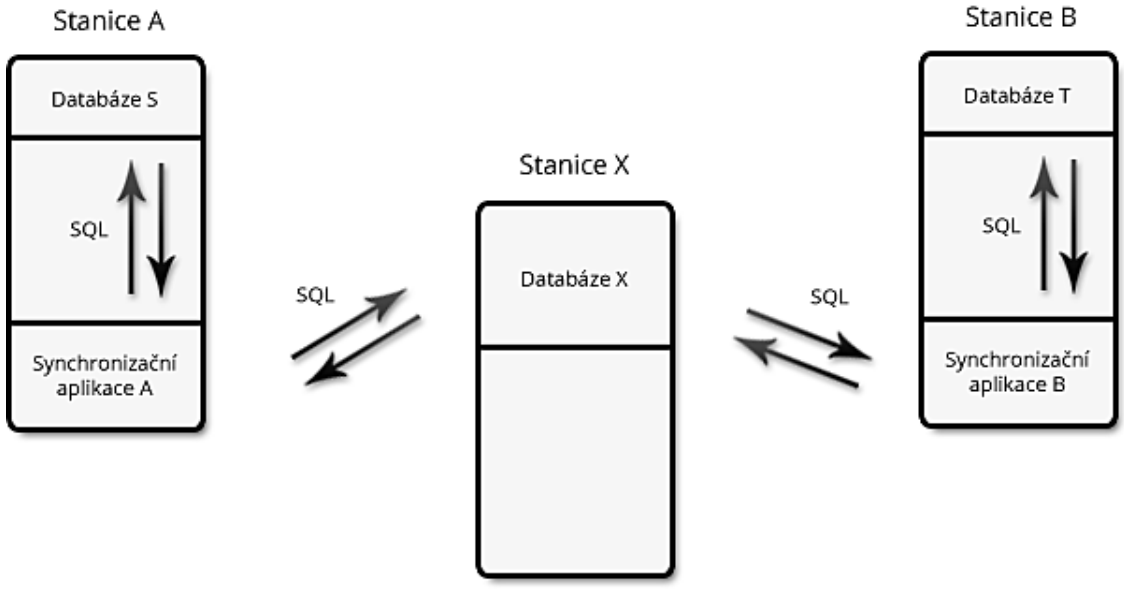


Obrázek 2.1: Architektura synchronizace bez využití třetí databáze.

V praxi se mohou objevit situace, kdy je synchronizační aplikace spuštěna na stanici B, či dokonce na zcela jiném stroji. Principiálně však jde stále o tutéž architekturu.

Výhodami této architektury je její relativní jednoduchost a přímočarost. Není potřeba žádný další server ani aplikace. Z toho plyne snadná údržba a menší riziko poruchovosti. Avšak z potřeby současné komunikace s oběma databázemi vyplývá, že je nemožné provádět synchronizaci, pokud je jedna z databází nedostupná.

Druhá obvyklá architektura je zobrazena na obrázku 2.2. Jak je vidět, zde do hry vstupuje třetí databáze X na stanici X a každá z databází S a T má svou vlastní synchronizační aplikaci. Princip je poté následující: Synchronizační aplikace A zkopíruje strukturu a data databáze S do databáze X. Synchronizační aplikace B pak zcela nezávisle na činnosti stanice A provede synchronizaci databáze T podle databáze X.



Obrázek 2.2: Architektura synchronizace s použitím třetí databáze jako prostředníka pro uchování dat.

Tento přístup má několik výhod. Především je odlehčena komunikace s databází S, což pro velmi vytížené databázové servery může být rozhodující faktor. Synchronizace je také možná bez ohledu na dostupnost stanic A a B. Více prvků umožňuje také lepší škálovatelnost celé architektury.

Tyto výhody jdou ovšem ruku v ruce s celou řadou nevýhod. Mimo potřeby dalšího databázového serveru je zde také větší náročnost na údržbu (spravujeme více aplikací a více strojů). S rostoucím počtem prvků v síti roste také pravděpodobnost, že komunikace na některém bodě selže. Může se také prodlužovat celková doba mezi kompletní synchronizací databází S a T, neboť nikde není garantováno, že synchronizační aplikace B bude spuštěna ihned po ukončení běhu aplikace A.

I u této architektury mohou být obě synchronizační aplikace umístěny v síti na libovolném místě, nikoliv nutně na stanicích A a B. Ve velmi speciálním případě se může jednat dokonce o jednu a tutéž aplikaci. Principiálně však jde stále o tutéž architekturu.

Klasifikace podle frekvence synchronizace

Klasifikace synchronizací podle frekvence — tedy podle toho, jak často se databáze synchronizují — je jedním z nejdůležitějších rozdělení.

1. Na vyžádání.
2. Pravidelná synchronizace.
3. Real-time synchronizace, neboli česky synchronizace v reálném čase.

Synchronizace na vyžádání proběhne pouze jednou. Tzn. pouze v jednom časovém okamžiku (ihned po skončení synchronizace) lze zaručit, že jsou obě databáze shodné, dále již každá z nich „žije svým vlastním životem“. Tento typ se využívá většinou v případech jako je například přesun celé databáze.

Pravidelná synchronizace je taková, která se opakuje v pravidelných časových intervalech. Je to pravděpodobně nejčastěji užívaným typem. A to díky možnosti nastavit, jak procesorově náročná má synchronizace být zkracováním či prodlužováním délky časových rozestupů mezi jednotlivými synchronizacemi. Zároveň zajišťuje dostatečnou aktuálnost všech údajů. Pro větší jistotu konzistence dat je ji většinou potřeba provádět jako synchronizaci obousměrnou (viz kapitolu 2.1.4). V praxi je tento typ obvykle realizován iniciováním synchronizace pomocí pravidelně spouštěného skriptu — například `cronem`.

Synchronizace v reálném čase — jak její název napovídá — je synchronizace taková, kdy se **každá** změna v databázi S okamžitě objeví i v databázi T. Takový přístup je pochopitelně velmi náročný na komunikaci mezi jednotlivými účastníky a tím pádem i na čas, objem přenesených dat, ale i náchylnost k chybám. Tuto variantu je možno v praxi realizovat třemi způsoby:

1. Pravidelným dotazování na databázi S v tak krátkých intervalech, že je možno je považovat za jeden neustálý dotaz. Jedná se vlastně o variantu pravidelné synchronizace tak, jak je popsána výše. Snaží se ovšem maskovat za synchronizaci v reálném čase.

2. Zabudováním systému řízeného událostmi přímo do jádra databáze. Událostmi v tomto případě jsou veškeré dotazy typu `UPDATE`, `DELETE`, `INSERT` a `ALTER`. Při výskytu události databáze sama odešle změny na nějaký synchronizační server. Jedná se o princip vzdáleně připomínající šíření změn v systému DNS (Domain name system - Systém doménových jmen).

3. Zabudováním odesílání aktualizací přímo od aplikace, která s databází pracuje. V případě, že aplikace provádí změnu v jedné databázi, provede ji synchronně i v databázi druhé.

Z logiky věci vyplývá, že nejvýhodnější a díky tomu také nejrozšířenější variantou je první uvedená.

2.1.4 Metriky a vlastnosti synchronizace

Mimo jiné z výše uvedeného tedy vyplývá, že u synchronizací obecně lze sledovat následující parametry:

Rychlost

Rychlostí je myšlena celková doba, kterou trvá uvedení databáze T ze svého původního stavu do stavu ekvivalentního databázi S. Tato doba je vyžadována bez ohledu na situaci vždy co možná nejkratší. Ovlivnit ji lze pokládáním vhodných a optimalizovaných dotazů na zúčastněné databáze či použitím rychlejší přenosové linky mezi servery.

Při příliš pomalé rychlosti hrozí inkonzistence výsledných dat v důsledku aktualizace databáze S či T v průběhu synchronizace.

Frekvence

Frekvencí synchronizace je myšlen počet opakování synchronizace za jednotku času. Řečeno více lidsky: Jak často se synchronizace provádí. Blíže bylo o tomto tématu pojednáno v sekci 2.1.3, proto zde již nebudeme opakovat jednou řečené.

Přesnost

Přesnost udává poměr, jak věrohodně se podařilo zobrazit databázi S v databázi T. Ideální poměr je 1:1, neboli 100 %. Tohoto poměru však lze dosáhnout pouze při převodu mezi stejnými typy databází. V takovém případě je však zpravidla lepší využít zabudované nástroje pro import a export. Teoreticky by 100% přesnosti šlo dosáhnout ještě při převodu mezi takovými typy databází, které se liší pouze v parametrech, jež na samotné uložení dat a práci s nimi nemají vliv. Takovým parametrem by mohla být například platforma, na níž je databázový server spustitelný. Já osobně jsem však žádné takové dvě databáze neobjevil. Nejblíže by však byly zřejmě MySQL a MariaDb.

Příliš malá přesnost může způsobit problémy při dalším užívání synchronizované databáze. Příkladem by mohla být aplikace shromažďující vědecké články, které se ukládají do řetězcového datového typu o velikosti 65 000 B. Jelikož však v databázi T takový datový typ není k dispozici, rozhodne se logika synchronizace, že využije jiný datový typ, který má velikost jen 60 000 B. Zdůvodní to tím, že články jsou výrazně kratší a rezerva je i tak dostačující. V případě, že se při dalším používání objeví článek o délce např. 63 000 znaků (za předpokladu, že 1 znak rovná se 1 B), bude v jedné databázi ořezán, kdežto v druhé nikoliv.

Přesnost jako taková je však obtížně měřitelná a tak je lepší ji vyjádřit spíše doprovodným komentářem.

Směrovost synchronizace

Dalším parametrem, kterým můžeme synchronizaci popsat, je její směrovost. Ta může být jednosměrná nebo obousměrná. Ve specifických případech (kdy synchronizujeme tři a více databází) by se dalo hovořit o vícesměrovosti, ve skutečnosti se však takovéto situace dají rozložit na několik jednosměrných synchronizací.

Tento parametr vyjadřuje, zda se pouze aktualizuje databáze T dle dat z S, nebo zda si databáze data navzájem doplňují.

Využití prostředníka

Dá se rovněž udat, zda se pro synchronizaci využívá třetí databáze, či nikoliv, tak jak je popsáno v sekci 2.1.3.

Statistické údaje

Mimo uvedené parametry lze samozřejmě z každé synchronizace pořizovat i statistické údaje, jako je počet synchronizovaných tabulek, počet aktualizovaných řádků a podobně. O těchto měřeních se zde však nemá smysl rozepisovat.

2.2 Rozdíly mezi databázemi MySQL a MSSQL

V této kapitole vytyčíme hlavní rozdíly mezi oběma databázemi. Neboť popis všech drobných odchylek je zcela mimo rozsah této práce, uvedli jsme pro ilustraci několik rozdílů v tabulce 2.5. Uvedené odchylky jsou vybrány tak, aby pokrývaly celé spektrum oblastí od obecnějších po konkrétnější. Rozdíly v syntaxi dotazů uvedeny nejsou vůbec, neboť každá databáze využívá jiný SQL (Structured Query Language - Strukturovaný dotazovací jazyk) dialekt a tak čtenáře odkážeme na učebnice jazyků SQL 92 a T-SQL.

Všechny informace o fungování databází MySQL a MS SQL jsou v této kapitole — pokud není uvedeno jinak — čerpány z [4] a [7].

Stav v MySQL	**Stav v MS SQL**
Hierarchie databáze	
Databáze, tabulka, sloupec	Databáze, schéma, tabulka, sloupec
Ukládací prostředí (anglicky storage engines)	
MyISAM, InnoDb, MERGE, MEMORY, BDB, EXAMPLE, FEDERATED, ARCHIVE, CSV, BLACKHOLE	Jediné ukládací prostředí.
Otevřenost zdrojových kódů	
Do roku 2008 otevřené zdrojové kódy	Neveřejné zdrojové kódy
Komerční produkt?	
Ne	Ano
Dotazovací jazyk	
SQL-92	T-SQL
Podpora XML	
Ne	Ano
Možnost ukládání obrázků	
Pouze využitím textových datových typů	Ano
Možnost nastavovat přístupová práva dle sloupců	
Ne	Ano
Maximální délka názvu objektů	
64 znaků	128 znaků
Duplikátní názvy spouštěčů	
Ne	Ano
Nepojmenované spouštěče	
Ano	Ne
Přístup k objektům	
‘objekt‘ (uvozovky nepovinné)	[objekt]
Největší možná délka uložených dat	
Dle velikosti datového typu `longtext` a zvoleného ukládacího prostředí	Omezeno pouze volným místem na disku (datový typ `varchar(max)`)
Podpora přepínače `ZEROFILL`	
Ano	Ne
Podpora přepínače `UNSIGNED`	
Ano	Ne
Podpora spouštěčů prováděných před operací	
Ano	Ne
Nulová hodnota pro časové datové typy	
0000-00-00 00:00:00	1753 January 01
Podpora inicializační hodnoty pro časové datové typy	
Od verze 5.5	Ano
Možnost uložit neplatné datum	
Ano	Ne
Libovolný rozsah časových datových typů	
Ano	Pouze u `datettime2`

Speciální podpora pro unicode	
Ne	Ano
Uvozovky kolem řetězců	
Libovolné (', ", ‘)	Pouze „’“
Podpora výčtových datových typů	
Ano	Ne
Určování znakové sady	
Pomocí `CHARACTER SET` a `COLLATION`	Pouze `COLLATION`
Možnost uvádět integritní omezení při vytváření tabulky	
Ano	Částečně
Podpora ID sloupců	
Pomocí `AUTO_INCREMENT`	Pomocí `IDENTITY`
Možnost nastavit krok inkrementace pro každou tabulku zvlášť.	
Ne	Ano
Slučování tabulek	
Při použití ukládacího prostředí `MERGE`	Ne
Cizí klíče vedoucí mezi databázemi	
Ano	Ne
Možnost přesunovat tabulky mezi databázemi	
Ano	Ne
Omezování vybraných řádků	
Pomocí `LIMIT`	Pomocí `TOP` a `ROW_NUMBER()`
Odesílání výsledků transakce	
Ihned	Na příkaz
Podporované platformy	
Unix, Mac OS, Windows	Windows

Tabulka 2.5: Vybrané rozdíly mezi databázemi MySQL a MS SQL.

Na všechny rozdíly musí být při synchronizaci brán zřetel. Několik vybraných problémů si nyní rozebereme podrobněji spolu s popisem jejich možného řešení.

2.2.1 Rozdílné datové typy

MySQL a SQL Server pracují s rozdílnými datovými typy. Při kopírování tabulek z jedné databáze do druhé je proto nutné pro každý sloupec správně zvolit adekvátní datový typ z druhé databáze tak, aby nemohlo dojít ke ztrátě dat vlivem jiné (menší) velikosti zvoleného datového typu. U čísel s plovoucí řádovou čárkou je nutno dbát na dodržení správné přesnosti čísel. Tabulky B.1 a C.1 shrnují zvolené mapování mezi jednotlivými datovými typy.

MS SQL Podpora pro datové typy `ENUM` a `SET`

MS SQL na rozdíl od MySQL nezná výčtové datové typy `ENUM` a `SET`. `ENUM` je datový typ, který umožňuje uložení nejvýše jedné hodnoty z uzavřené množiny prvků M. `SET` rozšiřuje

tuto funkcionalitu o možnost uložit libovolné množství hodnot z M. Tyto datové typy tedy nelze emulovat zvolením jiného datového typu.

Pro `ENUM` je však možné vytvořit spouštěče, které zajistí danou funkcionalitu. Spouštěč pro vkládání do sloupce typu `ENUM` by mohl vypadat například takto:

```
CREATE TRIGGER [dbo].[enum_trigger]
   ON [dbo].[enum_table]
   AFTER INSERT
AS
BEGIN
        IF INSERTED.enum_column NOT IN ('value1', 'value2')
        BEGIN
                THROW 51000, 'Unknown selected value.', 1;
        END
        ELSE
                INSERT INTO [enum_table] ([enum_column])
                SELECT INSERTED.enum_column AS enum_column
                FROM INSERTED
END
```

Toto řešení však vyžaduje ruční změnu všech zúčastněných spouštěčů při každé změně tabulky. Při změně v M je rovněž nutno ručně upravit všechny spouštěče.

Daleko pružnější a elegantnější je modelovat typ `ENUM` pomocí další tabulky, ve které budou uloženy všechny hodnoty z M a z původní tabulky povede pouze cizí klíč ukazující vždy na jednu z těchto hodnot. Jedná se tedy o modelování pomocí vztahu 1:N.

V případě datového typu `SET` již nelze řešení pomocí triggerů užít vůbec, neboť do jednoho sloupce nelze v MS SQL uložit více hodnot najednou. Je tedy nutné užít ještě o jednu tabulku více než v případě `ENUM` a modelovat tuto situaci jako vztah M:N.

Řešení pomocí dalších tabulek umožňuje snadnou správu všech parametrů a uložených hodnot, je ovšem nutné, aby aplikace pracující s databází - které nevyužívají ORM (Object Relational Mapping - Objektově-relační mapování) - o této architektuře věděly.

2.2.2 Podpora přepínače `unsigned`

MySQL podporuje pro všechny číselné datové typy (tzn. `int`, `tinyint`, `smallint`, `mediumint`, `bigint`, `double`, `float` a `decimal`) volitelnou možnost `unsigned`[4]. SQL Server nikoliv [7]. Tato volba změní rozsah uložitelných hodnot z původního intervalu $[min; max]$ na interval $[0; 2 * max]$. To přináší několik vlastností, které je potřeba emulovat. Jsou to:

1. Zaručit možnost uložení všech validního hodnot (0–2*max).

   Řešení: Toto lze zajistit zvolením většího datového typu (např. místo `int unsigned` se použije `bigint`).

2. Zaručit, že do sloupce nepůjde uložit hodnota, která nepatří do výsledného intervalu.

   Řešení: Zde se nabízí řešení pomocí spouštěče, ve kterém se při každé aktualizaci vkládaná hodnota zkontroluje. Toto řešení je však použitelné pouze při práci se sloupci

tabulky. Při použití proměnné typu `unsigned` uprostřed procedury či spouštěče již nelze žádnými prostředky ošetřit, že se do této proměnné skutečně nevalidní hodnota nedostane.

Je také nutno řešit, jak na nevalidní hodnotu zareagovat. Vytvořit výjimku či ohlásit chybu? Ani jedna varianta není ideální, protože ukončí provádění celého SQL dotazu a tak silně naruší běh aplikace.

3. Dále je potřeba zajistit správné počítání s hodnotou `unsigned`. Zde se jedná o to, že operace s `unsigned` čísly mohou mít jiné výsledky než s čísly ze standardního rozsahu. Příklad pro počítání s operandy typu `int [unsigned]`:

Operace	`int unsigned`	`int`
$0 - 1 =$	18446744073709551615	-1
$6 - 5 =$	1	1
$10 - 100 =$	18446744073709551526	90
$4294967295 - 1 =$	4294967294	chyba

Tabulka 2.6: Příklady rozdílů mezi výsledky početních operací při použití operandů typu `int` a `int unsigned`.

Řešení: Vhodným a také elegantním řešením by bylo přetížení operátorů pro numerické operace, protože však T-SQL dialekt přetěžování operátorů neumožňuje [7], nelze žádným způsobem zajistit správné výsledky početních operací.

2.2.3 Názvy spouštěčů

MySQL umožňuje používat duplikátní názvy omezení (`constraints`) a spouštěčů[4]. Rovněž umožňuje název zcela vynechat[4]. SQL Server ani jednu z těchto vlastností nepodporuje[7].

Řešení: Je proto nutné vytvořit algoritmus, který bude pro každé omezení generovat jedinečný název, který se v databázi nevyskytuje.

2.2.4 Podpora `Identity` v tabulkách MySQL

Pro generování jedinečného ID záznamu se v SQL Server tabulkách používá vlastnost `IDENTITY[(seed, increment)]`, kde `seed` udává počáteční hodnotu a `increment` hodnotu, o kterou se každý další záznam zvětší[7].

U databází MySQL se pro generování jedinečného ID užívá vlastnost `AUTO_INCREMENT`. Přímé nastavení `seed` a `increment` však není podporováno.

Řešení: Nastavení počáteční hodnoty (`seed`) lze nastavit jednoduše příkazem:

```
ALTER TABLE `table_name` AUTO_INCREMENT=42
```

Je však vyžadováno, aby tabulka již existovala a aby nad sloupcem s `AUTO_INCREMENT` byl definován primární klíč.

Druhá vlastnost - `increment` - je v MySQL řešena bohužel velmi nešťastně. Nelze ji nastavit pro každou tabulku zvlášť. Pokud přesto chceme v MySQL nastavit pro několik

tabulek rozdílnou hodnotu inkrementace, musíme buď migrovat na jiný databázový server (na SQL Server například), nebo se pokusit vymyslet nějaké vlastní řešení.

Jako vhodné řešení by se mohlo zdát použití spouštěče. Při každém vložení do tabulky by se ručně dopočítala hodnota ID nového záznamu. MySQL však bohužel spouštěčům neumožňuje manipulovat se sloupci typu `AUTO_INCREMENT` [9]).

Nabízí se obejít toto omezení zrušením užívání vlastnosti `AUTO_INCREMENT` a jejím nahrazením standardním sloupcem typu `int` společně se simulováním automatické inkrementace pomocí vhodně navržených spouštěčů. Tímto způsobem by skutečně dané chování emulovat šlo. Bohužel při každém novém záznamu se musí projít všechny předchozí a nalézt největší z nich. S počtem řádků v tabulce tedy roste doba počítání nového ID (jistého vylepšení by šlo dosáhnout použitím indexu). Ještě větší problém však spočívá v atomicitě dané operace. Co se stane, pokud budou do tabulky vkládány dva záznamy současně? Provedení spouštěče není v MySQL atomické[2] a může se tak stát, že dva záznamy budou mít shodné ID. Užití indexu `UNIQUE` není v tomto případě řešením, neboť dojde k chybě a provádění celého SQL dotazu/procedury/spouštěče se zruší.

V souvislosti s atomicitou někoho možná napadne řešení pomocí transakcí, které atomicitu zajišťují. Znamenalo by to však, že místo standardního SQL zápisu pro manipulaci s daty tabulky:

```
INSERT INTO `table_name` (id, name, town, ...)
        VALUES (1, 'John', 'Philadelphia', ...);

-- respektive aktualizace
UPDATE `table_name`
        SET name = 'Peter', town = 'Prague' WHERE id = 1;
```

Budeme nutit uživatele, aby užíval:

```
CALL insert_into_table_name(1, 'name', 'John');
CALL insert_into_table_name(1, 'town', 'Philadelphia');
...

-- respektive pro aktualizaci
CALL update_table_name('name', 'Peter', 1);
CALL update_table_name('town', 'Prague', 1);
...
```

A toto je - dle mého soudu - požadavek natolik omezující, že jej nelze na uživatele klást. Mimo to rovněž nelze zajistit, že se uživatel tímto požadavkem skutečně bude řídit a nedojde tak k narušení integrity databáze.

Z těchto důvodů tedy **nelze v databázi MySQL věrohodně emulovat vlastnost `increment`**. Při provádění synchronizace tabulek, jež obsahují nestandardní hodnotu vlastnosti `increment` (tedy jinou než 1), je tedy na tento fakt potřeba uživatele upozornit.

[2]Ve skutečnosti se spouštěče v MySQL dají považovat za atomické, protože samotné uložení výsledků spouštěče atomické je. Spuštění a výpočet spouštěče však atomické není. V našem případě tedy může dojít k tomu, že výsledek dvou spuštění jednoho spouštěče se pokusí uložit stejný výsledek.

2.2.5 Rozdílné SQL dialekty

Databáze MySQL komunikuje pomocí SQL dialektu nazvaným SQL-92. Databáze SQL Server komunikuje pomocí SQL dialektu nazvaným T-SQL. Obě databáze své dialekty navíc značně rozšiřují oproti zavedeným standardům. Ke každému databázovému systému se tedy musí přistupovat pomocí jiného programovacího jazyka.

Z toho vyplývá, že nelze při kopírování tabulek jednoduše vzít vytvářecí kód tabulky z jedné databáze a spustit jej v databázi druhé. Je nutno získat podrobné informace o struktuře tabulky ze systémových tabulek uložených v `INFORMATION_SCHEMA` a ručně vytvářecí dotaz složit.

Výrazně složitější problém však nastává u převodu spouštěčů, procedur, funkcí, pohledů a rutin z jedné databáze do druhé. Všechny tyto **struktury totiž obsahují uživatelem vytvořený programový kód, napsaný v dialektu, jemuž druhá databáze nerozumí**.

Je tedy **nutno vytvořit překladač z SQL-92 do T-SQL a zpět**. Obecně řečeno: Vytvořit překladač z jednoho programovacího jazyka do jiného.

Lze využít faktu, že dané programové struktury jsou na databázovém serveru uloženy. A protože na databázový server nelze uložit strukturu s gramatickou chybou (ohlídáno zabudovanými mechanismy), lze s naprostou jistotou předpokládat, že načtené kusy kódu neobsahují gramatické chyby. Z vytvářeného překladače je tedy možno vypustit celou syntaktickou a značnou část sémantické analýzy.

2.2.6 Rozdílná hierarchie databázových serverů

Standardní relační databáze obvykle rozlišují několik úrovní, na kterých lze spravovat oprávnění či implicitní kódování. SQL server zná takovéto úrovně celkem čtyři:

1. Databáze
2. Schéma
3. Tabulka
4. Řádek

V MySQL ovšem nelze rozlišit mezi databází a schématem. Od verze 5.0.2 [9] jsou konstrukce typu `CREATE SCHEMA...` a `CREATE DATABASE...` synonyma. Je zde tedy o jednu úroveň méně. Z výše popsaného vyplývá, že na SQL serveru se mohou v jedné databázi vyskytovat tabulky se shodnými názvy (každá v jiném schématu), v MySQL nikoliv.

Je tedy nutno ošetřit názvy těchto tabulek tak, aby nebyly shodné. Rovněž je potřeba ošetřit adresování mezi schématy ve výběrových dotazech např. při provádění procedury. Při převodu z SQL Serveru na MySQL je tedy nutno pamatovat na to, že některé adresy mohou být nejednoznačné.

2.2.7 Rozdílný přístup ke znakovým sadám a porovnávání

Při práci s MySQL lze specifikovat implicitní znakovou sadu databáze, tabulky či sloupce pomocí příkazu `CHARACTER SET`. Porovnávání potom lze nastavit příkazem `COLLATE`. Například:

```
CREATE DATABASE `example_db`
        CHARACTER SET 'utf8' COLLATE 'utf8_czech_ci';
```

Při neuvedení jedné z těchto hodnot se tato automaticky doplní implicitní hodnotou pro zvolené porovnávání respektive znakovou sadu. Implicitní hodnoty lze vyčíst ze systémových tabulek `INFORMATION_SCHEMA.COLLATIONS` a `INFORMATION_SCHEMA.CHARACTER_SETS`.

SQL server tato dvě nastavení sdružuje do jednoho příkazu `COLLATE`[3]. Například:

```
CREATE DATABASE example_db COLLATE Latin1_General_CS_AS_KS_WS ASC;
```

Jak je vidět, neshodují se ani formáty řetězců reprezentujících jednotlivé znakové sady a způsoby porovnávání. Je tedy nutno zvolit vhodné mapování mezi jednotlivými databázemi pro zjištěná nastavení jak nakládat se znakovými sadami.

[3]Možné parametry pro příkaz `COLLATE` lze vyčíst například zde: http://technet.microsoft.com/en-us/library/aa258233(v=sql.80).aspx

Kapitola 3

Návrh algoritmu

Nyní dáme dohromady znalosti nabyté v předcházející kapitole a postupnými kroky navrhneme algoritmus pro synchronizaci libovolných databází.

3.1 Předpoklady pro vytvoření algoritmu

Pro stručnější zápisy následujících algoritmů si uveďme některé předpoklady, kterých budeme při psaní kódu využívat.

Všechny uvedené pseudokódy předpokládají objektově orientovaný programovací jazyk. Značně se tím zkrátí samotný zápis a navíc lze těžit z výhod objektově orientovaného kódu. Ten poskytuje názorné prostředky pro modelování vztahů mezi jednotlivými prvky, vstupujícími do synchronizace. Zvolený implementační jazyk výsledné aplikace i s dalšími argumenty pro objektově orientovaný kód lze zjistit v sekci 3.3.

Dále předpokládejme existenci dvou objektů: `TargetDb` a `SourceDb`. Ty reprezentují zdrojovou a cílovou databázi a jsou to instance nějaké třídy, nazvěme ji třeba `Database`. Tato třída tedy přirozeně poskytuje i rozhraní pro práci s databází. Mezi typické metody by mohly patřit například: `connect()`, `getTables()` či `removeTable(tableName)`.

Dále předpokládejme, že každá tabulka je v programu reprezentována objektem typu `Table`. Obdobně pak procedury jsou reprezentovány objekty typu `Procedure`, spouštěče objekty typu `Trigger` atd.

Pro další zkrácení zápisu pak předpokládejme, že všechny objekty jsou inicializovány a že spojení s jednotlivými databázovými servery je již bezpečně ustaveno.

3.2 Dávková jednosměrná synchronizace

V následujícím textu uvedeme a vysvětlíme algoritmy pro jednosměrnou, dávkovou synchronizaci. Budeme uvádět jak algoritmy pro synchronizaci metodou kompletního přepisu dat, tak algoritmy pro synchronizaci metodou postupné aktualizace (popis obou metod je uveden v kapitole 2.1.3).

3.2.1 Algoritmus pro synchronizaci libovolných tabulek

Základní algoritmus, provádějící synchronizaci tabulky metodou postupné aktualizace, je znázorněn v pseudokódu 1 (předpokládejme, že metoda `synchronizeTable()` je členem objektu `TargetDb`).

Algorithm 1: Algoritmus synchronizace tabulky metodou postupné aktualizace.

```
1  def synchronizeTable(sourceTable):
2      if not this.tableExists(sourceTable.name):
3          Vytvoř tabulku se strukturou tabulky sourceTable;
4          Objekt reprezentující vytvořenou tabulku ulož do targetTable;
5      else:
6          targetTable = this.getTable(sourceTable.name);
7          if targetTable.getStructure() != sourceTable.getStructure():
8              Aktualizuj strukturu targetTable;
9      if not targetTable.hasPrimaryKey() and not targetTable.hasUniqueIndex():
10         targetTable.empty()
11     var foundRows = new Array();
12     for row in sourceTable:
13         if row in targetTable:
14             targetTable.updateRow(row);
15         else:
16             targetTable.insert(row);
17         foundRows[] = row;
18     for row in targetTable:
19         if row not in foundRows:
20             targetTable.delete(row);
```

Popišme si stručně jeho činnost. Na řádku 2 se dotazujeme databáze T, zda synchronizovaná tabulka existuje. Pokud ne, tabulku vytvoříme (řádky 3 a 4). Tento krok zahrnuje mnoho operací, jako je například zvolení správných datových typů či správná implementace některých přepínačů (`ZEROFILL`, `UNSIGNED`, ...). Řešení některých z nich bylo popsáno v kapitole 2.2. Pokud naopak zjistíme, že tabulka již v databázi T existuje, musíme zajistit, že má správnou strukturu (řádky 6-8). Na řádku 9 pak zjišťujeme, zda lze v tabulce bezpečně identifikovat každý řádek. Pokud ne, musíme celou tabulku vyprázdnit (viz kapitola 2.1.3). Na řádku 11 si vytvoříme pole, do kterého si budeme ukládat všechny řádky, které byly nalezeny ve vzorové tabulce. Později toto pole využijeme. Řádky 12-16 zajišťují správnou aktualizaci či vložení řádku do tabulky v případě, že řádek v tabulce nebyl nalezen. Posledním úkolem je odstranění přebývajících řádků z tabulky (v kódu řádky 18-20). Zda má být řádek v tabulce i po synchronizaci rozlišíme snadno podle pole `foundRows`.

Pro metodu přepisu celé tabulky by algoritmus byl výrazně jednodušší, jak ukazuje pseudokód 2.

Strukturu tabulky nemusíme kontrolovat, neboť ji vždy vytváříme znova dle tabulky vzorové. Rovněž odpadá kontrola přebývajících řádků, neboť tabulka je připravena prázdná, a tak se v ní po skončení synchronizace budou skutečně vyskytovat pouze řádky nalezené ve vzorové databázi. Jak lze vidět, tento zdánlivě neelegantní algoritmus je výrazně jednodušší a také rychlejší.

Ani jeden z výše popsaných algoritmů však nebere v potaz možná existující integritní omezení, jako jsou cizí klíče či primární klíče. Do sloupce s primárním klíčem nelze ve standardních databázích ručně vkládat hodnotu. Další problém se může vyskytnout u cizích

Algorithm 2: Algoritmus synchronizace tabulky metodou kompletního přepisu dat.

```
1 def synchronizeTable(sourceTable):
2     if this.tableExists(sourceTable.name):
3         Odstraň celou existující tabulku;
4     Vytvoř tabulku se strukturou tabulky sourceTable;
5     Objekt reprezentující vytvořenou tabulku ulož do targetTable;
6     for row in sourceTable:
7         targetTable.insert(row);
```

klíčů vedoucích **do i ze** synchronizované tabulky. Pokud totiž smažeme řádek, na který vede odkaz z jiné (ještě neaktualizované) tabulky, dojde k chybě vyvolané kontrolou integritních omezení.

Tyto problémy lze vyřešit odstraněním všech integritních omezení kolem synchronizované tabulky. Toto odstranění však musí proběhnout na začátku synchronizace celé databáze (nikoliv před synchronizací konkrétní tabulky), neboť cizí klíč odkazující se z právě synchronizované tabulky může ukazovat na hodnotu, která bude do databáze vložena až při synchronizaci další tabulky. Po skončení celé synchronizace lze pak všechna omezení opět vytvořit.

Integritní omezení je tedy nutno odstranit a opět vytvořit před respektive po celé synchronizaci. Z tohoto důvodu tento krok zahrneme až do algoritmů v následující kapitole, která ukazuje, jak synchronizovat celé databáze.

3.2.2 Algoritmus pro synchronizaci libovolných databází

Nyní již máme vytvořen algoritmus pro synchronizaci libovolné tabulky. Můžeme tedy přikročit k algoritmu pro synchronizaci celých databází. Ten je uveden v pseudokódu 3.

Algorithm 3: Algoritmus synchronizace celé databáze.

```
1  def synchronizeDatabase():
2      ... incializace proměnných, připojení k databázím atd. ...
       /* Odstranění všech existujících omezení.                    */
3      targetDb.removeConstraints();
       /* Odstranění přebývajících tabulek.                         */
4      for table in targetDb.getTables():
5          if not sourceDb.tableExists(table.name):
6              targetDb.remove(table)
       /* Synchronizace tabulek                                     */
7      for table in sourceDb.getTables():
8          targetTable.synchronizeTable(table);
       /* Vytvoření validních integritních omezení.                 */
9      for constraint in sourceDb.getConstraints():
10         targetDb.create(constraint);
```

Opět si stručně popišme činnost algoritmu. Nejdříve odstraníme veškerá integritní ome-

zení z celé databáze. Tím si uvolníme ruce pro vkládání, odstraňování a aktualizování dat v databázi. Pořadí odstraňování je velmi důležité. **Nejdříve je potřeba odstranit cizí klíče, poté všechny ostatní indexy mimo primárních klíčů (unikátní, vyhledávací, ...) a až nakonec samotné primární klíče.** V tomto pořadí nehrozí narušení žádných integritních omezení. Po dokončení této činnosti srovnáme seznam existujících tabulek v jedné i v druhé databází a přebývající tabulky z databáze T odstraníme (řádky 4-6). Následně procházíme jednu tabulku databáze S po druhé a převádíme je do databáze T (řádky 7 a 8). Na závěr celé synchronizace vytvoříme zpět veškerá integritní omezení, tentokrát v opačném pořadí než při odstraňování.

V pseudokódu synchronizace databáze není z důvodu stručnosti uvedena synchronizace pohledů, funkcí, procedur a spouštěčů. Základní kód by byl však obdobný jako kód na řádcích 4 až 8 v algoritmu 3. Jediným rozdílem by byla práce s jinými databázovými objekty než s tabulkami (tedy `targetDb.getProcedures()`, `targetDb.getViews()` atd.). Tento kód by byl vložen před opětovné vytváření integritních omezení, tedy mezi řádky 8 a 9.

Činnost jednotlivých metod, volaných z pseudokódů se značně liší dle použité databáze. Práce některých z nich byla naznačena v kapitole 2.2. Přesnou implementaci a praktické řešení dílčích problémů si lze prohlédnout ve zdrojových kódech na přiloženém CD (Compact Disc - Kompaktní disk).

3.3 Algoritmus ve výsledné aplikaci

Kompletní algoritmus je možné shlédnout implementovaný ve výsledné aplikaci na přiloženém CD. V této aplikaci byl jako implementační jazyk zvolen Python a to především z následujících důvodů:

Přenositelnost U aplikace napsané v Pythonu není potřeba řešit kompatibilitu napříč stroji. Skripty jsou spustitelné v prostředí Unixu, Windows i Mac OS[6]. V některých Linuxových prostředí je dokonce instalován již v základní distribuci.

Dostupnost rozšíření Python — jakožto mimořádně populární jazyk — má k dispozici velké spektrum dostupných knihoven. Lze se tedy pomocí něj pohodlně připojit k mnoha databázovým serverům[6].

Vyšší programovací jazyk Python je vyšší skriptovací programovací jazyk a proto odpadá nutnost správy užívané paměti[6].

Strmá křivka učení Python je obecně považován za jazyk s jednou z nejstrmějších učebních křivek na světě[6].

Objektová orientace Python disponuje relativně vyspělým objektově orientovaným rozhraním, pomocí kterého lze výhodně modelovat všechny prvky účastnící se synchronizace (tabulky, pohledy, databáze, ...) a vztahy mezi nimi.

Implementována byla metoda kompletního přepisu celé databáze.

Pro aktualizaci dat na základě kompletního přepisu bylo rozhodnuto po praktickém měření časových nároků u kompletního přepisu a u obyčejné aktualizace (popis obou metod viz kapitolu 2.1.3). Metoda přepisu dat z tohoto porovnání vyšla výrazně lépe. Jako logické vysvětlení se zdá být počet a typ pokládaných dotazů. Tabulka 3.1 shrnuje pokládané dotazy u obou metod.

Příkaz	Metoda	
	Přepis	**Aktualizace**
`SELECT TABLE_NAME FROM TABLES`	ano	ano
`DROP TABLE ...`	ano	ne
`CREATE TABLE ...`	ano	ne
`ALTER TABLE ...`	ne	ano
`SELECT * FROM ...`	ano	ano
`INSERT ...`	ano	ano
`UPDATE ...`	ne	ano
`DELETE ...`	ne	ano

Tabulka 3.1: Porovnání použitých dotazů u metody kompletního přepisu dat a u metody postupné aktualizace existujících záznamů.

Jak je vidět, metoda přepisu databáze pokládá méně časově náročných dotazů (`DROP TABLE + CREATE TABLE` je rychlejší než `ALTER TABLE` nad plnou tabulkou). Důležitým faktem, který z tabulky 3.1 nelze vyčíst je, že příkaz `SELECT * FROM ...` (tedy s velkou převahou časově nejnáročnější) se u metody kompletního přepisu používá pouze pro tabulky databáze S. U metody postupné aktualizace se navíc používá i pro všechny tabulky databáze T. Metoda postupné aktualizace používá také další vysoce časově náročný dotaz `UPDATE`.

Rovněž u synchronizovaných databází nelze předpokládat existenci primárního klíče či unikátního indexu u každé tabulky. Z těchto důvodů se tedy ukázala metoda přepisu dat jako výrazně výhodnější.

Kapitola 4

Implementace - případové studie

Tato kapitola si klade za cíl demonstrovat na dvou zvolených případových studií implementaci výše zmíněných algoritmů a praktické řešení problémů popsaných v kapitole 2.2. K tomuto účelu byly vybrány dva široce rozšířené a dostatečně komplexní systémy s otevřenými zdrojovými kódy:

Drupal
: Jeden ze světově nejpoužívanějších CMS (Content Management System - Systém pro správu obsahu).

Prestashop
: Mimořádně populární systém pro tvorbu a správu internetových obchodů.

Jak se lze dále v této kapitole přesvědčit, oba produkty využívají velmi robustní databáze a měly by proto implementaci vhodně prověřit (poznámka: Nejedná se o testy aplikace. Ty jsou uvedeny samostatně na přiloženém CD).

4.1 Případová studie: Drupal

První zvolenou databází pro případovou studii je databázový systém užívaný aplikací Drupal.

4.1.1 Seznámení s Drupalem

Drupal je CMS, umožňující snadnou správu webových portálů, jejichž zaměření je díky vysoké rozšiřitelnosti systému (pomocí modulů) téměř neomezeno: blogy, magazíny, zájmové weby, internetové obchody či korporátní informační systémy [5].

Mezi weby poháněné právě tímto systémem patří například web britské televizní stanice MTV[8].

Z výše zmíněného tedy plyne, že se jedná o vysoce komplexní systém, který vyžaduje důkladně promyšlenou strukturu databáze. Pro ukázku práce aplikace je tedy tato databáze velmi vhodná.

4.1.2 Přehled použité databáze

Pro tuto případovou studii byla použita nejnovější distribuce Drupalu (v době psaní této práce se jedná o verzi 7.28), získaná z oficiálních zdrojů na adrese

https://drupal.org/project/drupal. Instalováno bylo standardní nastavení se zvolenou českou lokalizací (pro demonstraci práce se znakovými sadami). Do této instalace bylo pro testovací účely dále vloženo několik článků, stránek a komentářů.

Výsledná databáze obsahuje:

- 73 tabulek
- 383 omezení (včetně systémových klíčů)
- 462 sloupců
- 51 různých datových typů (respektive kombinací nastavení datového typu pomocí přepínačů `unsigned` a podobně)
- 1 687 řádků (včetně vloženého článku a komentáře)

Jedná se tedy o robustní databázi, která klade vysoké nároky na synchronizační aplikaci a prověří tak široké spektrum jejích aspektů.

4.1.3 Synchronizace

Převáděnou databázi lze shlédnout na přiloženém CD v souboru /case_studies/drupal/mysql.sql. Celá databáze vytvořená během synchronizace je k dispozici rovněž na přiloženém CD ve stejné složce, soubor sql_server.sql.

Na této případové studii si ukažme synchronizaci jedné z největších tabulek celé databáze — tabulky `block`. Tato tabulka je do originální databáze zanesena tímto SQL-92 kódem (komentáře byly pro zkrácení zápisu odstraněny):

```
CREATE TABLE `block` (
        `bid` int(11) NOT NULL AUTO_INCREMENT,
        `module` varchar(64) NOT NULL DEFAULT '',
        `delta` varchar(32) NOT NULL DEFAULT '0',
        `theme` varchar(64) NOT NULL DEFAULT '',
        `status` tinyint(4) NOT NULL DEFAULT '0',
        `weight` int(11) NOT NULL DEFAULT '0',
        `region` varchar(64) NOT NULL DEFAULT '',
        `custom` tinyint(4) NOT NULL DEFAULT '0',
        `visibility` tinyint(4) NOT NULL DEFAULT '0',
        `pages` text NOT NULL,
        `title` varchar(64) NOT NULL DEFAULT '',
        `cache` tinyint(4) NOT NULL DEFAULT '1',
        PRIMARY KEY (`bid`),
        UNIQUE KEY `tmd` (`theme`,`module`,`delta`)
) ENGINE=InnoDB AUTO_INCREMENT=31 DEFAULT CHARSET=utf8;
```

Převod datových typů

V této tabulce se nachází celkem 12 sloupců, které využívají 7 různých datových typů. Datový typ `int` se v databázi SQL Server nachází a má i shodný rozsah jako v databázi MySQL. Avšak ve sloupci `bid` je zobrazovaná délka omezena na 11 míst. SQL Server však takovou funkcionalitu neposkytuje[7]. Lze se však pohlédnout po menším datovém typu, do kterého by se 11místné číslo dalo uložit a tím ušetřit prostor na disku. SQL Server však žádný takový datový typ neposkytuje[7], proto musíme použít `int`.

Dalším datovým typem je `varchar(64)`. SQL Server opět datový typ `varchar` zná. Jeho použití však není vhodné, protože pracuje s ne-unicodovými řetězci[7], zatímco MySQL standardně všude pracuje s unicode řetězci. Použijeme proto datový typ `nvarchar`. Počet znaků lze nastavit obdobně jako u MySQL. Mimo `nvarchar` by pro délku 64 bylo možné využít i datový typ `nchar(64)`. Tento typ by na disku zabíral také o 2 B méně, avšak my zůstaneme u `nvarchar`, aby byly databáze co možná nejvíce identické. Datový typ `varchar(32)` lze převést zcela analogicky.

Pro číselný datový typ `tinyint(4)` se nabízí varianta nepřevádět jej, neboť SQL Server rovněž tímto datovým typem disponuje. Avšak `tinyint` na SQL Serveru má rozsah $[0; 255]$, v databázi MySQL to je $[-128; 127]$[7],[9]. Musíme tedy použít datový typ `smallint`, zobrazovanou délku opět nelze na SQL Serveru nastavit.

Posledním datovým typem je `text`. Do tohoto typu lze uložit až 65 535 B[9]. Takto velký řetězcový datový typ SQL Server neposkytuje. Využijeme proto zajímavého parametru `max`, který říká, že datový typ může pojmout takové množství dat, kolik je na disku místa. Výsledný datový typ tedy bude `nvarchar(max)`.

Vlastnost `NOT NULL` je u obou databází shodná, není tedy třeba ji zvlášť ošetřovat.

Převod integritních omezení

Jak lze vidět ve výše uvedeném kódu tabulky, MySQL podporuje uvádění integritních omezení přímo při vytváření tabulky. SQL Server pouze částečně[7].

Hodnotu `AUTO_INCREMENT` převedeme na `IDENTITY(1,1)`. Parametry v závorkách nemusíme u tohoto případu nikterak ošetřovat, neboť u tabulky není nastavena žádná nestandardní hodnota.

Dalšími integritními omezeními je uvedení implicitních hodnot pro jednotlivé sloupce. V databázi SQL Server lze toto omezení nastavit příkazem:

```
ALTER TABLE [schema].[block] ADD DEFAULT ('0') FOR [delta]
```

Uzavření názvů všech objektů do hranatých závorek je u tohoto typu databáze vyžadován (na rozdíl od MySQL). Obdobně bychom nastavili výchozí hodnoty pro všechny ostatní sloupce.

Použití `IDENTITY` u sloupce `bid` rovněž vyžaduje zřízení primárního klíče nad tímto sloupcem. Toto omezení tentokrát lze uvést přímo do vytvářecího kódu tabulky. MySQL však nevyžaduje pojmenování tohoto klíče. SQL Server ano. Proto musíme vygenerovat nový název, který se v databázi nesmí vyskytovat. Celý kód může vypadat například takto:

```
CONSTRAINT [pk_block_bid_PRIMARY] PRIMARY KEY ([bid])
```

Dále se v tabulce vyskytuje složený unikátní klíč. Musíme proto tabulku upravit ještě o toto integritní omezení. Generovat název není v tomto případě nutné, neboť je již určen. Kód:

```
ALTER TABLE [dbo].[block]
        ADD CONSTRAINT [tmd]
        UNIQUE ([theme], [module], [delta])
```

Synchronizace dalších parametrů tabulky

U MySQL kódu je dále uvedeno užití ukládacího prostředí InnoDB. SQL Server však podporuje pouze jedno ukládací prostředí a tak nemáme jinou možnost, než tuto volbu akceptovat.

Posledním parametrem je volba kódování. I díky použití datového typu `nvarchar` je zajištěno, že tabulka bude s daty ve formátu UTF-8 pracovat správně. Je však dobré mít na paměti, že data nebudou ve formátu UTF-8 uložena, protože SQL Server UTF-8 nepoužívá. Místo toho je použito kódování UTF-16[7], což však v našem případě znamená jen to, že uložená data budou mít dvojnásobnou délku.

Výsledný kód

Tím je synchronizace struktury tabulky dokončena. Celý T-SQL kód pak tedy může vypadat například takto:

```
CREATE TABLE [schema].[block](
        [bid] [int] IDENTITY(1,1) NOT NULL,
        [module] [nvarchar](64) NOT NULL,
        [delta] [nvarchar](32) NOT NULL,
        [theme] [nvarchar](64) NOT NULL,
        [status] [smallint] NOT NULL,
        [weight] [int] NOT NULL,
        [region] [nvarchar](64) NOT NULL,
        [custom] [smallint] NOT NULL,
        [visibility] [smallint] NOT NULL,
        [pages] [nvarchar](max) NOT NULL,
        [title] [nvarchar](64) NOT NULL,
        [cache] [smallint] NOT NULL,
        CONSTRAINT [pk_block_bid_PRIMARY] PRIMARY KEY ([bid])
);

ALTER TABLE [schema].[block] ADD DEFAULT ('0') FOR [delta];
...

ALTER TABLE [schema].[block]
        ADD CONSTRAINT [tmd]
        UNIQUE ([theme], [module], [delta])
```

4.2 Případová studie: Prestashop

Nyní si demonstrujme synchronizaci na výrazně rozsáhlejším, avšak jednoúčelovém systému Prestashop.

4.2.1 Důvody pro synchronizaci Prestashopu

Prestashop je profesionální systém pro vedení robustních internetových obchodů. Jako takový musí ve své databázi zajistit bezpečné uložení všech soukromých dat klientů, podrobných informací o každé provedené transakci či realizovat množství vazeb mezi entitami vystupujícími v obchodě.

4.2.2 Přehled použité databáze

Těmto požadavkům odpovídá také složitá struktura použité databáze:

- 270 tabulek
- 749 omezení (včetně systémových klíčů)
- 1 595 sloupců
- 99 různých datových typů (respektive kombinací nastavení datového typu pomocí přepínačů `unsigned` a podobně)
- 6 095 řádků

Tato čísla odpovídají standardní instalaci Prestashopu verze 1.6.0. Pro testovací účely tentokrát nebylo třeba databázi plnit daty, neboť základní instalace již dostatek ukázkových záznamů obsahuje.

4.2.3 Synchronizace

Celou zdrojovou databázi lze opět shlédnout na přiloženém CD, v souboru /case_studies/prestashop/mysql.sql. Převedenou databázi lze najít v téže složce, soubor sql_server.sql.

V této případové studii si ukažme, jak byly v databázi SQL Server emulovány výčtové datové typy. Konkrétně budeme převádět typ `ENUM`.

Zdrojová data

Datový typ `ENUM` se vyskytuje například v rozsáhlé tabulce `ps_product`. Definice sloupce s tímto datovým typem je následující:

```
...
condition enum('new','used','refurbished') NOT NULL
...
```

Realizace vztahu 1:N

Datový typ `ENUM` v praxi modeluje vztah 1:N. Z toho vyplývá, že budeme potřebovat ještě jednu tabulku navíc. Řádky této tabulky pak budou reprezentovat jednotlivé hodnoty uvedené v závorce při definici datového typu.

Nová tabulka musí mít dostatečně unikátní název, aby nehrozila kolize s již existující tabulkou, či s některou tabulkou v budoucnu vytvořenou. Jako vhodné se k tomuto účelu jeví použití prefixu „enum_“.

Rovněž tabulka musí obsahovat index, pomocí kterého bude moci být realizována vazba. Celá tabulka pak může vypadat například takto:

```
CREATE TABLE [schema].[enum_ps_condition_type](
        [id] [int] NOT NULL,
        [value] [varchar](max) NOT NULL,
       CONSTRAINT [pk_enum_ps_condition_type_id_enumtype]
                PRIMARY KEY
);

INSERT INTO [schema].[enum_ps_condition_type] ([id], [value])
VALUES (1, N'new'), (2, N'used'), (3, N'refurbished');
```

Definici původního sloupce typu `ENUM` pak zaměníme tak, aby na něj bylo možno navázat cizí klíč.

```
...
[condition] [int] NOT NULL
...
```

Nakonec zbývá pouze vytvoření onoho cizího klíče. Opět je potřeba volit dostatečně unikátní název integritního omezení.

```
ALTER TABLE [schema].[ps_products]
ADD CONSTRAINT fk_enum_condition
FOREIGN KEY ([condition])
REFERENCES [schema].[enum_ps_condition_type]([id])
```

Tím je realizace struktury databáze podporující výčtový datový typ hotova.

Kapitola 5

Závěr

V této práci byl prezentován obecný algoritmus pro obousměrnou synchronizaci databází SQL Server a MySQL metodou přepisu dat. Tento algoritmus byl poté implementován v konzolové aplikaci a demonstrován ve dvou případových studiích.

Byly stanoveny základní kroky a jejich pořadí pro bezpečnou synchronizaci dat. Tomu předcházela analýza rozdílů mezi synchronizovanými databázemi a u vybraných odchylek i rozebrána možná řešení.

Jako vedlejší výsledek byl teoretickému rozboru podroben obrat „synchronizace databází“, jehož výsledkem bylo rozdělení synchronizací podle několika kritérií, stanovení jejich principů a nutných kroků k jejich úspěšnému provedení.

Bylo rovněž vysvětleno, z jakých důvodů je potřeba synchronizaci provádět a že jejím správným provedením lze dosáhnout výrazných úspor nejen z ekonomického hlediska, ale i z hlediska lidských zdrojů. Rovněž je ušetřeno velké množství času, který by jinak byl stráven tvorbou již jednou vytvořené databáze.

Jako další směr, kudy by se mohla práce ubírat, vidím především možnost zobecnit výzkum na libovolné, nebo alespoň nejpoužívanější databáze. Této myšlence je uzpůsobena i výsledná aplikace, která svou strukturou umožňuje kdykoliv připojit k podporovaným databázím moduly pro podporu databází nových.

Literatura

[1] DB-Engines Ranking [online]. `http://db-engines.com/en/ranking`.

[2] Synchronizace - Wikipedie [online]. `http://cs.wikipedia.org/wiki/Synchronizace`.

[3] *Akademický slovník cizích slov A-Ž*. Academia, 2000, iSBN 80-200-0982-5.

[4] Official MySQL manual [online]. `http://www.mysql.com/`, 2014.

[5] Buytaert, D.: Oficiální prezentace systému Drupal [online]. `https://drupal.org/`.

[6] Guttag, J. V.: *Introduction to Computation and Programming Using Python*. The MIT Press, 2013, iSBN 978-0-262-51963-2.

[7] Lacko, L.: *Mistrovství v SQL Server 2012*. Computer Press, 2013, iSBN 978-80-2513-773-4.

[8] Robbins, J.: Press Release: Lullabot, Drupal, and MTV UK [online]. `http://www.lullabot.com/blog/article/press-release-lullabot-drupal-and-mtv-uk`.

[9] Widenius, M.; Axmark, D.: *MySQL Reference Manual*. O'Reilly Community Press, 2002, iSBN 978-0-596-00265-7.

Příloha A

Obsah CD

/case_studies
Zdrojové kódy převedených databází z kapitoly 4.

/src
Adresář se zdrojovými soubory synchronizační aplikace.

/tests
Adresář s ukázkami testovacích skriptů v jazycích T-SQL a SQL-92.

/tex
Adresář se zdrojovými soubory této práce.

/readme.txt
Pokyny pro spuštění synchronizační aplikace.

/thesis.pdf
Tato práce ve formátu PDF (Portable Document Format - Přenosný formát dokumentů).

Příloha B

Mapování datových typů MySQL na datové typy MS SQL

MySQL datový typ	Odpovídající MS SQL datový typ
`tinyint`	`smallint`
`smallint`	`smallint`
`mediumint`	`int`
`int`	`int`
`bigint`	`bigint`
`decimal[(`p `[,` s`])]`, kde $p \in \langle 0; 38 \rangle$	`decimal[(`p `[,` s`])]`
`decimal[(`p `[,` s`])]`, kde $p > 38$	`float[(`p `[,` s`])]`
`float(`p`)`	`float(`p`)`
`float[(`p`,` s`)]`	`float(53)`
`double[(`p`,` s`)]`	`float(53)`
`date`	`date`
`datetime`	`datetime2`
`timestamp`	`smalldatetime`
`time`	`time`
`year`	`smallint`
`char[(`l`)]`	`nchar[(`l`)]`
`varchar[(`l`)]`, kde $l < 8000$	`nvarchar[(`l`)]`
`varchar[(`l`)]`, kde $l > 8000$	`nvarchar(max)`
`tinytext`	`nvarchar(255)`
`text[(`l`)]`, kde $l < 8000$	`nvarchar[(`l`)]`
`text[(`l`)]`, kde $l > 8000$	`nvarchar(max)`
`mediumtext`	`nvarchar(max)`
`longtext`	`nvarchar(max)`
`bit[(`l`)]`	`varbinary[(`l`)]`
`binary[(`l`)]`	`binary[(`l`)]`
`varbinary[(`l`)]`	`varbinary[(`l`)]`
`tinyblob`	`varbinary(255)`
`blob[(`l`)]`, kde $l < 8000$	`varbinary[(`l`)]`
`blob[(`l`)]`, kde $l > 8000$	`varbinary(max)`
`mediumblob`	`varbinary(max)`
`longblob`	`varbinary(max)`
`enum`	tabulka + cizi klíč
`set`	2 tabulky + cizí klíče

Tabulka B.1: Mapování MySQL datových typů na SQL Server datové typy.

Příloha C

Mapování datových typů MS SQL na datové typy MySQL

MS SQL	MySQL
int	int
tinytint	tinyint
smallint	smallint
bigint	bigint
float	float
real	float
numeric	decimal
decimal	decimal
bit	tinyint(1)
binary	binary
varbinary	varbinary(255)
money	decimal
smallmoney	decimal
char(l), kde $l < 256$	char(l)
char(l), kde $l >= 256$	longtext(l)
char	longtext
nchar(l), kde $l < 256$	char(l)
nchar(l), kde $l >= 256$	longtext(l)
nchar	longtext
varchar(max)	longtext

MS SQL	MySQL
varchar(l), kde $l < 256$	varchar(l)
varchar(l), kde $l >= 256$	longtext(l)
varchar	longtext
nvarchar(l), kde $l < 256$	varchar(l)
nvarchar(l), kde $l >= 256$	longtext(l)
nvarchar	longtext
text	longtext
ntext	longtext
date	date
datetime	datetime
datetime2	datetime
smalldatetime	datetime
datetimeoffset	datetime
time	time
timestamp	timestamp
rowversion	timestamp
image	tinyblob
uniqueidentifier	varchar(160)
sysname	varchar(160)
xml	text

Tabulka C.1: Mapování MS SQL datových typů na MySQL datové typy.