Увод в системите бази от данни

Базите от данни са в основата на всеки бизнес. Освен това, те намират редица приложения при научни изследвания.
Базите от данни се управляват от специален софтуер, който ще наричаме **СУБД** (система за управление на бази от данни).
СУБД представлява мощен инструментариум за създаване и ефективно управление на големи обеми от данни. Данните трябва да се поддържат и съхраняват за толкова време за колкото е необходимо. Освен това, те трябва да се предпазват от неправилен достъп, който може да наруши целостта им (integrity), както и от неправомерен достъп (security).
Поради това, в наши дни СУБД е доста сложен софтуер.
Примери за СУБД са **Oracle**, **DB2**, **Microsoft SQL server**. Първите две системи, за разлика от третата се използват в сериозни организации.

По-нататък под **база от данни** ще разбираме наборът от данни, които се поддържат в даден момент от една СУБД.

От потребителска гледна точка една СУБД трябва да притежава следните свойства:

- **устойчива памет** – данните трябва да се съхраняват независимо от процесите; базите от данни трябва да позволяват ефективен достъп до много големи обеми от данни – времето на достъп до данните трябва да е независимо от техния обем;
- **програмен интерфейс** – СУБД предоставя мощен език за заявки на потребителя или на приложната програма, която използва данните;
- **управление на транзакции** – СУБД поддържа едновременен достъп до данните; той се осъществява чрез множество процеси, наречени **транзакции**; всеки потребител, който работи с данните трябва да инициира такъв процес; СУБД поддържа изолация на транзакциите – всяка транзакция е **атомарна**, т.е. тя или завършва изцяло или бива изцяло отхвърлена; ако по време на изпълнение на една транзакция възникне проблем тя се отменя, заедно с всички изменения, които е причинила върху базата от данни;
- **устойчивост на данните** – при възникване на аварии или грешки данните трябва да могат да бъдат възстановени; естествено, това е свързано с допълнителни операции за четене и писане.

От архитектурна гледна точка една СУБД трябва да предоставя следните функции:

- СУБД позволява на потребителя да създава нови бази от данни; за целта потребителят задава **схема** на базата от дании; схемата е логическа структура на данните и тя се описва с помощта на специален **език за дефиниция на данните** (**DDL** – data definition language); този език е символен или графичен;
- СУБД трябва да осигури възможност за търсене в данните и за модификация на данните; това се постига с помощта на **език за**

манипулация с данните (**DML** – data manipulation language); **езикът за заявки** е подезик на езика за манипулация с данните, който се използва при търсене на данни;
- СУБД трябва да съхранява безопасно и дългосрочно много големи обеми от данни; съхраняването трябва да осигурява ефективно търсене и изменение на данните;
- СУБД трябва да управлява едновременен достъп до данните от много потребители - те не трябва да си влияят един на друг и да развалят целостта на данните.

Еволюция на СУБД

Първите СУБД се появяват през 60-те години като резултат от развитието на файловите системи най-вече в посока към безопасно и дългосрочно съхранение. Те намират приложение при резервацията на самолетни билети, при банковите системи, при корпоративните записи и др. При тях визуализацията на данните е така както те се съхраняват. Естествената еволюция на файловите системи доведе до поява на следните два модела:
- **йерархичен модел** – например IMS на IBM; при него данните са във вид на дърво; в действителност някои данни може да присъстват в повече от едно дърво, ако към тях се използват връзки (reference);
- **мрежови модел** – този модел се появи по-късно; при него навигацията в данните е на значително по-високо ниво.

През 70-те години за пръв път се появиха СУБД, които се подчиняват на така наречения **релационен модел**, който измести напълно другите два модела. Този модел е разработен от Тед Код (Ted Codd).
При релационните СУБД за първи път се появява език за заявки от високо ниво.
Релационният модел почива на сериозни математически основи, което стимулира серия от изследвания в областта на базите от данни.
Този модел е изключително прост от гледна точка на потребителя – използва се само една структура от данни, наречена **таблица** (**релация**). Реализацията на таблицата е скрита за потребителя. Използват се езици за заявки от високо ниво. Примерна таблица:

accountNo	balance	type
01234	1000.00$	savings
56789	3849.40$	checking

В таблицата всяка колона си има име, което се нарича **атрибут** (accountNo, balance, type). Всеки ред в таблицата се нарича **кортеж** (имаме два кортежа). В рамките на една колона стойностите са еднотипни.

Езикът **SQL** (**structured query language**) е разработен във връзка с реализацията на релационния модел. В наши дни по стандарт всички СУБД използват SQL. Направени са изследвания, които показват, че най-честите заявки са в рамките на една таблица. Поради тази причина, основната конструкция, която се използва в SQL е следната:
SELECT име_на_атрибут
FROM име_на_таблица
WHERE условие
Условието може да има следния вид *име_на_атрибут=стойност.*
В зависимост от типа вместо равенство може да се използва някой друг предикат за сравнение. Няколко условия могат да се комбинират с помощта на логическите съюзи.
Ще опишем как се изпълнява конструкцията. В таблицата, зададена във FROM клаузата се извършва търсене по всички кортежи, като се проверява условието, зададено в WHERE клаузата. SELECT клаузата определя кои атрибути на намерените кортежи да бъдат изведени.
Примери: (предполагаме, че горната таблица се казва Accounts)
SELECT balance
FROM Accounts
WHERE accountNo=01234
Тук ще се изведе 1000.00$.
SELECT accountNo
FROM Accounts
WHERE type='checking' AND balance > 0
Тук ще се изведе 56789.

СУБД оптимизират заявките за да може те да бъдат ефективно изпълнени.

През средата на 90-те години се появиха обектно-ориентирани СУБД. Въпреки че не успяха да се наложат, те повлияха на релационния модел и в наши дни СУБД работят с **обектно-релационен модел**. Този модел е разширение на обектно-ориентирания модел, но запазва пълна съвместимост с релационния модел. Обектно-ориентираният подход е водещ при проектирането на софтуер и данни в наши дни.
Има тенденция СУБД да са вградени в операционните системи и в бъдеще да изместят файловите системи.

СУБД се развиват в посока на съхранение на данните във връзка с мултимедията, която е много голям по обем (например от порядъка на терабайтове). Концепцията на работа на една СУБД е, че данните се обработват в оперативната памет, а се съхраняват върху дискове. Проблемът при големите бази данни е, че дисковете все още не са достъчно големи. Поради тази причина се промени моделът за съхранение на данните. Той беше на две нива – оперативна памет и вторична памет. Сега вече се появи и **третична памет** – шкафове с носители, които се управляват от роботи (например Juke Box, лентови силози). Недостатъкът е, че времето за достъп до третичната памет е много голямо – приблизително няколко секунди, за разлика от вторичната памет, при която това време е около 10-20 ms.

За да може да се оптимизира управлението на различните памети се използват **индексна структура** на данните или **паралелизъм** в няколко варианта. Паралелизмът може да означава паралелно работещи процесори, паралелно работещи дискове. Възможно е използването на така наречените **разпределени системи** – при изпълнението на една заявка тя се разбива на няколко части, които се обработват от различни разпределени компютърни системи.

Концепции в развитието на СУБД

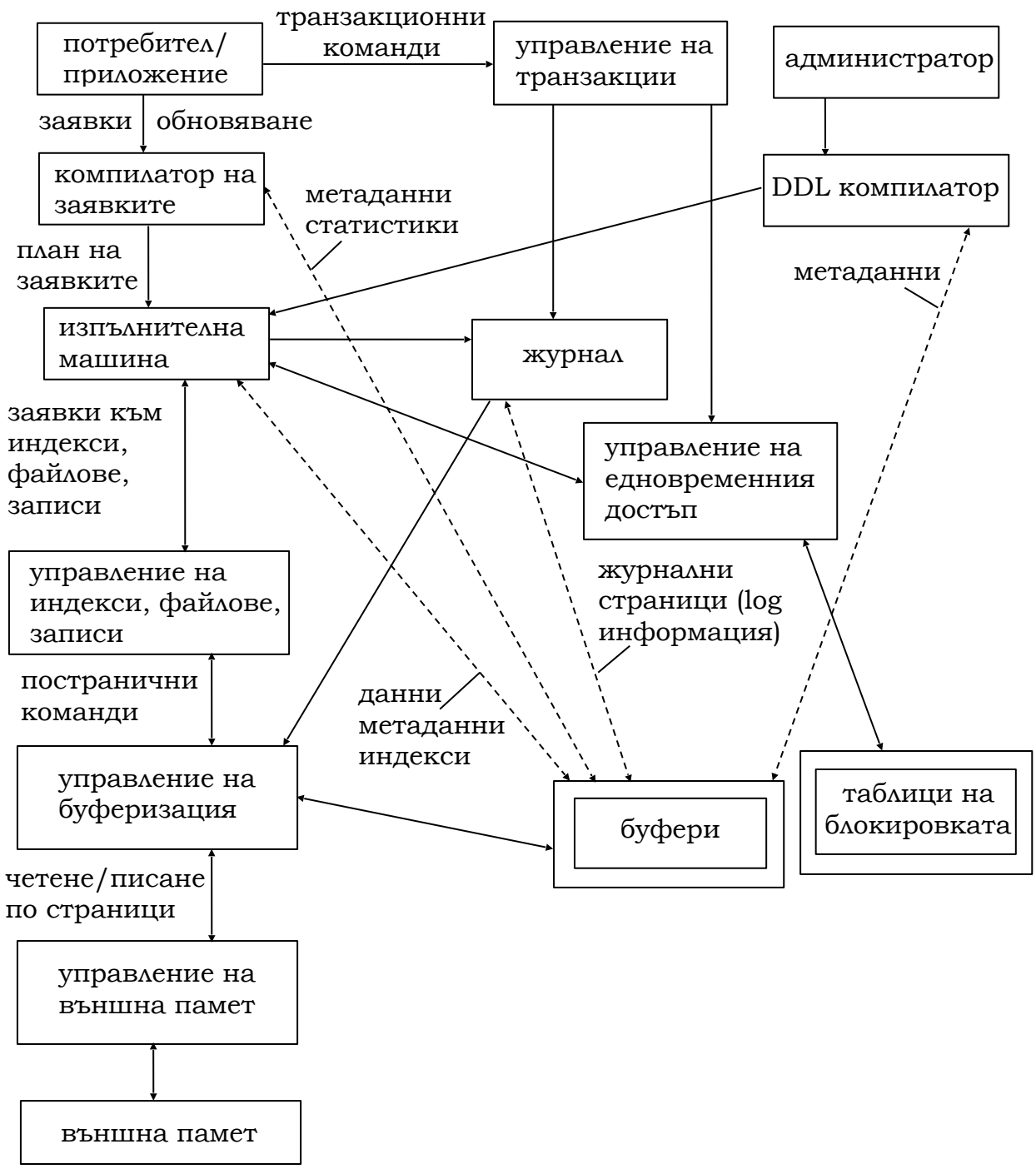
Client-server е архитектура, при която един процес (**клиент**, **client**) изпраща заявка за изпълнение към друг процес (**сървър**, **server**). Когато сървърът изпълни заявката, той връща данните на клиента. Естествено, СУБД е сървърът, а клиентите са потребителите или приложните програми. При такава организация се постига полезно натоварване на мрежата.
Проблем при архитектурата client-server е претоварването на сървърите. Поради тази причина се налагат архитектури client-server с множество нива. При тях СУБД отново е сървърът, но негови клиенти са така наречените **приложни сървъри**. Те имат за клиенти **уеб-сървъри**, които вече взаимодействат с потребителите. Приложните сървъри имат за задача да реализират бизнес-логиката на СУБД. Чрез такова разслоение отговорността за изпълнение на заявките се поделя между различни компютърни системи.

Един проблем на съвременните СУБД е търсене в мултимедийни данни. Възможно е мултимедийните обекти да се индексират с ключове, по които да се осъществява търсенето. Такъв подход, обаче, е неприложим в следните ситуации – търсене на кадър от филм, търсене на фрагмент от картинка и т.н.
Обектно-релационният модел поддържа операции за работа с мултимедийни обекти. Стремежите са данните да се визуализират пред потребителя във виртуална реалност.

Има тенденция когато една база от данни се създаде, тя да може занапред да съществува самостоятелно. За целта се използват **складове за данни** (**data warehouse**), в които информацията се интегрира и унифицира. Във връзка с постиженията в изкуствения интелект са разработени така наречените **OLAP технологии** (**OLAP technologies**), които позволяват ефективен анализ на данните в складовете. С тези технологии може да се осъществява търсене по образци (**data mining**).

Структура на СУБД

DDL команди


потребител/
приложение
транзакционни
команди
управление на
транзакции
администратор
заявки
обновяване
компилатор на
заявките
метаданни
статистики
DDL компилатор
план на
заявките
метаданни
изпълнителна
машина
журнал
заявки към
индекси,
файлове,
записи
управление на
едновременния
достъп
управление на
индекси, файлове,
записи
журнални
страници (log
информация)
постранични
команди
данни
метаданни
индекси
управление на
буферизация
буфери
таблици на
блокировката
четене/писане
по страници
управление на
външна памет
външна памет

Потребителят дава заявка към системата. Това става в интерактивен режим. Възможно е това взаимодействие да се осъществява чрез приложение. Самите заявки не променят базата от данни, докато при обновяване тя се променя.

Компилаторът на заявките изяснява дали синтаксисът на заявките е правилен и след това проверява тяхната семантика. В крайна сметка компилаторът на заявките изработва **план на заявката**. Обикновено този план е във вид на дървовидна структура. Във възлите на структурата стоят операции от релационната алгебра, а в листата се намират константи, имена на атрибути или цели таблици. Тъй като заявките са от високо ниво се изисква те да бъдат оптимизирани. Това е

функция на компилатора на заявките. Всъщност, той се състои от три части:

- синтактичен анализатор;
- семантичен анализатор;
- оптимизатор на заявки.

При оптимизиране на заявките възниква един фундаментален проблем – езиците за заявки са ориентирани за сравнително прости заявки. Например, SQL е разработен върху концепцията, че най-често заявките са в рамките на една таблица, по-рядко върху две таблици и много рядко върху три или повече таблици. При оптимизиране на проста заявка може да се загуби ефективност – времето за оптимизиране да надхвърля многократно времето за изпълнение на заявката. Обикновено в интерактивен режим заявките не се оптимизират. Когато обаче една заявка се съхранява, т.е. тя ще се изпълнява много пъти, то тя се оптимизира. Освен самият сорс на заявката се съхранява и нейният оптимизиран план (естествено, оптимизацията се извършва само веднъж при първото изпълнение на заявката).

Изпълнителната машина има за задача да изпълнява заявките по плана, който е предоставен от компилатора на заявките. Изпълнителната машина изпраща заявки към файлове и заявки към индекси за да извърши своята работа. Всъщност, изпълнителната машина използва различни методи на достъп, познавайки вътрешната структура и организация на физическо ниво на таблиците. Независимо от метода на достъп, той се поддържа във вид на страници.

Външната памет реално е организирана на страници, т.е. четенето и писането се извършва постранично. Размерът на една страница може да е 4K, 16K и др. Обикновено данните се прехвърлят между оперативната памет и външната памет. При СУБД ефективността на една заявка се измерва в това колко постранични операции (прехвърляния) се извършват при нейното изпълнение.

Управлението на външната памет реално извършва четенето и писането върху външната памет.
Управлението на буферизацията се занимава с това дали търсените страници не се намират в буферите на оперативната памет. По този начин се спестяват дискови операции.

Обикновено оптимизациите на заявките се извършват независимо, т.е. съобразно само една заявка. Има СУБД при които в компилатора на заявките постъпват по няколко заявки едновременно, които се компилират в общ план и съответно върху тях се извършва обща оптимизация. Ефективността на такава стратегия силно зависи от управлението на буферизацията.

Транзакциите са процесите в СУБД. Изпълнението им трябва да бъде такова, че те да не си влияят една на друга и да изглежда все едно се изпълняват последователно една след друга.

Когато един потребител работи интерактивно със системата, всяка негова заявка се счита за една транзакция. Той, обаче, може да изпълнява няколко заявки в рамките на една транзакция с помощта на специални команди. Има и други команди за управление на транзакциите.

Модулът управление на транзакциите използва друг модул – **журнал**, в който се регистрират транзакциите. По време на изпълнението на една транзакция, изпълнителната машина записва в журнала всякакви промени, които извършва върху транзакцията. Грубо казано, журналът се състои от записи. Всеки запис има нова част и стара част. Изпълнителната машина извършва промените върху новата част. Ако по време на изпълнение на транзакцията възникне проблем, т.е. тя не успее да завърши, чрез старата част на записа се възстановява старото състояние на базата от данни.

Администраторът на базата от данни отговаря за нейната структура. Освен това, той може да дава различни права на достъп (например read, write, read/write) на други потребители. Администраторът дефинира схемата на базата от данни като подава DDL-команди на DDL-компилатора. Самите DDL-команди след компилация се изпълняват от изпълнителната машина.

Управлението на едновременния достъп е модул, който има за задача да гарантира атомарността на транзакциите. Това става с помощта на таблици на блокировката, които се намират в оперативната памет. Когато една транзакция се обработва от модула управление на транзакциите, то е известно кои таблици ще се използват за четене/ писане в рамките на транзакцията. Тогава този модул се обръща към управлението на едновременния достъп, при което се блокират съответните таблици (релации). В таблицата на блокировката се съдържа информация кои от релациите са блокирани само за четене или за четене и за обновяване. Ако се прави опит за повторно блокиране за четене, няма проблем, т.е. от една релация едновременно могат да четат много транзакции. От друга страна, само една транзакция може да блокира релация за обновяване, т.е. ако една транзакция пише в релация, то други транзакции нямат достъп до тази релация (нито за четене, нито за обновяване). Най-често блокировката се извършва на отделни релации. Възможно е блокировката да се извършва на ниво кортежи, т.е. върху една и съща релация може да работят едновременно няколко транзакции, но върху различни кортежи. Ако блокировката е на по-едро ниво, то се губи ефективност. Ако блокировката е на по-ситно ниво, то се усложнява значително управлението на едновременния достъп.

Транзакциите са процеси – възможно е няколко транзакции да попаднат в дедлок. Механизмът, който се използва за преодоляване на този проблем е откриване и възстановяване от дедлок – всички транзакции, които участват в дедлока се елиминират.

Буферите се намират в оперативната памет и с тях взаимодейства модула управление на буферизацията. Ако една страница се прочита от външната памет, то тя първо се записва в буферите.

Изпълнителната машина взаимодейства с модула за управление на едновременния достъп – например, тя изпълнява командите на този модул.

Журналът се управлява самостоятелно и работи пряко с управлението на буферизацията. Всички останали модули се изпълняват на изпълнителната машина. Журналът, обаче, слиза на по-ниско ниво, тъй като той се нуждае от гаранция, че неговите изменения са направени върху диска преди да се обработва базата от данни.

Метаданните са описание на схемата на базата от данни и те се разполагат в буфери в оперативната памет. DDL-компилаторът се нуждае от метаданните – например, ако администраторът изменя базата от данни.
Журналните страници също се разполагат в буфери в оперативната памет. Компилаторът на заявките също има свои буфери – метаданни, които се използват при проверка на семантиката и статистики, които се натрупват от СУБД и се използват при оптимизиране на заявките.
В хода на своята работа изпълнителната машина също използва буфери – метаданни, данни, индекси.

Често при описание на свойствата на транзакциите се използва съкращението **ACID** – атомарност (atomicity), цялостност (consistency), изолирано изпълнение (isolation), надеждност и устойчивост (durability).

По-отношение на СУБД се изучават следните компоненти:
- проектиране на базите от данни – каква информация се съхранява в базата от данни;
- структура на базата от данни – какви структури от данни, типове от данни се използват и какви са връзките между типовете данни;
- програмиране на базата от данни – програмиране на заявки и на конвенционални програми;
- реализация на СУБД – този въпрос няма да го разглеждаме подробно.

Модел на данните същност-връзки (Entity-Relationship, ER)

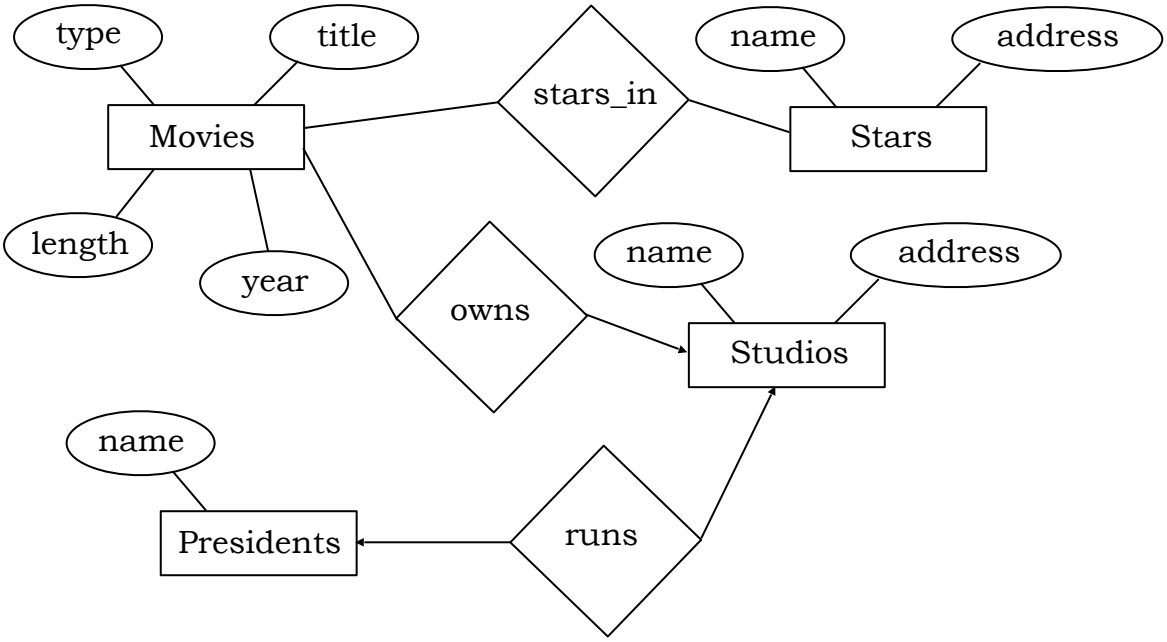
Този модел на данните се използва главно за проектиране на бази от данни. Процесът на проектиране на една база от данни започва от това каква информация да се съхранява в нея. В общия случай това е доста труден въпрос – нужни са познания в приложната област.
Процесът на проектиране можем да разделим на две части:
- определяне на информацията в базата от данни;
- определяне на връзките между компонентите на информацията.

Моделът ER дава възможност с определени структури да се опишат същности, а с други структури да се опишат връзките между тези същности.

Схемата на базата от данни се описва с помощта на език или друга подходяща нотация. След като бъде описана схемата на базата от данни, изпълнителната машина трябва по подходящ начин да превърне тази схема във физическа база от данни върху диска. В съвременнните СУБД обикновено описанието се прави ER, след което това описание се транслира към SQL и тогава се подава на DDL-компилатора.
Друг подход за описание на базата от данни е обектно-ориентираният подход. При този подход се използват така наречените **UML-диаграми**. Те са ориентирани по-скоро към описание на софтуера, отколкото описание на самите данни както е при ER-диаграмите.
Недостатък на релационния модел при проектиране е, че структурата която се използва е само една – таблицата (релацията). Чрез тази структура трябва да се описват както същностите, така и връзките. Затова при проектиране се предпочита ER-модела пред релационния модел.

При модела ER съществени елементи са множествата **същности**, множествата **атрибути** и множествата **връзки**.
Същността е нещо, което се възприема самостоятелно, т.е. което може да бъде отделено от другите същности. Същността може да е предмет, явление, но може да бъде и абстрактно понятие.
При проектирането на информацията е важно до каква степен тя е детайлизирана. Една същност се описва с конкретни стойности за нейните атрибути. Стойностите на атрибутите могат да бъдат атомарни, т.е. числа, дати, низове и др. Друг вариант е стойностите на атрибутите да са цели структури с фиксиран брой компоненти. В нашите разглеждания ще считаме, че всички стойности на атрибути са атомарни.
Между две или повече множества от същности може да има **връзки**. Най-простите връзки са бинарни, но може да има и по-сложни връзки между повече от две множества същности.

Схемата на базата от данни при модела ER се описва с **диаграма същност-връзки** (**диаграма ER**), която представлява граф. Множествата същности се представят с правоъгълници, атрибутите се представят с овали, връзките се представят с ромбове. Ще разгледаме един пример.



Stars и Studios са различни множества от същности, въпреки че имат едни и същи атрибути. Правило е да се използват колкото се може по-малко атрибути. Естествено, същностите и атрибутите трябва да са така подбрани, че да отразяват адекватно реалния свят.

Диаграмата ER е описание на схемата на базата от данни. С нея не се описва конкретното съдържание на базата от данни. Конкретните данни се наричат **екземпляр** на базата от данни и те могат да се менят с времето, докато схемата на базата от данни е устойчива и не се променя.

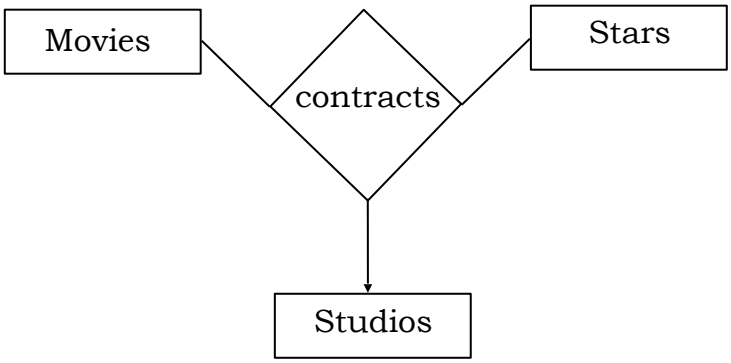
Това е причината по която DML-езика се отделя от DDL-езика.
В екземпляр на базата от данни първо се определят конкретни крайни множества от същности, а след това за всяка конкретна същност се определя конкретна стойност за всеки един от нейните атрибути.
Екземплярът на връзките се представя малко по-специфично.
Нека R е връзка между множествата от същности E_1, E_2, …, E_n.
Тогава екземпляр на R е краен списък от наредени n-торки (e_1, e_2, …, e_n), където e_i е конкретна същност от множеството E_i, i = 1, 2, …, n.
Между две множества от същности може да има повече от една връзка.
Много често за визуализация на представянето на една връзка се използва таблица. Ще разгледаме пример.
Нека Sharon Stone и Arnold Schwarzeneger са конкретни същности от Stars. Нека Basic Instinct и Total Recall са конкретни същности от Movies. Тогава един екземпляр на връзката stars_in може да е следния:

Movies	Stars
Basic Instinct	Sharon Stone
Total Recall	Arnold Schwarzeneger
Total Recall	Sharon Stone

Важна характеристика на една връзка е нейната **мултипликативност**.
Нека R е връзка между множествата същности E и F.
Ако всеки елемент на E може да бъде свързан с най-много един елемент на F, казваме че R е връзка **много към един** (many-one) от E към F.
В диаграмата ER за такива връзки се обозначава стрелка от R към F.
Пример от по-горе е връзката owns, което означава че един филм може да бъде притежаван най-много от едно студио.
Ако R е връзка много към един от E към F и R е връзка много към един от F към E, казваме че R е връзка **един към един** (one-one) между E и F. В диаграмата ER за такива връзки се обозначава стрелка от R към F и от R към E. Пример от по-горе е връзката runs, което означава, че едно студио може да има най-много един президент и освен това един президент може да ръководи най-много едно студио.
Ако R не е връзка много към един от E към F и R не е връзка много към един от F към E, казваме че R е връзка **много към много** (many-many) между E и F. В диаграмата ER за такива връзки не се обозначават стрелки. Пример от по-горе е връзката stars_in, което означава, че една звезда може да участва в повече от един филм и в един филм могат да участват повече от една звезди.

Връзки между повече от две множества същности се срещат рядко.
Нека E_0, E_1, …, E_n са множества от същности и R е връзка. Аналогично на по-горе, казваме че R е връзка **много към един** от E_1, …, E_n към E_0, ако за всеки елемент e_1 от E_1, e_2 от E_2, …, e_n от E_n съществува най-много един елемент e_0 от E_0, така че (e_0, e_1, …, e_n) е наредена (n+1)-орка от R.
В диаграмата ER за такива връзки се обозначава стрелка от R към E_0.

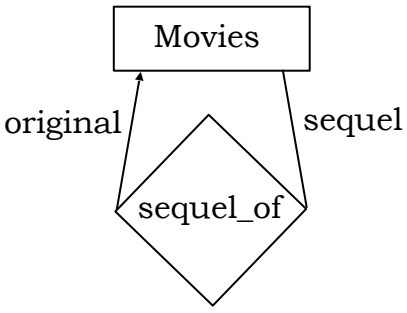
Ще разгледаме един пример.



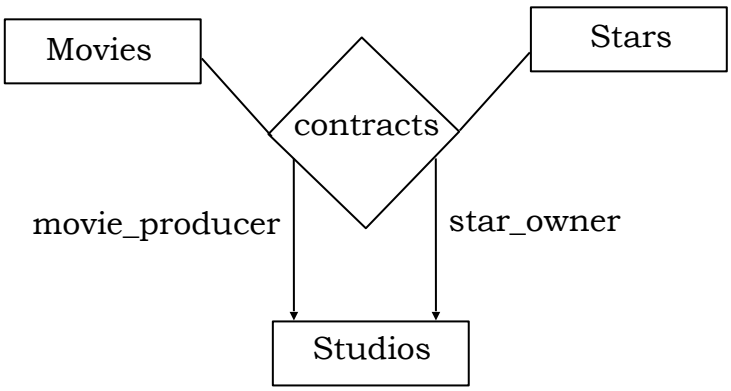
Връзката в случая е договор на звезда за участие във филм през определено студио. Стрелката означава, че една звезда за определен филм може да сключва договор най-много с едно студио.

Роли във връзката

Възможно е едно множество от същности да участва повече от един път в една връзка. Всяко участие на множеството същности във връзката се интерпретира по различен начин, така че различните участия трябва да се именуват. Именно тези участия се наричат **роли** във връзката.
Ще разгледаме два примера.



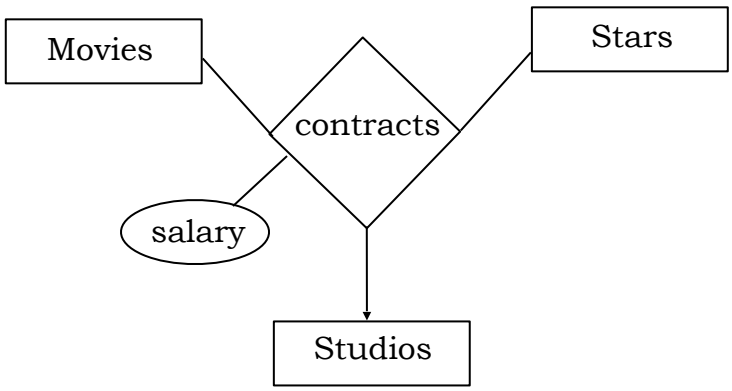
Бинарната връзка sequel_of отразява, че един филм е продължение на друг. Ролята original обозначава оригиналът на даден филм (той е единствен и затова връзката sequel_of в ролята original е много към един). Ролята sequel обозначава продълженията на даден филм.



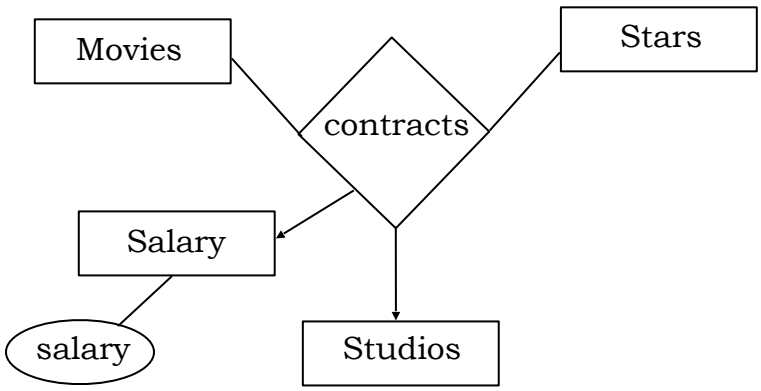
Във връзката contracts участват две роли на Studios. Ролята movie_producer обозначава студиото, което продуцира филма, а ролята star_owner обозначава студиото, което е сключило договор със звездата. Така всеки екземпляр на връзката се състои от наредени четворки от вида (studio1, studio2, movie, star), където star е конкретна звезда, movie е конкретен филм, studio1 е конкретно студио в ролята movie_producer, което е сключило договор със star, studio2 е конректно студио в ролята star_owner, което продуцира movie.

Атрибути на връзка

Ще въведем използването на атрибути за връзките, въпреки че има варианти на модела ER, където това не се допуска. Атрибути за една връзка се използват само когато не е възможно те да бъдат свързани с някое от множествата същности. В конкретен екземпляр на връзката с всяка наредена n-торка от списъка се свързва стойност за всеки един от атрибутите на връзката. Ще разгледаме пример.



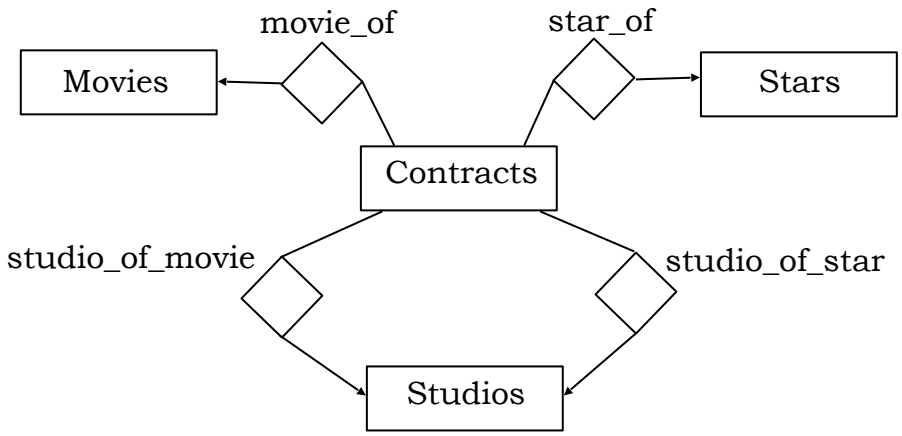
Атрибутът salary указва какво е заплащането на звездата от студиото по участието в даден филм. Практически е ясно, че този атрибут не може да се асоциира с никое от трите множества същности.
В моделите в които не се използват атрибути за връзките трябва да се добавят допълнителни множества същности, които характеризират самите връзки. Ще направим това за горния пример.



В диаграмата отразяваме факта, че дадена звезда по даден филм получава точно определена заплата.

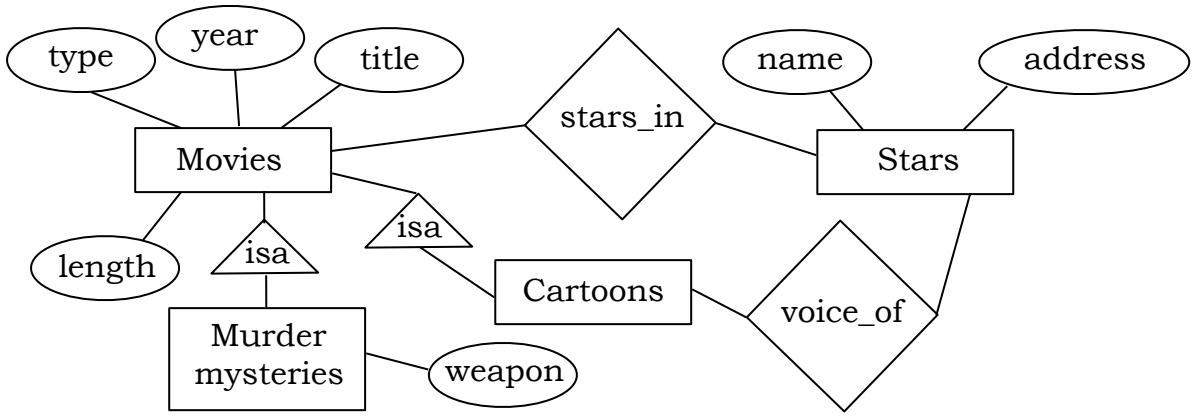
Представяне на небинарна връзка чрез бинарни връзки

В повечето релационни СУБД, където се използва ER-модела са позволени само бинарни връзки. Там в ER-диаграмата връзката се представя само чрез стрелка. При такива СУБД стремежът е към компактност на представянето – при визуализация ромбът заема излишно пространство.
Въпреки тези ограничения, ние можем да представяме небинарни връзки чрез бинарни. Принципът е следния: връзката заменяме с множество същности и след това добавяме бинарни връзки много към един от новото множество същности към множествата същности на връзката. При това, ако някое от множествата същности има няколко роли, то добавяме толкова бинарни връзки за това множество колкото са ролите. Вижда се, че при този подход не отразяваме ограниченията, които произлизат от типа на небинарната връзка.
Ще се справим с този проблем като въведем функционални и многозначни зависимости в релационния модел.
Като пример ще преобразуваме множествената връзка от по-горе, в която има роли.



ISA връзки

Характерно за съвременните СУБД е поддръжка на обектно-релационния модел на данните. За целта в модела ER са предвидени подкласове. Често пъти в едно множество може да има същности със специални свойства, при това тези свойства не се асоциират с всички същности от множеството. В такива случаи тези същности се дефинират в специални множества от същности, които се наричат **подкласове**.
В тях същностите могат да имат допълнителни атрибути и да участват в допълнителни връзки. Подкласовете от своя страна могат да имат свои подкласове и т.н. Между обикновените множества същности (класовете) и подкласовете се поддържа обикновено просто наследяване, т.е. всеки подклас има точно един родител. Това става с помощта на специални връзки, които се наричат **isa връзки**. Тези връзки са винаги един към един. Те се изобразяват с триъгълник, насочен към родителя във връзката. Ще разгледаме един пример.



Тук е мястото да подчертаем разликата между обектно-ориентирания подход и обектно-релационния модел. При обектно-ориентирания подход всеки обект принадлежи на точно определен клас, въпреки че може да бъде интерпретиран като обект от по-горен клас. При обектно-релационния модел една същност е едновременно същност от всички по-горни подкласове и множества.

Принципи на проектирането

Ще посочим някои основни принципи, които трябва да се спазват при проектиране на бази от данни с модела ER.

Съответствие

Проектът трябва да съответства на спецификацията на приложението. Тази спецификация се извлича от възложителя. Съответствието означава, че множествата същности, атрибутите и връзките трябва добре да отразяват реалността. Също така, връзките трябва да са адекватно подбрани по тип (много към много, един към един, много към един).

Простота и избягване на излишество

Излишните елементи, т.е. тези елементи, които нямат отношение към спецификацията трябва да бъдат избягвани. Трябва да се избягва както излишество на атрибути, така и излишество на връзки. Възможно е вместо връзка да се добавя допълнителен атрибут. Това не е винаги препоръчително, тъй като се увеличава вероятността за грешки.

Избиране на точните връзки

Добавянето на всяка една връзка трябва да бъде добре обмисляно – трябва да се внимава дали новата връзка не е следствие от

дефинираните преди това връзки. Определянето на точните връзки се извършва въз основа на проучвания в приложната област.

Избор на точния елемент

Понякога имаме избор да използваме различни елементи за да отразяваме реалността. Например, възможно е вместо комбинация от връзки и множества от същности да се използват атрибути.

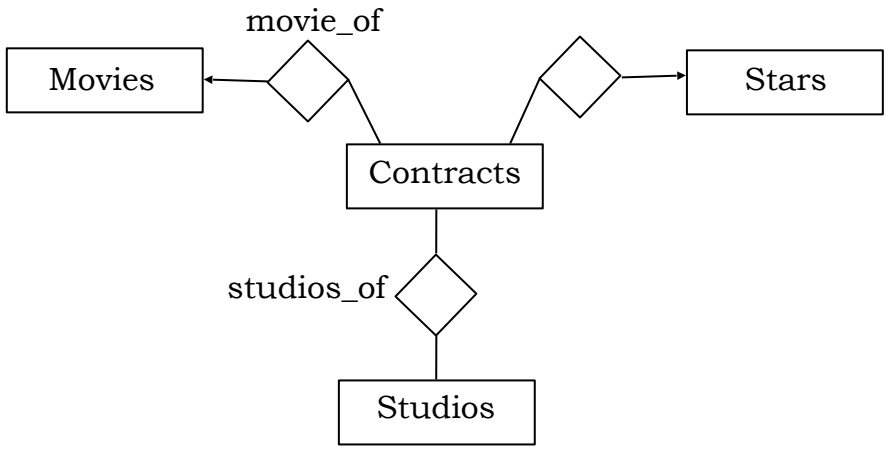
Нека E е множество от същности. Нека са изпълнени следните условия:

- всички връзки, в които участва E са много към един към E;
- всички атрибути на E заедно трябва да представят една същност, т.е. атрибутите на E (ако са повече от един) трябва да са независими един от друг – това е за да се избегне дублиране на информацията;
- в нито една връзка E не трябва да участва повече от веднъж, т.е. в никоя връзка E не трябва да има различни роли.

При тези предположения заместваме E по следния начин: нека R е връзка много към един от F към E. Тогава премахваме връзката R и във F добавяме атрибутите на E и атрибутите на R. Трябва да внимаваме да не се получи конфликт с имената на атрибутите, т.е. ако се наложи трябва да преименуваме добавените към F атрибути. Ако връзката R не е бинарна постъпваме по следния начин: добавяме атрибутите на E към атрибутите на R и не премахваме R. След като извършим процедурата върху всички връзки, в които участва E премахваме и самото множеството E.

Друг проблем при връзките е следния: възможно е в една връзка множество същности да участва с неопределен брой роли. В горните примери е възможно един филм да бъде продуциран от неопределен брой студия. В такъв случай можем да преобразуваме връзката в бинарна, както по-горе, но връзката към въпросното множество същности да е много към много. В примера това изглежда така:

star_of



Тук един договор може да бъде свързан с повече от едно студия.

Моделиране на ограничения

Една реална база от данни не може да бъде създадена само въз основа на множествата същности, техните атрибути и връзките. Трябва да се наложат някакви ограничения върху конкретните екземпляри на базата от данни. Тези ограничения могат да се класифицират по следния начин:

- **ключове** – атрибут или множество от атрибути, които уникално идентифицират съответните същности; две различни същности трябва да се различават по стойността на поне един ключов атрибут;
- **ограничения на единствената стойност** – в определен контекст да имаме уникалност на стойността; ключовете са такова ограничение, също връзките много към един;
- **ограничения за референтна цялостност** – в базата от данни да се извършват обръщения само към съществуващи обекти;
- **ограничения по домен** – стойността на даден атрибут да бъде от определено крайно множество;
- **общи ограничения** – валидни за цялата база от данни; те са най-тежки, тъй като трябва да се проверяват при всяка промяна на

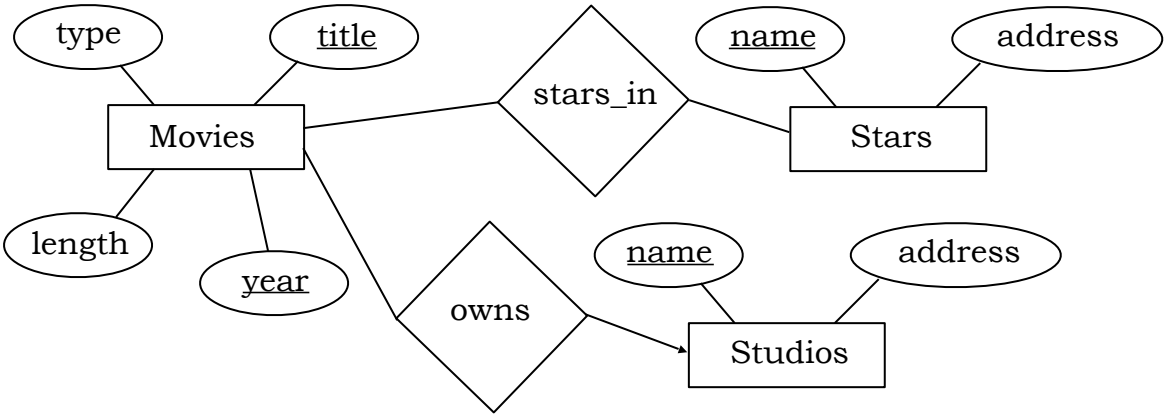
базата от данни; трябва да се стремим общите ограничения да са сведени до минимум по съображения за ефективност.

Ключове

Ключ за едно множество от същности E е подмножество K на атрибутите на E, такова че всеки две различни конкретни същности e_1, e_2 от E трябва да се различават по стойността на поне един от атрибутите в K.
Естествено, това не изключва възможността стойностите на някои атрибути от K за e_1 и e_2 да съвпадат. Последното е възможно в случай, че K се състои от повече от един атрибут.
Всяко множество от същности трябва да притежава ключ. Всъщност, това не е абсолютно задължително за модела ER – възможно е две конкретни същности да се различават по начина на свързване с конкретни същности от други множества.
Идентификацията на същностите се осъществява чрез ключа. Ако не е зададен ключ, то СУБД автоматично добавя поле с уникален идентификатор за всяка конкретна същност.

Възможно е за едно множество същности да има повече от един възможен ключ. В такъв случай се избира един от тях, който наричаме **първичен ключ**.
Когато едно множество от същности участва в isa-йерархия, то в ключа на това множество задължително трябва да участват само атрибути на коренното множество.

В ER диаграмата атрибутите, които участват в ключа на едно множество от същности се подчертават. Подчертава се единствено първичния ключ. Ще разгледаме пример.



Предполагаме, че може да има филми с еднакви заглавия, но само ако са произведени в различни години. Също, идентифицираме звездите само по имена – например, считаме че звездите с еднакви имена използват псевдоними.

Ограничения на единствената стойност

В проекта на една база от данни често пъти има най-много една стойност, която играе определена роля. Например, в горните примери един филм може да бъде притежаван най-много от едно студио.
Има няколко начина, по които ограниченията на единствената стойност се представят в диаграмата ER.

Всеки атрибут на дадена същност има единствена стойност. Понякога е възможно даден атрибут да няма стойност, но в този случай трябва да използваме **null-стойност** за този атрибут. Ако СУБД не поддържа null-стойности можем да използваме стойност, която не е от множеството стойности на съответния атрибут – например, ако за даден филм не се знае неговата продължителност да използваме стойност -1 за атрибута length. Релационните СУБД поддържат null-стойности, въпреки че те не се въвеждат във формалните разглеждания.
Недопустимо е стойността на ключов атрибут да е null, тъй като тя се използва за идентификация на същностите.

Връзка, която е много към един е друг източник на ограничение на единствената стойност.

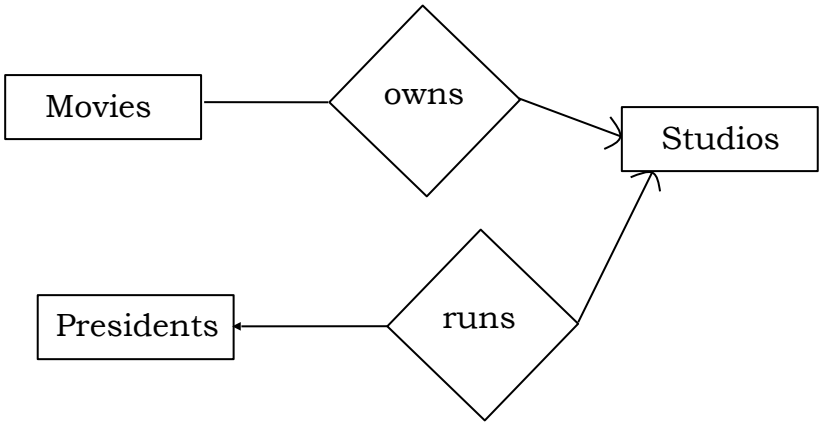
Ограничения за референтна цялостност

Ограниченията на единствената стойност означават, че трябва да има най-много една стойност, която играе определена роля. Ограниченията за референтна цялостност са по-силни – трябва да има точно една стойност. Например, даден атрибут задължително да трябва да притежава стойност. Най-вече ограниченията за референтна цялостност се използват при връзките между множества от същности. Например, ако наложим такова ограничение за връзката owns от по-горе, то изискваме всеки филм да бъде притежаван от студио.
Има няколко начина, по които можем да наложим ограничения за референтна цялостност:

- забрана за изтриване на дефинирана същност, ако тя е във връзка с някоя друга същност - например, ако дадено студио притежава филми да не можем да го изтрием;
- изтриването на същност да включва изтриването на всички същности, които са във връзка с нея - например, изтриването на дадено студио да включва изтриването на всички филми, които се притежават от него.

В конкретния пример, при вмъкване на нов филм той задължително трябва да се свърже със съществуващо студио. Също, ако за даден филм променим студиото, което го притежава, то новото студио задължително трябва да съществува.

В ER-диаграмата ограниченията за референтна цялостност се представят като се разшири нотацията на стрелките и се използват закръглени стрелки. Ако използваме закръглена стрелка вместо обикновена за обозначаване на връзка много към един от E към F, то изискваме всяка същност от множеството E да е свързана с точно една съществуваща същност от множеството F.
Ще разгледаме пример.



Стрелките се интерпретират по следния начин:

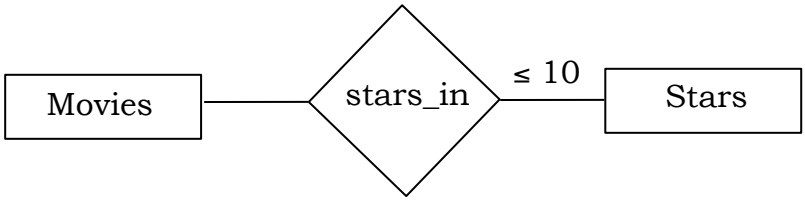
- стрелката от owns към Studios означава, че всеки филм трябва да се притежава от единствено съществуващо студио;
- стрелката от runs към Studios означава, че всеки президент трябва да управлява единствено съществуващи студио;
- стрелката от runs към Presidents означава, че всяко студио се управлява най-много от един президент, но може да има студия, които в определен момент нямат президенти.

Други ограничения

Ограниченията по домен означават, че даден атрибут може да приема стойности от определено крайно множество. Те могат да бъдат отразени чрез задаване на тип за атрибута. По-големи ограничения по домен можем да получим, ако директно изброим стойностите, които даден атрибут може да приема – например, дължината на филм (атрибута length) да приема стойност цяло число от 0 до 240. В модела ER няма специална нотация за ограничения по домен, въпреки че при конкретните СУБД такава нотация със сигурност има.

Има и други **общи ограничения**, които не попадат в разгледаните. Например, можем да наложим ограничение за броя на същностите, които могат да са свързани с конкретна същност по дадена връзка.

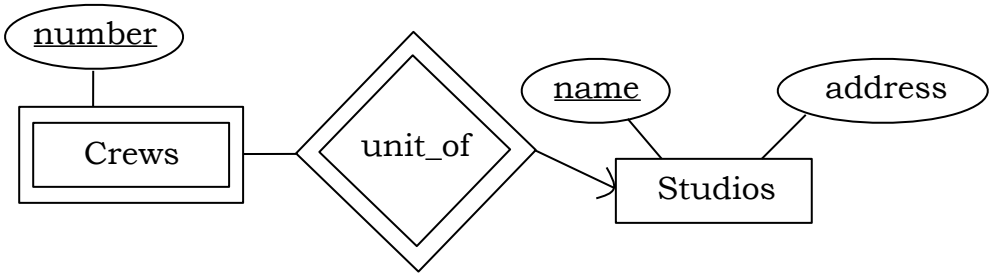
Частен случай на такова ограничение са връзките много към един (в частност един към един). В модела ER всяко ребро, което свързва връзка с множество същности можем да надпишем с ограничение за броя на същностите, които могат да са във връзка с конкретни същности от другите множества на връзката. Ще разгледаме пример.



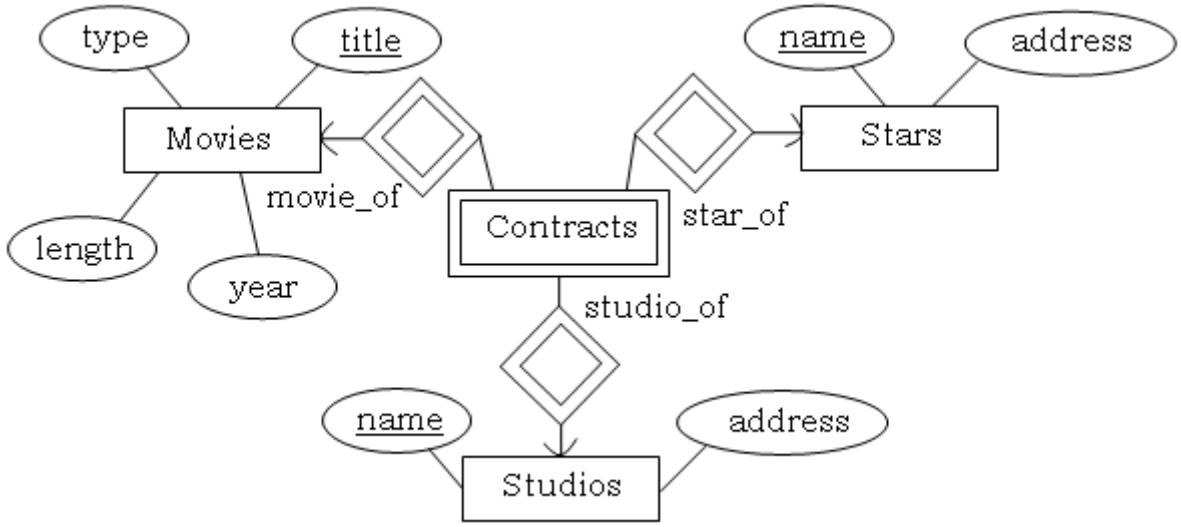
Ограничението означава, че във всеки филм може да участват най-много 10 звезди. Чрез такъв тип ограничения можем да представяме
стрелките – вместо обикновената стрелка можем да надписваме съответното ребро с ≤ 1, а вместо закръглена стрелка можем да надписваме съответното ребро с =1.

Слаби множества от същности

Слабите множества от същности са новост в модела ER и покриват част от проблемите при проектирането. За едно обикновено множество от същности ключът е винаги подмножество на неговите атрибути. За едно слабо множество от същности някои атрибути на неговия ключ може да са атрибути на друго множество същности.
Слаби множества от същности се използват когато имаме множества от същности, които са организирани в йерархия на класификация, която не е isa-йерархия. Ако същностите на едно множество E са подсъщности на друго множество F, то е възможно имената на същностите от E да не са уникални, докато не се вземат под внимание имената на същностите от F, на които съответните същности от E са подсъщности.
Ще разгледаме един пример.



В едно студио може да има няколко екипа. В рамките на едно студио екипите имат уникални номера, но е възможно различни студия да номерират екипите си по еднакъв начин. Поради тази причина, номерът не е достатъчен за да се идентифицира един екип. Така ключът на множеството Crews се образува от неговия атрибут number и атрибутът name на множеството Studios.
След малко ще обясним смисъла на двойните правоъгълници и ромбове.
Друг пример за слаби множества от същности са свързващите множества от същности, които се получават когато преобразуваме небинарни връзки към бинарни. Техният ключ се формира от ключовете на множествата същности, които участват във връзката, която е била преобразувана. Ще разгледаме пример.



Преобразували сме тернарната връзка между звезда, филм и студио в бинарни. Ключът на същностите от множеството Contracts се формира от ключовете на множествата Movies, Stars и Studios.

В диаграмите ER слабите множества от същности отбелязваме с двоен правоъгълник.
Нека E е слабо множество от същности.
Тогава неговият ключ се състои от:

- нула или повече от атрибутите на E;
- ключови атрибути на множества същности, които са свързани чрез специални **поддържащи връзки** много към един от E към тези множества същности.

Една връзка R е поддържаща за слабото множество E, ако:

- тя е бинарна и е много към един от E към някое множество F;
- съществува ограничение за референтна цялостност от E към F, т.е. всяка конкретна същност от E трябва да е свързана с точно една конкретна същност от F чрез връзката R (в диаграмата ER трябва да има закръглена стрелка от R към F).

В горните означения множеството F наричаме **поддържащо множество**. В ER диаграмите поддържащите връзки отбелязваме с двоен ромб.

Ако поддържащото множество от своя страна е слабо, то неговият ключ се определя рекурсивно по неговите поддържащи множества.
Ако към едно и също поддържащо множество се използват повече от една поддържащи връзки, то ключът на това множество участва в ключа на слабото множество толкова пъти, колкото са връзките. В този случай е нужно да се преименуват различните участия на ключа на поддържащото множество в ключа на слабото множество.

Релационен модел на данните

В релационния модел данните се представят по единствен начин: чрез двумерна таблица, която наричаме **релация**. Всяка релация има уникално име. Примерна релация с име Movies:

title	year	length	type
Star Wars	1977	124	color
Mighty Ducks	1991	104	color
Wayne's World	1997	95	color

Имената на колоните в таблицата се наричат **атрибути**. Всеки ред в таблицата се нарича **кортеж**.
Схема на една релация наричаме името на релацията, заедно с крайното множество от атрибутите на релацията. Схемата на релацията Relation, която има атрибути Attr1, Attr2, …, AttrN записваме по следния начин: Relation (Attr1, Attr2, …, AttrN). Например, схемата на горната релация е: Movies (title, year, length, type).

Атрибутите на една релация са множество, а не списък. Въпреки това се предполага, че при визуализация атрибутите се записват в определен стандартен ред. Този стандартен ред винаги ще определяме от записа на схемата на релацията.

Схема на една база от данни наричаме множеството от схемите на релациите, които участват в тази база от данни.

Всеки кортеж притежава една компонента за всеки атрибут на релацията. За да обозначаваме отделен кортеж на една релация обикновено записваме неговите компоненти, заградени в скоби и разделени със запетаи. При това, компонентите на кортежа записваме в съответствие със стандартния ред на атрибутите.
Например, за релацията Movie първият кортеж се обозначава по следния начин: (Star Wars, 1977, 204, color).

В релационния модел се изисква всяка компонента на всеки кортеж да има атомарна стойност – например цяло число, реално число, низ с фиксирана дължина. По никакъв начин не се позволява тази стойност да се разбива на по-малки компоненти.
С всеки атрибут на една релация се свързва **домен** – елементарен тип, който определя какви стойности може да приема компонентата на

кортежите, която съответства на атрибута. Домените на атрибутите са част от схемата на релацията, въпреки че засега няма да въвеждаме специални означения за тях.
Например, за релацията Movies атрибутът title има домен символен низ, атрибутите length и year имат домен цяло число, атрибутът type има домен множеството { color, blackAndwhite }.

Релациите са множества от кортежи, не списъци от кортежи. Поради тази причина, редът в който се записват кортежите на една релация е незначителен. Освен това, редът в който се записват атрибутите на една релация също е незначителен. Когато променяме този ред, обаче, трябва да променяме и редът в който се записват компонентите на съответните кортежи.
Така една релация може да бъде задавана по много еквивалентни начини.

Екземпляри на релация

Релациите не са статични, те се променят в течение на времето – добавят се нови кортежи, променят се съществуващи кортежи, изтриват се съществуващи кортежи.
По-рядко се променя схемата на една релация – добавяне или изтриване на атрибути. В една голяма релация това са твърде тежки операции – засягат се милиони кортежи. Друг проблем при добавяне на атрибут е, че не винаги е ясно какви стойности трябва да бъдат зададени за него във всички съществуващи кортежи.

Множеството на кортежите на една релация се нарича **екземпляр** на релацията. Конвенционалните СУБД поддържат само едно множество кортежи за дадена релация в даден момент и това множество наричаме **текущ екземпляр** на релацията.

Преобразуване на ER диаграми в релационни схеми

Както вече споменахме, моделът ER се използва при проектиране на една база от данни. Получената ER диаграма след това трябва да се преобразува в релационна схема. Възможно е проектирането изцяло да се извършва в релационен модел, както е при СУБД Oracle.

Преобразуването на ER диаграма в релационна схема се осъществява по следния начин:

- всяко множество същности се преобразува в релация, като атрибути на релацията са атрибутите на множеството същности;
- всяка връзка се преобразува в релация, като атрибути на релацията са атрибутите на връзката и ключовите атрибути на множествата същности, които участват във връзката.

Този начин не се прилага в следните ситуации, при които се изисква по-специално преобразуване:

- преобразуване на слаби множества от същности;
- преобразуване на isa-връзки;
- понякога две релации се комбинират в една – например, релация за множество същности E се комбинира с релация за връзка много към един от E към някое друго множество от същности.

Ще разгледаме примери. Засега не разглеждаме слаби множества същности. ER диаграмата за филмите, звездите и студията (в нейния първоначален вид) се преобразува към следните схеми на релации:
Movies (title, year, length, type)
Stars (name, address)
Studios (name, address)
Presidents (name)
Екземплярите на множествата същности се преобразуват по естествен начин към кортежи. Пример за екземпляр на релацията Stars:

name	address
Carrie Fisher	123 Maple Str., Hollywood
Mark Hamill	456 Oak Rd., Brentwood
Harrison Ford	789 Palm Dr., Beverly Hills

При преобразуване на връзка, ако едно множество същности има повече от една роля, то неговите ключови атрибути се включват толкова пъти в атрибутите на релацията за връзката, колкото са ролите му. При това, трябва да се прави подходящо преименуване за да няма дублиране на имената на атрибутите на релацията за връзката.

Връзките в ER диаграмата за филмите, звездите и студията се преобразуват към следните схеми на релации:
StarsIn (title, year, starName)
Owns (title, year, studioName)
Runs (studioName, presidentName)
Ключовите атрибути на множеството Movies са title, year, ключовият атрибут на множеството Stars е name, ключовият атрибут на множеството Studios е name, ключовият атрибут на множеството Presidents е name. Преименуването на атрибутите за връзката Runs е съществено, тъй като ключовите атрибути на множествата Studios и Presidents имат еднакви имена.
Като друг пример, връзката с ролите на Studios се преобразува към следната схема на релация:
Contracts (title, year, starName, producingStudio, studioOfStar).
Ключовите атрибути на Studios участват два пъти, тъй като множеството Studios има две роли във връзката contracts. Поради това извършваме преименуване. Ако връзката contracts има атрибут salary, то този атрибут salary също трябва да се включи към атрибутите на релацията.

При преобразуване на екземпляр на връзка R между множества същности E_1, E_2, …, E_n в екземпляр на релацията за връзката всяка наредена n-торка (e_1, e_2, …, e_n), e_i е конкретна същност от E_i, i = 1, 2, …, n се преобразува към кортеж, като за всяка конкретна същност e_i компонентите на кортежа, съответни на ключовите атрибути за множеството E_i са стойностите на тези ключови атрибути за същността e_i, i = 1, 2, …, n. Естествено, ако връзката има атрибути, то с наредената n-торка са свързани стойности за тези атрибути, така че компонентите на кортежа, които съответстват на атрибутите на връзката ще приемат тези стойности.

Комбиниране на релации

Комбинирането на релации може да се извършва в хода на преобразуването на ER-диаграма в релационни схеми.
Типична ситуация е следната: имаме две множества същности E, F и връзка R много към един от E към F. В този случай ключовите атрибути на E се съдържат както в релацията за E (заедно с останалите атрибути на E), така и в релацията за R. Релацията за R съдържа още ключовите атрибути на F, както и атрибутите на R, ако има такива. Тъй като R е връзка много към един, последните атрибути се определят еднозначно от ключовите на атрибути на E, което означава, че можем да комбинираме релациите на E и R в една релация със следните атрибути:

- всички атрибути на E;
- ключовите атрибути на F;
- атрибутите на R.

Един проблем е, че в кортеж, съответстващ на същност от E, която не е свързана със същност от F съответните компоненти за ключовите атрибути на F и за атрибутите на R трябва да имат null-стойности.

Като пример релацията за множеството същности Movies може да се комбинира с релацията за връзката Owns, тъй като връзката Owns е много към един от Movies към Studios. В резултат получаваме следната схема на релация: Movies (title, year, length, type, studioName).

Неудачно е релациите за всички връзки да се комбинират с релации за множества същности, особено, ако връзките са много към много. Например, да опитаме да комбинираме релацията Movies и релацията за връзката StarsIn.

Получаваме следната схема:
Movies (title, year, length, type, starName). Тъй като в един филм може да участва повече от една звезда, то за всяко такова участие трябва да има по един кортеж в екземпляр за Movies. Това, обаче, води до дублиране на останалата информация за филмите.

Преобразуване на слаби множества същности

Нека W е слабо множество от същности.
Преобразуването се извършва по следния начин:

- релацията за слабото множество W включва атрибутите на W и всички останали атрибути, които формират ключа на W; тези атрибути се определят от поддържащите връзки на W;
- релацията за една връзка, в която участва W, трябва да съдържа всички ключови атрибути на W, включително тези, които не са собствени атрибути на W; това важи само за връзки, които не са поддържащи;
- поддържащите връзки въобще не се преобразуват в релации – ако имат атрибути, те се добавят към атрибутите на релацията за W.

Като пример да преобразуваме слабото множество Crews, което разгледахме по-горе. Получаваме следните две схеми на релации:
Studios (name, address)
Crews (crewNumber, studioName).
Да предположим, все пак, че сме добавили и схема за връзката unit_of:
UnitOf (crewNumber, studioName, studioName1). Тук сме включили, както обикновено, ключовете на множествата същности Studios и Crews.
При това положение, кортежите във всеки екземпляр на релацията ще имат еднакви стойности за studioName и studioName1. Поради тази причина можем да премахнем атрибутът studioName1 от схемата.
Новополучената схема е абсолютно идентична на Crews, затова можем да я премахнем.

От примера по-горе може да останем с погрешно впечатление, че ако атрибутите на една релация са подмножество на атрибутите на друга релация, то може да премахнем първата релация.
Това не може да се осъществи, ако между релациите няма семантична връзка - например, релациите Studios и Stars имат едни и същи атрибути, но представляват съвсем различни същности.
Има случаи, в които между релациите има връзка, но елиминирането отново не може да се извърши.
Да разгледаме следния пример с две релации:
People (name, #ss)
TaxPayers (name, #ss, money)
Кортежите на първата релация отговарят на всички хора, които имат социални осигуровки. Кортежите на втората релация отговарят на хората, които имат социални осигуровки и са платили определена сума.
Ако премахнем релацията People, ще загубим информацията за хората, които са социално осигурени, но не са плащали данъци.

Преобразуване на isa-йерархии

При преобразуване на isa-йерархии се използват три подхода.

ER подход

При този подход всяко множество същности в йерархията се преобразува към отделна релация, която включва собствените атрибути

на множеството същности и ключовите атрибути на коренното множество същности. Връзките isa не се преобразуват към релации.

Като пример isa-йерархията от по-горе с корен Movies се преобразува към следните схеми на релации:
Movies (title, year, length, type)
Cartoons (title, year)
MurderMysteries (title, year, weapon)
За всеки филм има кортеж в екземпляра на Movies. Ако един филм е конкретна същност от Cartoons, то за него ще има кортеж както в екземпляра на Movies, така и в екземпляра на Cartoons. Аналогично, ако един филм е конкретна същност от MurderMysteries, то за него ще има кортеж както в Movies, така и в MurderMysteries.
Ако един филм е едновременно конкретна същност както в Cartoons, така и в MurderMysteries, то за него ще има кортежи и в трите релации.
Връзката voice_of се преобразува към следната релация:
VoiceOf (starName, title, year).
Тук starName е ключът на множеството същности Star, title, year е ключът на Cartoons, който съвпада с ключа на Movies.
Да отбележим, че схемата на релацията Cartoons е подсхема на релацията Voices. Въпреки това, можем да елиминираме релацията Cartoons само ако сме сигурни, че всички анимационни филми са озвучени.

Обектно-ориентиран подход

За всяко поддърво на йерархията, което съдържа корена се образува по една релация, като в нея се включват атрибутите на всички множества същности от поддървото.

В примера с йерархията с корен Movies имаме четири поддървета:
- поддървото, съставено само от Movies;
- поддървото, съставено от Movies и Cartoons;
- поддървото, съставено от Movies и MurderMysteries;
- поддървото, съставено от Movies, Cartoons и MurderMysteries.

Така трябва да съставим четири схеми на релации:
Movies (title, year, length, type)
MoviesC (title, year, length, type)
MoviesMM (title, year, length, type, weapon)
MoviesCMM (title, year, length, type, weapon)

За всяка конкретна същност от Movies има точно един кортеж в точно една от дефинираните релации. Затова подходът е обектно-ориентиран – всяка същност принадлежи на собствения си клас.
Връзката voice_of отново трябва да преобразуваме към следната схема на релация: VoiceOf (title, year, starName).
Ако има значение в коя релация е кортежът за даден филм от Cartoon, то за връзката voice_of трябва да съставим по една релация за всяко съответно поддърво. Ако поставим такова изискване връзката Voices ще се преобразува към следните две релации:

VoicesC (title, year, starName)
VoicesCMM (title, year, starName)

Използване на null-стойности

Създава се една обща релация, която включва всички атрибути на всички множества същности от йерархията. За всяка конкретна същност има точно един кортеж в релацията. Ако тази същност не принадлежи на някое множество същности от йерархията, то компонентите в кортежа, съответни на собствените атрибути на това множество същности трябва да съдържат null-стойности.

В примера с йерархията Movies трябва да образуваме една релация:
Movies (title, year, length, type, weapon).
За филмите, в които няма мистериозни убийства стойността на атрибута weapon ще е null. Връзката voice_of се преобразува аналогично към релация VoiceOf (title, year, starName). И в този случай могат да се дефинират няколко релации за връзката voice_of, ако има значение в кои множества същности от йерархията попада съответния анимационен филм.

Сега ще сравним трите подхода по някои критерии.

1. Скъпо е да се отговаря на заявки, в които участват няколко релации. По този критерий най-добрият подход е с null-стойностите. Другите два подхода имат предимства и недостатъци в зависимост от заявките. Например:
    a. Нека заявката е “кои филми от 1999г. са по-дълги от 150 минути?”. Тогава при ER подхода търсенето ще се осъществи само в релацията Movies, докато при обектно-ориентирания подход трябва да се търси във всяка една от релациите.
    b. Нека заявката е “какво оръжие се използва в анимационните филми, които са по-дълги от 150 минути?”. Тогава при ER подхода търсенето ще започне в релацията Movies, където ще се определят филмите, които са по-дълги от 150 минути. След това ще продължи в релацията Cartoons за да се определят кои от намерените са анимационни филми, след това в релацията MurderMysteries за да се определи в кои от тях има мистериозни убийства. Така търсенето засяга и трите релации. При обектно-ориентирания подход трябва да се търси само в една релация – MoviesCMM, където е налице всичката нужна информация.
2. Добре е да не използваме много релации. Тук отново подходът с null-стойности е най-добър. Обектно-ориентираният подход е най-лош, тъй като броят на поддърветата расте експоненциално с нарастване на броя на множествата същности в йерархията.
3. Минимизиране на пространството и избягване на повторенията. Тук най-добър е обектно-ориентираният подход, тъй като за всяка същност има точно един кортеж, който съдържа точно тази информация, която е смислена за кортежа. При ER подхода за една същност може да има няколко кортежа, но в тях се дублират

само ключовете. Подходът с null-стойности е най-лош по този критерий за големи йерархии, въпреки че отново за всяка същност има единствен кортеж – при него има огромно разхищение на пространство.

При конкретни СУБД, които поддържат ER модела има опции коя от трите стратегии да се използва.

Функционални зависимости

Ще разгледаме по-подробно ограничения на единствената стойност в релационния модел, които ще наречем функционални зависимости. Освен тях се използват многозначни зависимости и ограничения за референтна цялостност. Всички тези ограничения се използват за усъвършенстване на схемата на една база от данни.

Функционална зависимост в една релация R наричаме твърдение от вида от следния вид: ако два кортежа имат едни и същи компоненти, отговарящи на атрибутите A_1, A_2, …, A_n, то те имат едни и същи компоненти, отговарящи на атрибута B. Казваме още, че атрибутите A_1, A_2, …, A_n функционално определят атрибута B.
Бележим функционалната зависимост по следния начин: $A_1\ A_2 \ldots A_n \rightarrow B$.
Ако A_1, A_2, …, A_n функционално определят повече от един атрибут, т.е. имаме $A_1\ A_2 \ldots A_n \rightarrow B_1$, $A_1\ A_2 \ldots A_n \rightarrow B_2$, …, $A_1\ A_2 \ldots A_n \rightarrow B_k$, то съкратено записваме $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_k$.

Като пример в релацията
Movies (title, year, length, type, studioName, starName)
имаме следните функционални зависимости:
title year → length, title year → type, title year → studioName или съкратено
title year → length type studioName.
Първите две зависимости произтичат от това, че в диаграмата ER
title, year е ключ на множеството същности Movies, а третата зависимост произтича от това, че има връзка много към един от Movies към Studios.
От друга страна, за релацията Movies не е в сила функционалната зависимост title year → starName, тъй като в един филм може да участва повече от една звезда.

Функционалните зависимости са ограничения за единствената стойност, които се отнасят към схемата на една релация, а не към определен неин екземпляр. По даден екземпляр можем да съдим за отсъствие, но не и за присъствие на функционална зависимост в схемата на релацията.

Ще определим понятието ключ в релационния модел.
Казваме, че множеството $\{A_1, A_2, \ldots, A_n\}$ от един или повече атрибути на релацията R образуват **ключ** на R, ако A_1, A_2, …, A_n функционално определят всички атрибути на R и $\{A_1, A_2, \ldots, A_n\}$ е минимално по включване с това свойство. Ако ключ се състои от единствен елемент, не поставяме фигурни скоби.

Например, в релацията Movies по-горе { title, year, starName } е ключ.

Възможно е една релация да има повече от един ключ. Обикновено в такава ситуация се избира един ключ, който се обявява за **първичен ключ**. Първичният ключ се използва при съхранение на релацията. Естествено, добре е първичният ключ да съдържа колкото се може по-малко атрибути.

В означението за схемата на релацията ще подчертаваме атрибутите, които образуват първичния ключ на тази релация. Например:
Movies (<u>title</u>, <u>year</u>, length, type, studioName, <u>starName</u>).

В модела ER няма изискване за минималност на ключовете.
За минималност на ключа можем да съдим само ако разполагаме с пълния набор от функционални зависимости между атрибутите.

Суперключ на една релация R се нарича множество от атрибути, което съдържа ключ. С други думи, атрибутите на суперключа трябва да определят функционално всички атрибути на R, но не е задължително суперключът да е минимален. Естествено, всеки ключ е суперключ, но обратното не е вярно.

Откриване на ключове в релации

Ако една релация се получава от множество същности на ER диаграма, то ключът на тази релация съвпада с ключа на множеството същности. Например:
Movies (<u>title</u>, <u>year</u>, length, type)
Stars (<u>name</u>, address)
Ако една релация се получава от бинарна връзка, то нейният ключ се определя в зависимост от типа на връзката:

- ако връзката е много към много, то ключът на релацията за връзката се състои от ключовете на двете множества същности;
- ако връзката е много към един от E към F, то ключът на релацията за връзката се състои от ключовите атрибути на E;
- ако връзката е един към един между E и F, то ключът на релацията за връзката се състои или от ключовите атрибути на E или от ключовите атрибути на F, т.е. релацията няма единствен ключ.

Примери:
Owns (<u>title</u>, <u>year</u>, studioName)
StarsIn (<u>title</u>, <u>year</u>, <u>starName</u>)

Ако една релация се получава от небинарна връзка, то от диаграмата ER не може да се определят всички функционални зависимости, които съществуват между нейните атрибути. Със сигурност е изпълнено следното: ако връзката е много към един към някое множество същности E, т.е. има стрелка към E, то със сигурност съществува ключ на релацията за връзката, който не съдържа ключовите атрибути на E.

Правила за функционалните зависимости

Ще разгледаме някои правила, чрез които от даден набор функционални зависимости в релация можем да извличаме нови функционални зависимости в тази релация.

Функционалните зависимости за една релация често могат да бъдат представени по различен начин. По-формално, казваме че множествата S и T от функционални зависимости са **еквивалентни**, ако множеството от екземплярите на релацията, удоволетворяващи S съвпада с множеството от екземплярите на релацията, удоволетворяващи T.
По-общо, множеството функционални зависимости S **следва от** множеството от функционални зависимости T, ако всеки екземпляр на релацията, който удоволетворява T, удоволетворява и S.
Естествено, ако от S следва T и от T следва S тогава и само тогава, когато S и T са еквивалентни.

Нека е дадена функционална зависимост $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$.
Тогава можем да заменим тази функционална зависимост със следните функционални зависимости: $A_1\ A_2\ \ldots A_n \rightarrow B_i$, i = 1, 2, …, m. Това правило наричаме **правило за разделяне**.
Обратно, функционалните зависимости $A_1\ A_2\ \ldots A_n \rightarrow B_i$, i = 1, 2, …, m можем да заменяме с една функционална зависимост $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$. Това правило наричаме **правило за комбиниране**.
Очевидно е, че и в двата случая полученото множество функционални зависимости е еквивалентно на изходното.
Например, множествата функционални зависимости { title year → length, title year → type, title year → studioName } и
{ title year → length type studioName } са еквивалентни.

Казваме, че функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B$ е **тривиална**, ако $B = A_i$ за някое $i \in \{ 1, 2, \ldots, n \}$. Естествено, тривиалните функционални зависимости са изпълнени във всяка релация.

По-общо, казваме че функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$ е **тривиална**, ако $\{ B_1, B_2, \ldots, B_m\} \subseteq \{ A_1, A_2, \ldots, A_n\}$.
Казваме, че функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$ е **нетривиална**, ако съществува $i \in \{ 1, 2, \ldots, m\}$, такова че $B_i \notin \{ A_1, A_2, \ldots, A_n\}$.
Казваме, че функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$ е **напълно нетривиална**, ако за всяко $i \in \{ 1, 2, \ldots, m\}$, имаме $B_i \notin \{ A_1, A_2, \ldots, A_n\}$.

Например, функционалната зависимост title year → title е тривиална, функционалната зависимост title year → year length е нетривиална, а функционалната зависимост title year → length е напълно нетривиална.

Нека $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$ е нетривиална функционална зависимост.

Тогава можем да премахнем онези B_i, които съвпадат с някое A_j. Получаваме нова функционална зависимост $A_1\ A_2\ \ldots A_n \rightarrow C_1\ C_2\ \ldots C_k$, която очевидно е еквивалентна на изходната. Това правило наричаме **правило за отстраняване на тривиалните зависимости**.

Нека $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$ и $B_1\ B_2\ \ldots B_m \rightarrow C_1\ C_2\ \ldots C_k$ са функционални зависимости. Тогава добавяме нова функционална зависимост $A_1\ A_2\ \ldots A_n \rightarrow C_1\ C_2\ \ldots C_k$. Това правило наричаме **транзитивно правило**. Ясно е, че добавената функционална зависимост следва от изходните функционални зависимости, така че получаваме еквивалентно множество от функционални зависимости.

Например, да разгледаме релацията
Movies (<u>title</u>, <u>year</u>, length, type, studioName, studioAddress).
За нея са в сила функционалните зависимости
title year → studioName, studioName → studioAddress.
По транзитивното правило получаваме, че функционалната зависимост title year → studioAddress също е в сила за тази релация.

Изчисляване на покритие на атрибути

Нека $\{ A_1, A_2, \ldots, A_n\}$ е множество от атрибути, S е множество от функционални зависимости. **Покритие** на $\{ A_1, A_2, \ldots, A_n\}$ относно S е множеството от всички атрибути B, такива че всеки екземпляр на релацията, който удоволетворява S удоволетворява и функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B$, т.е. от S следва $A_1\ A_2\ \ldots A_n \rightarrow B$.
Означаваме покритието по следния начин: $\{ A_1, A_2, \ldots, A_n\}^+$.
Естествено, $\{ A_1, A_2, \ldots, A_n \} \subseteq \{ A_1, A_2, \ldots, A_n\}^+$, тъй като тривиалните функционални зависимости винаги са изпълнени.

Ще опишем алгоритъм за изчисляване на покритието на $\{ A_1, A_2, \ldots, A_n\}$ относно S.

1. Инициализираме $X = \{ A_1, A_2, \ldots, A_n \}$.
2. Търсим функционална зависимост $B_1\ B_2\ \ldots B_m \rightarrow C$ в S, такава че $B_1, B_2, \ldots, B_m \in X$, но $C \notin X$. Тогава добавяме C към X.
3. Повтаряме стъпка 2., докато вече не е възможно добавяне на атрибути в X. Този момент винаги ще настъпи, тъй като X винаги расте, а множеството от атрибути е крайно.
4. Покритието $\{ A_1, A_2, \ldots, A_n\}^+$ съвпада с X.

Ще разгледаме пример. Нека атрибутите на релацията са A, B, C, D, E, F. Нека S се състои от следните функционални зависимости
AB → C, BC → AD, D → E, CF → B.
Да изчислим покритието на $\{ A, B\}$ в S.

1. $X = \{ A, B \}$.
2. $X = \{ A, B, C\}$, чрез AB → C.
3. $X = \{ A, B, C, D\}$, чрез BC → AD.
4. $X = \{ A, B, C, D, E \}$, чрез D → E.

Невъзможно е X да нараства още, така че $\{ A, B\}^+ = \{ A, B, C, D, E \}$.

Покритието се използва при определяне дали дадена функционална зависимост следва от множество функционални зависимости по следната схема. Нека $A_1\ A_2\ \ldots A_n \rightarrow B$ е функционална зависимост, S е множество от функционални зависимости. Тогава образуваме покритието на $\{ A_1, A_2, \ldots, A_n \}$ относно S.

1. Ако $B \in \{ A_1, A_2, \ldots, A_n\}^+$, то функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B$ следва от S.
2. Ако $B \notin \{ A_1, A_2, \ldots, A_n\}^+$, то функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B$ не следва от S.

По-общо, функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$ следва от S тогава и само тогава, когато $\{ B_1, B_2, \ldots, B_m\} \subseteq \{ A_1, A_2, \ldots, A_n \}^+$.

В примера, който разгледахме преди малко нека да определим дали функционалната зависимост $D \rightarrow A$ следва от S.
Първо изчисляваме $\{ D\}^+$.

1. $X = \{ D \}$.
2. $X = \{ D, E \}$, чрез $D \rightarrow E$.

Невъзможно е X да нараства още, така че $\{ D\}^+ = \{ D, E \}$. Тъй като $A \notin \{ D, E \}$, то функционалната зависимост $D \rightarrow A$ не следва от S.

Ще се заемем със задачата да покажем, че описаният алгоритъм е коректен. Трябва да покажем две неща:

1. (коректност) Ако $B \in \{ A_1, A_2, \ldots, A_n\}^+$, то функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B$ действително следва от S.
2. (пълнота) Ако $A_1\ A_2\ \ldots A_n \rightarrow B$ следва от S, то задължително $B \in \{ A_1, A_2, \ldots, A_n\}^+$.

С индукция по броя на изпълненията на стъпка 2. от алгоритъма ще покажем, че за всеки атрибут D в X, имаме, че функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow D$ следва от S.
База: преди да се изпълни стъпка 2. имаме $X = \{ A_1, A_2, \ldots, A_n\}$, така че функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow D$ е тривиална и следва от S.
Предположение: нека твърдението е изпълнено след k-тото изпълнение на стъпка 2.
Стъпка: нека D е добавено в X при k+1-то изпълнение на стъпка 2. Тогава съществува функционална зависимост $B_1\ B_2\ \ldots B_m \rightarrow D$ от S и $B_1\ B_2\ \ldots B_m$ са били в X след k-тото изпълнение на стъпка 2.
По индукционното предположение функционалните зависимости $A_1\ A_2\ \ldots A_n \rightarrow B_i$, $i = 1, 2, \ldots, m$ следват от S, също $B_1\ B_2\ \ldots B_m \rightarrow D$ е в S, така че $A_1\ A_2\ \ldots A_n \rightarrow D$ следва от S по транзитивното правило.

Тъй като след изпълнението на алгоритъма $X = \{ A_1, A_2, \ldots, A_n\}^+$, то коректността е доказана.

Нека $B \notin \{ A_1, A_2, \ldots, A_n\}^+$. Ще покажем, че $A_1\ A_2\ \ldots A_n \rightarrow B$ не следва от S.
За целта разглеждаме следният екземпляр на релацията:

	$\{ A_1, A_2, \ldots, A_n\}^+$	останалите атрибути (тук е B)

t:	1	1	…	1	0	0	…	0
s:	1	1	…	1	1	1	…	1

Нека $C_1\ C_2\ \ldots C_k \rightarrow D$ е функционална зависимост от S. Да допуснем, че тя не е в сила за горния екземпляр. Тъй като в него има само два кортежа, то те я нарушават, т.е. стойностите им за $C_1\ C_2\ \ldots C_k$ съвпадат, но стойностите за D не съвпадат. Ясно е, че при това положение $C_1, C_2, \ldots, C_k \in \{ A_1, A_2, \ldots, A_n\}^+$, $D \notin \{ A_1, A_2, \ldots, A_n\}^+$. Но това е противоречие, тъй като D трябва да е било добавено към X чрез функционалната зависимост $C_1\ C_2\ \ldots C_k \rightarrow D$ когато X е било $\{ A_1, A_2, \ldots, A_n\}^+$. И така, функционалните зависимости от S са в сила за посочения екземпляр. От друга страна, очевидно функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B$ не е в сила за него, тъй като $A_i \in \{ A_1, A_2, \ldots, A_n\}^+$, i = 1, 2, …, n, но $B \notin \{ A_1, A_2, \ldots, A_n\}^+$.
Така намерихме екземпляр на релацията, който удоволетворява S, но не удоволетворява $A_1\ A_2\ \ldots A_n \rightarrow B$, така че $A_1\ A_2\ \ldots A_n \rightarrow B$ не следва от S.
С това доказахме пълнотата.

Да отбележим, че $\{ A_1, A_2, \ldots, A_n \}^+$ се състои от всички атрибути на релацията $\Leftrightarrow \{ A_1, A_2, \ldots, A_n \}$ е суперключ на тази релация. Така, ако трябва да проверим дали дадено множество $\{ A_1, A_2, \ldots, A_n \}$ е ключ на дадена релация, то първо проверяваме дали $\{ A_1, A_2, \ldots, A_n \}^+$ изчерпва всички атрибути и след това се уверяваме, че никое собствено подмножество на $\{ A_1, A_2, \ldots, A_n \}$ няма това свойство.

Множество от функционални зависимости за една релация, от което могат да се извлекат всички останали функционални зависимости в тази релация се нарича **база** на релацията. База, която е минимална по включване с това свойство наричаме **минимална база**. Минималните бази могат да се използват за представяне на всички функционални зависимости в дадена релация.
Като пример да разгледаме релацията R (A, B, C) със следните функционални зависимости $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow A$, $B \rightarrow C$, $C \rightarrow A$, $C \rightarrow B$, $AC \rightarrow B$, $AB \rightarrow C$, $BC \rightarrow A$. Тогава множествата
$\{ A \rightarrow B, B \rightarrow A, B \rightarrow C, C \rightarrow A \}$ и $\{ A \rightarrow B, B \rightarrow C, C \rightarrow A \}$ образуват минимални бази на релацията.

Проектиране на функционални зависимости

Нека R е релация, която удоволетворява множеството от функционалните зависимости F. Да предположим, че S е нова релация, която се получава от R с премахване на атрибути. Казваме, че S е **проекция** на R. Тогава функционалните зависимости, които са в сила за S са точно онези, които следват от F и в които участват само атрибути на S.

Като пример да разгледаме релация R (A, B, C, D) с множеството от функционални зависимости $F = \{ A \rightarrow B, B \rightarrow C, C \rightarrow D \}$.

Нека новата релация е S (A, C, D). За да намерим функционалните зависимости за S изчисляваме покритията на всички подмножества на { A, C, D } относно F. Естествено, можем да пропуснем ∅ и { A, C, D}, тъй като от тях не може да се получи нетривиална функционална зависимост. Имаме $\{ A\}^+ = \{ A, B, C, D \}$, $\{ C\}^+ = \{ C, D \}$, $\{ D\}^+ = \{ D\}$. Така получаваме функционални зависимости A → C, A → D, C → D. Няма смисъл да разглеждаме надмножества на { A}, тъй като $\{ A\}^+$ съдържа всички атрибути на S. Така остава само $\{ C, D\}^+ = \{ C, D\}$, от което не получаваме нова функционална зависимост.
Окончателно, в S се проектират следните функционални зависимости: A → C, A → D, C → D.

Аксиоми на Армстронг

Както вече видяхме, чрез алгоритъма за намиране на покритие винаги можем да определим дали дадена функционална зависимост следва от дадено множество от функционални зависимости.
Аксиомите на Армстронг са правила, чрез които може да се извлече всяка функционална зависимост, която следва от дадено множество функционални зависимости. Те са следните:

1. **Рефлексивност** - ако $\{ B_1, B_2, \ldots, B_m\} \subseteq \{ A_1, A_2, \ldots, A_n\}$, то $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$.
2. **Нарастване** – ако $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$ и $C_1, C_2, \ldots, C_k$ са произволни атрибути, то $A_1\ A_2 \ldots A_n\ C_1\ C_2 \ldots C_k \rightarrow B_1\ B_2 \ldots B_m\ C_1\ C_2 \ldots C_k$.
3. **Транзитивност** – ако $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$ и $B_1\ B_2 \ldots B_m \rightarrow C_1\ C_2 \ldots C_k$, то $A_1\ A_2 \ldots A_n \rightarrow C_1\ C_2 \ldots C_k$.

Проектиране на схеми на релационните бази от данни

При проектиране на схема на една релация трябва да се избягват следните аномалии:

1. Излишество – информацията безсмислено да се дублира в кортежите.
2. Аномалии на обновяването – при обновяване на данните да не са актуализирани всички засегнати кортежи.
3. Аномалии на изтриването – при изтриване на данни да се губи друга информация като страничен ефект.

Като пример да разгледаме следният екземпляр на релацията Movies, към която е присъединена релацията за връзката StarsIn.

title	year	length	type	studioName	starName
Star Wars	1977	124	color	Fox	Carrie Fisher
Star Wars	1977	124	color	Fox	Mark Hamill
Star Wars	1977	124	color	Fox	Harrison Ford
Mighty Duck	1991	104	color	Disney	Emilio Estevez

Wayne's World	1992	95	color	Paramount	Dana Carvey
Wayne's World	1992	95	color	Paramount	Mike Meyers

Тази релация съдържа и трите аномалии:

1. Информацията type, length излишно се дублира.
2. Ако обновяваме length на Star Wars, например, трябва да засегнем всичките три кортежа – това увеличава вероятността за грешка.
3. Ако трябва да изтрием Emilio Estevez от звездите, които участват във филма Mighty Duck, то трябва да премахнем целият кортеж, което ще доведе до загуба на останалата информация за филма.

Декомпозиция на релациите

Общоприетият начин за елиминиране на изброените аномалии е декомпозицията на релациите – една релация се разбива на две нови релации. По-формално, нека R (A_1, A_2, …, A_n) е релация.
Тогава тя се **декомпозира** на две нови релации
S (B_1, B_2, …, B_m) и T (C_1, C_2, …, C_k), ако:

1. { B_1, B_2, …, B_m} ∪ { C_1, C_2, …, C_k} = { A_1, A_2, …, A_n}.
2. Кортежите в S са **проекции** на кортежите на R по B_1, B_2, …, B_m. Това означава, че от всеки кортеж на екземпляр на R избираме само компонентите, които съответстват на B_1, B_2, …, B_m и по този начин получаваме кортежите на S. Естествено, от два различни кортежа на R могат да се получат две еднакви проекции, но в екземпляра на S поставяме само едната от тях.
3. Аналогично, кортежите в T са **проекции** на кортежите на R по C_1, C_2, …, C_k.

Да разгледаме релацията
Movies (<u>title</u>, <u>year</u>, length, type, studioName, <u>starName</u>) от по-горе и да я декомпозираме на следните две релации:
Movies1 (<u>title</u>, <u>year</u>, length, type, studioName)
Movies2 (<u>title</u>, <u>year</u>, <u>starName</u>)
Съответният екземпляр на Movies1 ще е следния:

title	year	length	type	studioName
Star Wars	1977	124	color	Fox
Mighty Duck	1991	104	color	Disney
Wayne's World	1992	95	color	Paramount

Съответният екземпляр на Movies2 ще е следния:

title	year	starName
Star Wars	1977	Carrie Fisher

Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Mighty Duck	1991	Emilio Estevez
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

По този начин елиминираме всички аномалии. Действително:

1. Няма излишество на информация – length и type за всеки филм се появяват само веднъж в екземпляра на Movies1. Повторението на title, year в Movies2 не може да се избегне, тъй като те образуват ключ за филмите, чрез който те се идентифицират.
2. Избегната е аномалията за обновяване – промяната на length или type за всеки филм трябва да се извърши само на едно място в съответния кортеж от Movies1.
3. Избегната е аномалията на изтриването – при премахване на участието на звезда в даден филм, информацията за филма в Movies1 се запазва.

Нормална форма на Boyce-Codd

Като цяло нормалните форми са условия, които ако са изпълнени в дадена релация, то в нея със сигурност няма аномалии от определен вид. Целта на декомпозицията е да разбие релациите на по-малки релации, за които е в сила условието за нормална форма.

Казваме, че една релация R е в **нормална форма на Boyce-Codd** (**BCNF**), ако във всяка нетривиална функционална зависимост
$A_1\ A_2 \ldots A_n \rightarrow B$ за R имаме, че $\{A_1, A_2, \ldots, A_n\}$ е суперключ.
Еквивалентна дефиниция е следната: във всяка нетривиална функционална зависимост $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$ за R имаме, че $\{A_1, A_2, \ldots, A_n\}$ е суперключ.

Например, релацията Movies, която разгледахме преди малко не е в нормална форма на Boyce-Codd. Действително,
title year → length type studioName е нетривиална функционална зависимост, но { title, year} не е суперключ, тъй като ключът е { title, year, starName}.
От друга страна, релациите Movies1 и Movies2 са в нормална форма на Boyce-Codd. Действително, ключът на Movies1 е { title, year } и той със сигурност присъства в лявата част на всяка нетривиална функционална зависимост. В Movies2 няма нетривиални функционални зависимости.

Ще покажем, че всяка релация с два атрибута е в BCNF.
Нека R (A, B) е релация с два атрибута.

1. Ако за R няма нетривиални функционални зависимости, то R естествено е в BCNF. В този случай R има единствен ключ - { A, B}.

2. Нека за R е в сила A → B, но не е в сила B → A. Тогава R има единствен ключ { A} и той се съдържа във всяка нетривиална функционална зависимост за R (тя е единствена – A → B).
3. Нека за R е в сила B → A, но не е в сила A → B. Този случай е аналогичен на предния.
4. Нека за R е в сила A → B и B → A. Тогава R има два ключа { A} и { B}. Нетривиалните функционални зависимости за R са A → B и B → A и те не нарушават BCNF.

Декомпозиция в BCNF

Чрез подходящи декомпозиции всяка схема на релация може да се декомпозира на няколко схеми, така че да са изпълнени следните условия:

1. Всички получени релации да са в BCNF.
2. Информацията трябва да се запазва, т.е. от екземпляри на новополучените релации еднозначно да се възстановява съответният екземпляр на първоначалната релация.

Най-простият вариант е да разбием схемата на релацията на схеми с по два атрибута, които със сигурност ще са в BCNF. При такава произволна декомпозиция, обаче, се нарушава условие 2. както ще видим по-нататък.

Стратегията за декомпозиция, която възприемаме е следната.
Нека $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$ е нетривиална функционална зависимост и $\{ A_1, A_2, \ldots, A_n\}$ не е суперключ. При това ще предполагаме, че в дясната част на функционалната зависимост участват всички атрибути от $\{ A_1, A_2, \ldots, A_n \}^+$. Това условие не е задължително, но може да се приеме като оптимизация. Тогава декомпозираме релацията R на следните две релации:

1. Първата релация има атрибути $A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$.
2. Втората релация има атрибути $A_1, A_2, \ldots, A_n$ и всички останали атрибути на R, които не участват във функционалната зависимост.

Ако новополучените релации не са в BCNF, то към тях прилагаме същата процедура. При това, функционалните зависимости в новите релации се изчисляват чрез проектиране на функционалните зависимости от изходната релация. Процесът на декомпозиране ще е краен, тъй като винаги получаваме релации с по-малко атрибути, а всяка релация с два атрибута е в BCNF.

Да разгледаме пример с релацията
MovieStudio (<u>title</u>, <u>year</u>, length, type, studioName, studioAddress).
Ключът е { title, year } и функционалната зависимост
studioName → studioAddress нарушава BCNF и съдържа максимална дясна част. Следвайки стратегията за декомпозиция разбиваме релацията MovieStudio на две нови релации със следните схеми:
MovieStudio1 (title, year, length, type, studioName) и
MovieStudio2 (studioName, studioAddress).

Лесно се вижда, че ключът на MovieStudio1 е { title, year }, а ключът на MovieStudio2 е { studioName}. Новите релации са в BCNF, тъй като title year → length type studioName е единствената нетривиална функционална зависимост за MovieStudio1, а MovieStudio2 има два атрибута. Така процесът на декомпозиция приключва.

Да разгледаме по-сложен пример със следната релация:
MovieStudioPres (title, year, studioName, president, presAddress).
Функционалните зависимости, които предполагаме са следните:
title year → studioName
studioName → president
president → presAddress.
Единственият ключ на MovieStudioPres е { title, year }, така че последните две функционални зависимости нарушават BCNF.
Да предположим, че започваме с първата функционална зависимост studioName → president. Първо добавяме в дясна част всевъзможните атрибути от { studioName} $^{+}$ и получаваме функционалната зависимост studioName → president presAddress. След това декомпозираме на две релации със следните схеми:
MovieStudioPres1 (title, year, studioName)
MovieStudioPres2 (studioName, president, presAddress).
Първата релация е в BCNF, тъй като единствената функционална зависимост е title year → studioName и { title, year } е единственият ключ.
Втората релация не е в BCNF, тъй като функционалната зависимост president → presAddress се е проектирала в нея, а studioName е единственият ключ. Така разбиваме MovieStudioPres2 на две релации със следните схеми:
MovieStudioPres21 (studioName, president)
MovieStudioPres22 (president, presAddress).
Новополучените релации са в BCNF. Ключът на първата релация е studioName, а ключът на втората релация е president.
Лесно се вижда, че ако започнем с втората функционална зависимост ще достигнем до същия резултат.

Възстановяване на информацията след декомпозиция

Ще покажем, че след декомпозиция извършена по гореописаната стратегия информацията може да се възстанови, т.е. екземплярите на получените релации еднозначно определят екземпляра на декомпозираната релация. Идеята е да използваме **съединение** на кортежите.

За да не претрупваме означенията ще си мислим, че разглеждаме релация с три атрибута R (A, B, C). Да предположим, че функционалната зависимост B → C нарушава BCNF. Ако A → B е функционална зависимост, то { A} е единственият ключ. В противен случай, единственият ключ е { A, B }. И в двата случая декомпозицията по функционалната зависимост B → C води до релации със следните схеми: R1 (A, B), R2 (B, C). Нека t (a, b, c) е кортеж в екземпляр на релацията R.

Тогава проекцията на t в R1 е (a, b), проекцията на t в R2 е (b, c). Съединението представлява следното: от всеки два кортежа от екземплярите на R1 и R2, които се съгласуват по съединяващите атрибути (в случая B), т.е. имат еднакви компоненти, съответни на съединяващите атрибути, образуваме кортеж от екземпляра на R. В случая кортежът (a, b) на R1 и (b, c) на R2 се съгласуват по съединяващия атрибут B и от тях получаваме кортежът t (a, b, c) от екземпляра на R.
Ясно е, че при съединението се възстановяват всички кортежи на екземпляра на релацията R. Въпросът е дали няма да се получат излишни кортежи.
Да предположим, че в екземпляра на R има друг кортеж v (d, b, e). Тогава проекцията на v в R1 е (d, b), проекцията на v в R2 е (b, e). При съединението кортежът (a, b) от екземпляра на R1 се съгласува с кортежа (b, e) от екземпляра на R2. По този начин получаваме кортеж w (a, b, e). Въпросът е дали w е кортеж от екземляра на R. Тъй като B → C е функционална зависимост в R и кортежите t, v имат еднакви компоненти за B, то те трябва да имат еднакви компоненти за C, т.е. получаваме c = e, така че x = w. Така w действително е кортеж от екземпляра на R. По този начин, ако декомпозицията се извършва използвайки функционална зависимост по описаната стратегия, то със сигурност чрез съединение еднозначно можем да възстановяваме екземплярите на декомпозираната релация.
Аргументите, които приведохме естествено се обобщават когато A, B, C са множества от атрибути.

Ще покажем, че при произволна декомпозиция не е сигурно, че екземплярът на първоначалната релация ще може да се възстанови. Нека R (A, B, C) е същата релация, но функционалната зависимост B → C не е в сила. Нека разгледаме екземпляр на R, който се състои от следните два кортежа: (a, b, c), (d, b, e). Да предположим, че декомпозираме релацията на R1 (A, B) и R2 (B, C). Тогава проектираният екземпляр на R1 ще бъде (a, b), (d, b), а проектираният екземпляр на R2 ще бъде (b, c), (b, e). Като извършим съединение получаваме следните кортежи: (a, b, c), (a, b, e), (d, b, c), (d, b, e). Така екземплярът на R не се възстановява правилно – има два излишни кортежа (a, b, e) и (d, b, c). Този пример показва, че не трябва да се извършва безпринципна декомпозиция.

Трета нормална форма

Третата нормална форма е по-слаба от нормалната форма на Boyce-Codd, но тя също се използва.

Да предположим, че е дадена релация със следната схема:
Bookings (title, theater, city).
Кортежът (p, t, c) интерпретираме по следния начин: пиесата p се играе в театъра t, който се намира в град c.
Разумни са следните функционални зависимости:

theater → city и city title → theater.
Чрез втората функционална зависимост предполагаме, че една пиеса не може да се играе по едно и също време в два различни театъра на един и същи град.
Ключовете на релацията са следните: { title, city }, { title, theater}.
Естествено, { city, theater } не е ключ. При това положение е очевидно, че релацията Bookings не е в BCNF - theater → city е нетривиална функционална зависимост и theater не е суперключ. По стратегията за декомпозиция разпадаме Bookings на две релации със следните схеми:
Bookings1 (theater, title)
Bookings2 (theater, city)
При тази декомпозиция, обаче, функционалната зависимост
city title → theater се изгубва. Действително, ако да разгледаме следните екземпляри на Bookings1 и Bookings2:

theater	title
Guild	The Net
Park	The Net

theater	city
Guild	Menlo Park
Park	Menlo Park

Те са коректни, в смисъл, че не нарушават функционалните зависимости, които са проектирани в тях. След съединението им, обаче, се получава следният екземпляр на релацията Bookings:

theater	title	city
Guild	The Net	Menlo Park
Park	The Net	Menlo Park

В този екземпляр на Bookings е нарушена функционалната зависимост title city → theater.
По този начин при декомпозиране към нормална форма на Boyce-Codd не винаги запазваме функционалните зависимости.
Решението на проблема е да отслабим условието на Boyce-Codd.

Казваме, че една релация R е в **трета нормална форма (3NF)**, ако за всяка нетривиална функционална зависимост A_1 A_2 …A_k → B имаме, че { A_1, A_2, …, A_k} е суперключ или B е част от ключ.

Например, релацията Bookings е в 3NF, тъй като във функционалната зависимост theater → city имаме, че city е част от ключ.

Третата нормална форма запазва функционалните зависимости, но при нея има може да има излишества. Може да се покаже, че всяка релация може да се декомпозира подходящо в релации, които са в трета нормална форма.

Също са дефинирани първа и втора нормална форма, но тях няма да ги разглеждаме, тъй като те много рядко се използват на практика.
Четвъртата нормална форма ще разгледаме по-нататък.

Многозначни зависимости

Многозначните зависимости са обобщение на функционалните зависимости. Най-общо те са твърдения за независимост на атрибути.

Ще разгледаме пример за релация в BCNF, в която има излишества. Естествено, тези излишества няма да са породени от функционални зависимости. Най-общо такива излишества произтичат от връзки много към много.
Да разгледаме следният екземпляр на релацията
StarsIn (name, street, city, title, year), която описва участие на актьор във филм заедно с неговия адрес. Предполагаме, че една звезда може да има повече от един адрес.

name	street	city	title	year
Carrie Fisher	123 Maple Str.	Hollywood	Star Wars	1977
Carrie Fisher	5 Locust Ln.	Malibu	Star Wars	1977
Carrie Fisher	123 Maple Str.	Hollywood	Empire Strikes Back	1980
Carrie Fisher	5 Locust Ln.	Malibu	Empire Strikes Back	1980
Carrie Fisher	123 Maple Str.	Hollywood	Return of the Jedi	1983
Carrie Fisher	5 Locust Ln.	Malibu	Return of the Jedi	1983

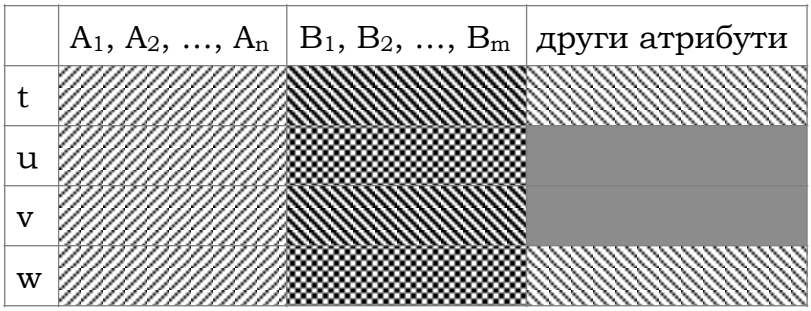
Звездата Carrie Fisher има два адреса и играе в три филма. За да се отрази това в релацията трябва да се комбинира всеки адрес с всеки филм, което води до очевидно излишество. Въпреки това, релацията е в BCNF, тъй като в нея няма функционални зависимости и всички атрибути образуват ключ.

Многозначна зависимост в една релация R наричаме твърдение от следния вид: ако в екземпляр на R, компонентите на кортежите, отговарящи на атрибутите $A_1, A_2, \ldots, A_n$ съвпадат, то компонентите на кортежите, съответни на атрибутите $B_1, B_2, \ldots, B_m$ са независими от стойностите на всички останали атрибути. Бележим многозначната зависимост по следния начин: $A_1\ A_2 \ldots A_n$ Ã $B_1\ B_2 \ldots B_m$.
По-прецизно, казваме че многозначната зависимост
$A_1\ A_2 \ldots A_n$ Ã $B_1\ B_2 \ldots B_m$ е в сила за една релация R, ако за всяка двойка кортежи t, u в екземпляр на R, които се съгласуват по $A_1, A_2, \ldots, A_n$ съществува кортеж v, такъв че:

- v се съгласува с t и u по $A_1, A_2, \ldots, A_n$;
- v се съгласува с t по $B_1, B_2, \ldots, B_m$;
- v се съгласува с u по всички останали атрибути на R.

Естествено, в това правило кортежите t и u могат да участват симетрично, така че съществува кортеж w, такъв че:

- w се съгласува с t и u по $A_1, A_2, \ldots, A_n$;
- w се съгласува с u по $B_1, B_2, \ldots, B_m$;
- w се съгласува с t по всички останали атрибути на R.



Като следствие при фиксирани стойности на $A_1, A_2, \ldots, A_n$ съответните стойности на $B_1, B_2, \ldots, B_m$ и всички останали атрибути се комбинират по всевъзможни начини в различни кортежи на екземпляра на релацията R.

Както при функционалните зависимости можем да допускаме някои от атрибутите $A_1, A_2, \ldots, A_n$ да са сред $B_1, B_2, \ldots, B_m$. Изрично ще отбележим, че за разлика от функционалните зависимости за многозначните зависимости не е в сила правилото за разделяне.
По-долу ще дадем пример, който показва това.
Може да се провери, че правилото за комбиниране е в сила за многозначните зависимости.

В примера от по-горе съществува следната многозначна зависимост: name Ã street city. Действително, всевъзможните адреси на звездата Carrie Fisher се комбинират с всевъзможните филми, в които тя участва.

Правила за многозначните зависимости

Ще разгледаме някои правила за многозначните зависимости, които наподобяват правилата за функционалните зависимости, но има някои разлики.

Ако за една релация R е в сила многозначната зависимост
$A_1\ A_2 \ldots A_n$ Ã $B_1\ B_2 \ldots B_m$, то за нея е в сила многозначната зависимост
$A_1\ A_2 \ldots A_n$ Ã $C_1\ C_2 \ldots C_k$, където $C_1, C_2, \ldots, C_k$ съдържат $B_1, B_2, \ldots, B_m$ и някои от $A_1, A_2, \ldots, A_n$. Също за R е в сила многозначната зависимост
$A_1\ A_2 \ldots A_n$ Ã $D_1\ D_2 \ldots D_r$, където $D_1, D_2, \ldots, D_r$ са тези от $B_1, B_2, \ldots, B_m$, които не с от $A_1, A_2, \ldots, A_n$. Тези две правила наричаме **правила за тривиалните зависимости**.

Ако за една релация R са в сила многозначните зависимости
$A_1\ A_2 \ldots A_n$ Ã $B_1\ B_2 \ldots B_m$ и $B_1\ B_2 \ldots B_m$ Ã $C_1\ C_2 \ldots C_k$, то за R е в сила многозначната зависимост $A_1\ A_2 \ldots A_n$ Ã $C_1\ C_2 \ldots C_k$. Това правило наричаме **транзитивно правило**. Лесно се съобразява, че то е коректно чрез дефиницията за многозначна зависимост.

Ще покажем с пример, че за многозначните зависимости не е в сила правилото за разделяне. В релацията от по-горе е в сила многозначната зависимост name Ã street city, но не е в сила многозначната зависимост name Ã street. Действително, да разгледаме следните два кортежа:

Carrie Fisher	123 Maple Str.	Hollywood	Star Wars	1977
Carrie Fisher	5 Locust Ln.	Malibu	Star Wars	1977

За тях по дефиниция трябва да съществува следния кортеж:

Carrie Fisher	123 Maple Str.	Malibu	Star Wars	1977

Това, обаче, не е изпълнено.
За многозначни зависимости има две нови правила.

Ако функционалната зависимост $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$ е в сила за една релация R, то за R е в сила многозначната зависимост $A_1\ A_2\ \ldots A_n$ Ã $B_1\ B_2\ \ldots B_m$. Действително, нека t и u са кортежи в екземпляр на R които се съгласуват по $A_1, A_2, \ldots A_n$. Тогава кортежът v = u се съгласува с t по $B_1, B_2, \ldots, B_m$, тъй като t и u се съгласуват по $B_1, B_2, \ldots, B_m$. Тук сме използвали, че за R е в сила $A_1\ A_2\ \ldots A_n \rightarrow B_1\ B_2\ \ldots B_m$. Също v се съгласува с u по всички останали атрибути, тъй като v = u. Аналогично, можем да изберем w = t.

За многозначните зависимости е в сила следното **правило за допълнение**, което не е в сила за функционалните зависимости:
ако за R е в сила многозначната зависимост $A_1\ A_2\ \ldots A_n$ Ã $B_1\ B_2\ \ldots B_m$, то за R е в сила многозначната зависимост $A_1\ A_2\ \ldots A_n$ Ã $C_1\ C_2\ \ldots C_k$, където $C_1, C_2, \ldots, C_k$ са всички атрибути на R без $A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$.
Това правило е коректно - разменяме ролите на v и w в дефиницията.
Като пример, тъй като за горната релация е в сила многозначната зависимост name Ã street city, то за нея е в сила многозначната зависимост name Ã title year.

Четвърта нормална форма

Излишеството, което се поражда от многозначните зависимости може да се избегне чрез подходяща декомпозиция, подобно на декомпозицията при BCNF. В резултат на тази декомпозиция се елиминират излишествата, породени както от многозначните, така и от функционалните зависимости.

Казваме, че многозначната зависимост $A_1\ A_2\ \ldots A_n$ Ã $B_1\ B_2\ \ldots B_m$ е **нетривиална**, ако:

1. $\{ B_1, B_2, \ldots, B_m \} \cap \{ A_1, A_2, \ldots, A_n \} = \varnothing$.
2. $A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$ не изчерпват всички атрибути на R.

Казваме, че една релация R е в **четвърта нормална форма** (4NF), ако във всяка нетривиална многозначна зависимост $A_1\ A_2\ \ldots A_n$ Ã $B_1\ B_2\ \ldots B_m$ $\{ A_1, A_2, \ldots, A_n\}$ е суперключ.

Това означава, че всяка нетривиална многозначна зависимост всъщност е функционална и в лявата и част има суперключ.

Примерната релация по-горе не е в четвърта нормална форма. Действително, за нея е в сила нетривиалната многозначна зависимост name Ã street city и name не е суперключ - единственият ключ (суперключ) на тази релация е множеството от всички атрибути.

Естествено, всяка релация, която се намира в четвърта нормална форма се намира и в трета нормална форма, т.е. 4NF ⇒ BCNF. Това следва от факта, че всяка функционална зависимост е многозначна.
Обратното не е вярно, както показва разгледания пример.

Всяка релация с два атрибута се намира в четвърта нормална форма, тъй като в нея не може да има нетривиални многозначни зависимости.

Декомпозиция в 4NF

Алгоритъмът за декомпозиране в 4NF е аналогичен на този за BCNF. Нека A_1 A_2 …A_n Ã B_1 B_2 …B_m е нетривиална многозначна зависимост, която нарушава 4NF. Тогава разбиваме релацията R на две релации със следните схеми:

- първата съдържа атрибутите A_1, A_2, …, A_n, B_1, B_2, …, B_m;
- втората съдържа атрибутите A_1, A_2, …, A_n и всички останали атрибути, които не участват в многозначната зависимост.

Ако новополучените релации не са в 4NF, то към тях прилагаме същата процедура. Този процес ще е краен, тъй като винаги получаваме релации с по-малко атрибути, а всяка релация с два атрибута е в 4NF.
При това с аналогични аргументи може да се покаже, че от екземпляри на новополучените релации чрез съединение на кортежите еднозначно се възстановява съответният екземпляр на оригиналната релация.
Причината е, че декомпозицията не е безпринципна, а се извършва на базата на многозначните зависимости.

Както вече споменахме, 4NF влече BCNF, която от своя страна влече 3NF. Свойствата на декомпозицията при трите нормални форми могат да се обобщят в следната таблица:

свойство	3NF	BCNF	4NF
отстранява излишества, породени от функционални зависимости	в повечето случаи	да	да
отстранява излишества, породени от многозначни зависимости	не	не	да
запазва функционалните зависимости	да	в някои случаи	в някои случаи

запазва многозначните зависимости	в някои случаи	в някои случаи	в някои случаи

Релационна алгебра

Релационната алгебра е нотация за описване на заявки към релации. Множеството от операциите, които осигурява релационната алгебра не е пълно по Тюринг, в смисъл че съществуват операции, които не могат да се опишат със средствата на релационната алгебра, но могат да се опишат например на C++ или на кой да е от обикновените езици за програмиране. Предимството е, че това дава възможност за по-ефективно оптимизиране на заявките.
Операциите от релационната алгебра формално се извършват върху множества от кортежи, т.е. върху екземпляри на релации. Обикновено, обаче, СУБД използват друг модел на релациите, в който екземпляр на една релация е мултимножество от кортежи, т.е. допуска се повторение на кортежи в релациите. Причината е, че операциите с мултимножества се реализират по-ефективно от операциите с множества - при тях не трябва да се проверява условието за единственост на кортежите.

Примерна схема на база от данни

По-долу в някои примери ще използваме базата от данни, която има следната схема:

Movie (title : string,
year : integer,
length : integer,
inColor : boolean,
studioName : string,
producerC# : integer)
StarsIn (movietitle : string,
movieyear : integer,
starname : string)
MovieStar (name : string,
address : string,
gender : char,
birthdate : date)
MovieExec (name : string,
address : string,
cert# : integer,
netWorth : integer)
Studio (name : string,
address : string,
presC# : integer)

Схемата се състои от пет релации. Посочили сме атрибутите на всяка от релациите, заедно с техните области от стойности. Ключовите атрибути във всяка релация са подчертани.
Новите елементи, които не сме разглеждали са следните:

- добавени са продуценти на филми, които се идентифицират чрез техния сертификационен номер; в релацията Movie атрибутът producerC# е номерът на продуцента на филма;

- добавен е атрибут пол за звездите от тип символ - ‘F’ за жена или ‘M’ за мъж;
- атрибутът inColor приема стойност true, ако филмът е цветен и стойност false, ако е черно-бял;
- президентите на студията, подобно на продуцентите на филми се идентифицират със сертификационни номера.

Основи на релационната алгебра

В релационната алгебра операндите, които използваме са конкретни релации или променливи, които означават релации. В класическата релационна алгебра, както споменахме, релациите са множества от кортежи. Традиционните операции в релационната алгебра делим на четири групи:

- обикновените операции с множества - обединение, сечение, разлика, приложени към релации;
- операции, които премахват части от релация – селекция, която премахва кортежи (редове) и проекция, която премахва атрибути (колони);
- операции, които комбинират кортежите на две релации - декартово произведение, което съединява кортежите на две релации по всички възможни начини и различни други операции за съединение, които съединяват само част от тези кортежи;
- операция, наречена преименуване, която не се отразява върху кортежите на една релация, но променя нейната схема.

Изразите, които строим с помощта на операциите и операндите на релационната алгебра наричаме **заявки**.

Операции с множества върху релации

Когато прилагаме операция с множества към две релации R и S, то трябва да са изпълнени следните условия:

- R и S трябва да имат едни и същи схема с идентични множества от атрибути, както и области от стойности за тези атрибути;
- преди да се изпълни операцията колоните на R и S трябва да се подредят по един и същи начин.

Операциите, които допускаме са **обединение**, **сечение** и **разлика** на релации. Във всички случаи получаваме нова релация, която има идентична схема на R и S и множество от кортежи, което е съответно обединение, сечение или разлика на множествата кортежи на R и S.
Обединението бележим по следния начин: $R \cup S$.
Сечението бележим по следния начин: $R \cap S$.
Разликата бележим по следния начин: R - S.
Да отбележим, че разликата не е комутативна операция, т.е. релациите R - S и S - R в общия случай са различни.
Понякога се налага да изпълняваме операции с множества върху релации, които имат еднакъв брой атрибути с идентични множества от стойности, но с различни имена. В такъв случай можем да използваме

предварително операцията за преименуване, за да унифицираме напълно двете схеми на релациите.
Ще разгледаме някои примери.
Нека R е следната релация:

name	address	gender	birthdate
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Нека S е следната релация:

name	address	gender	birthdate
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Обединението R ∪ S е следната релация:

name	address	gender	birthdate
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Сечението R ∩ S е следната релация:

name	address	gender	birthdate
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99

Разликата R - S е следната релация:

name	address	gender	birthdate
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Разликата S - R е следната релация:

name	address	gender	birthdate
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Проекция

Операцията **проекция**, приложена върху релация R се използва за образуване на нова релация, в която участват само някои от колоните на R. Бележим $\pi_{A_1,A_2,\ldots,A_n}(R)$, естествено A_1, A_2, …, A_n са част от атрибутите на R. Схемата на новата релация се състои само от атрибутите A_1, A_2, …, A_n. Естествено, кортежите на новата релация са

проекции на кортежите на R върху атрибутите A_1, A_2, …, A_n. Ако някои кортежи от R имат еднакви стойности за A_1, A_2, …, A_n, то в релацията $\pi_{A_1,A_2,\ldots,A_n}(R)$ включваме само една от съответните проекции - в релационната алгебра, базирана на множества не допускаме дублиране на кортежите. Като пример, ако R е релацията от по-горе, то релацията $\pi_{name,\ birthdate}(R)$ има следния вид:

name	birthdate
Carrie Fisher	9/9/99
Mark Hamill	8/8/88

Селекция

Резултатът от **селекцията**, приложена върху релация R е друга релация с множество кортежи, което е подмножество на кортежите на R.
Кортежите, които се включват в резултата са точно онези кортежи от R, които удоволетворяват някакво условие C за стойностите на атрибутите.
Резултатът от селекцията бележим по следния начин: $\sigma_C(R)$.
Схемата на релацията $\sigma_C(R)$ съвпада със схемата на R.
C е условен израз, в който като операнди участват константи или имена на атрибути на R. В C могат да се използват различните операции за сравнение - =, <, >, ≠, <=, >=, както и логическите съюзи and, not, or.
Условието C се прилага към всеки кортеж t на релацията R, като името на всеки атрибут в условието C се замества със съответната му стойност от кортежа t. Ако условието се оцени като истина, то кортежът t се включва в релацията $\sigma_C(R)$, в противен случай не се включва.
Ще разгледаме пример. Нека R е следната релация:

title	year	length	inColor	studioName	producerC#
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890
Wayne's World	1992	95	true	Paramount	99999

Тогава релацията $\sigma_{length >= 100}(R)$ има следния вид:

title	year	length	inColor	studioName	producerC#
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890

Действително, за първия кортеж след заместването условието C приема вида 124 >= 100 и се оценява с истина, за втория кортеж условието C приема вида 104 >= 100 и се оценява с истина, а за третия кортеж C приема вида 95 >= 100 и се оценява с лъжа.

Релацията $\sigma_{length>=100\ AND\ studioName =' Fox'}(R)$ има следния вид:

title	year	length	inColor	studioName	producerC#
Star Wars	1977	124	true	Fox	12345

Действително, за първия кортеж след заместването условието C приема вида 124 >= 100 AND ‘Fox’ = ‘Fox’ и се оценява с истина, тъй като и двете части на конюнкцията се оценяват с истина, за втория кортеж условието C приема вида 104 >= 100 AND ‘Disney’ = ‘Fox’ и се оценява с лъжа, тъй като втората част на конюнкцията се оценява с лъжа, а за третия кортеж C приема вида 95 >= 100 AND ‘Paramount’ = ‘Fox’ и се оценява с лъжа, тъй като и двете части на конюнкцията се оценяват с лъжа.

Декартово произведение

Декартово произведение на релациите R и S е нова релация, която бележим с R x S и която има за кортежи всевъзможните съединения на кортеж от R с кортеж от S. Схемата на R x S е обединение на схемите на R и S с тази особеност, че ако R и S имат атрибути с еднакви имена предварително трябва да се извърши преименуване на дублиращите се атрибути. Използваме така наречената **точкова нотация** - за да различаваме атрибутът A на релациите R и S записваме R.A за атрибута от R и S.A за атрибута от A. Ще считаме, че атрибутите на R предшестват атрибутите на S в схемата на R x S.
Ще разгледаме пример.
Нека R е следната релация:

A	B
1	2
3	4

Нека S е следната релация:

B	C	D
2	5	6
4	7	8
9	10	11

Тогава R x S е следната релация:

A	R.B	S.B	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Тъй като B е атрибут и на двете релации сме използвали R.B и S.B в схемата за R x S. За останалите атрибути не е нужно да се извършва

преименуване, тъй като те не се дублират. В релацията R x S всеки от кортежите на R е съединен с всеки от кортежите на S.

Естествено съединение

Много често се налага при образуване на декартово произведение на релации да не се извършват всевъзможни съединения на кортежи. **Естествено съединение** на две релации R и S е нова релация, която означаваме с R ⋈ S. Нейните кортежи са всевъзможни съединения на кортеж от R с кортеж от S, които се съгласуват по общите атрибути на релациите R и S. По-прецизно, нека A_1, A_2, …, A_n са общите атрибути в схемите на релациите R и S. Тогава схемата на R ⋈ S е теоретико-множествено обединение на схемите на R и S, т.е. не се извършва преименуване и атрибутите A_1, A_2, …, A_n участват само по веднъж в схемата на R ⋈ S. Два кортежа r и s се съединяват успешно, ако те се съгласуват по стойностите на A_1, A_2, …, A_n. Тогава съответният кортеж на R ⋈ S се съгласува по всички атрибути на R с кортежа r и по всички атрибути на S с кортежа s. Това естествено е възможно точно когато двата кортежа r и s са успешно съединени.
Ще отбележим, че тази операция за естествено съединение е същата, която използвахме при възстановяване на информацията след декомпозиция в нормална форма на Boyce-Codd.

Ще разгледаме примери.
За релациите от по-горе R и S релацията R ⋈ S има следния вид:

A	B	C	D
1	2	5	6
3	4	7	8

Единственият общ атрибут на R и S е B. Така кортеж от R се съгласува с кортеж от S точно когато стойностите им за B съвпадат. Резултатните кортежи в R ⋈ S включват компоненти за следните атрибути:
A (от R), B (от R или S), C (от S) и D (от S). Както се вижда, третият кортеж на S не се съгласува с никой от кортежите на R. Такъв кортеж се нарича **висящ кортеж**.
Нека U е следната релация:

A	B	C
1	2	3
6	7	8
9	7	8

Нека V е следната релация:

B	C	D
2	3	4

2	3	5
7	8	10

Тогава релацията ⋈ има следния вид:

A	B	C	D
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

Общите атрибути на U и V са B и C. Така кортеж от U се съгласува с кортеж от V точно когато стойностите им за B и C съвпадат.
В този пример първият кортеж на U се съгласува с първите два кортежа на V, а вторият и третият кортеж на U се съгласуват с третия кортеж на V. При това естествено съединение няма висящи кортежи.

Тита-съединение

При естественото съединение кортежите на релациите се комбинират по точно едно условие - съвпадение на общите атрибути. Понякога се налага кортежи да се съединяват и по други условия.
Тита-съединение на релациите R и S е нова релация, която означаваме по следния начин: $R \bowtie_C S$. Нейната схема е обединение на схемите на R и S, при това се извършва преименуване на атрибутите, които се дублират. C е условен израз, подобен на условния израз от селекцията, но в него могат да участват имена на атрибути и на R и на S.
Кортежите на $R \bowtie_C S$ получаваме по следния начин:

- образуваме релацията R x S;
- избираме онези кортежи на R x S, които удоволетворяват условието C (след заместване на имената на атрибутите със съответните стойности).

Да отбележим, че при тита-съединението, за разлика от естественото съединение в резултатната схема общите атрибути не се сливат в едно копие. Причината е, че при естественото съединение това има смисъл, тъй като условието за съединение е съвпадащи атрибути. При тита-съединението условието може да е произволно.
Ще разгледаме примери.
За релациите U и V от по-горе релацията $U \bowtie_{A < D} S$ изглежда така:

A	U.B	U.C	V.B	V.C	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10

9	7	8	7	8	10

Релацията U $\bowtie_{A < D \text{ AND } U.B \neq V.B}$ S изглежда така:

A	U.B	U.C	V.B	V.C	D
1	2	3	7	8	10

Комбиниране на операциите

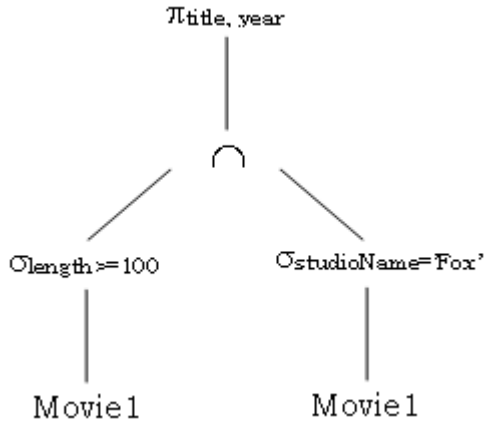
За конструиране на по-сложни заявки трябва да комбинираме сравнително простите операции, които въведохме. С други думи, като операнди можем да използваме както дадените релации, така и резултатните релации от вече извършени операции.
По този начин можем да получаваме произволно сложни заявки.

Сложните заявки ще записваме с изрази, като всеки израз се получава чрез прилагане на операция към негови подизрази. За еднозначно прочитане на подизразите ще използваме скоби. Възможно е за представяне на изразите да използваме обичайната дървовидна структура.

Ще разгледаме примери с нормализираните релации
Movie1 (title, year, length, type, studioName),
Movie2 (title, year, starName).
Да предположим, че имаме следната заявка: “Да се намерят заглавията и годините на филмите, произведени от Fox и дълги поне 100 минути.”
Едно решение е следното:

1. Намираме всички филми, дълги поне 100 минути.
2. Намираме всички филми, произведени от Fox.
3. Намираме сечението на 1. и 2.
4. Намираме проекцията на 3. по заглавие и година.

Съответният израз от релационната алгебра се представя чрез следното дърво:



Във вътрешните възли на дървото стоят операциите. Тъй като използваме и унарни операции, то някои възли имат само един наследник. В листата на дървото са дадените релации. Съвременните СУБД получават заявките, представени именно по този начин. Оптимизацията и изпълнението на заявките се извършват въз основа на това дърво.

Друг начин да запишем заявката е чрез обичайната за записване на израз с поставяне на скоби по следния начин:
$\pi_{title,\ year}\ (\sigma_{length>=100}\ (Movie1) \cap \sigma_{studioName='Fox'}\ (Movie1))$.

Много често е възможно различни изрази от релационната алгебра да водят до едно и също изчисление. Например, дадената заявка може да се представи и чрез следния израз:
$\pi_{title,\ year}\ (\sigma_{length>=100\ AND\ studioName='Fox'}\ (Movie1))$.

По-общо в сила са следните закони:
$\sigma_C\ (R) \cap \sigma_D\ (R) = \sigma_{C\ AND\ D}\ (R)$
$\sigma_C\ (R) \cup \sigma_D\ (R) = \sigma_{C\ OR\ D}\ (R)$
$\sigma_C\ (R) - \sigma_D\ (R) = \sigma_{C\ AND\ NOT\ D}\ (R)$

Като друг пример да разгледаме следната заявка: “Да се намерят звездите, които участват във филми, дълги поне 100 минути.” Естествено, тук трябва да използваме естествено съединение за да възстановим информацията от декомпозираните релации. Съответният израз от релационната алгебра е следния:
$\pi_{starName}\ (\sigma_{length>=100}\ (Movie1 \bowtie Movie2))$.

Операция за преименуване

Операцията за преименуване се използва както за преименуване на атрибутите на една релация, така и за промяна на нейното име. Резултатът от преименуването на релация R бележим по следния начин: $\rho_{S(A_1, A_2, \ldots, A_n)}(R)$. Изискването е n да съвпада с броя на атрибутите на R. Резултатната релация има име S и схема S (A_1, A_2, …, A_n). Кортежите на S съвпадат с кортежите на R. Възможно е да променяме само името на релацията R, без да променяме имената на атрибутите като използваме операцията в следната форма: $\rho_S(R)$. Схемата и екземплярът на R се запазват при тази операция, а името се променя на S.

Ще разгледаме пример.
Нека R е следната релация:

A	B
1	2
3	4

Нека S е следната релация:

B	C	D
2	5	6
4	7	8
9	10	11

При образуване на декартово произведение на R и S можем да не искаме да използваме точковата нотация. В такъв случай явно трябва да преименуваме съвпадащите атрибути.
Например, релацията R x $\rho_{S\,(X,C,D)}$ (S) изглежда така:

A	B	X	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Друг начин е да извършим декартовото произведение и след това да преименуваме. Изразът $\rho_{RS\,(A,B,X,C,D)}$ (R x S) се изчислява до същата релация, с тази разлика че в този случай тя има име - RS.

Зависими и независими операции

Някои от операциите на релационната алгебра могат да бъдат изразени с помощта на други операции.
Например, сечението $R \cap S$ може да се изрази чрез обединение и разлика по следния начин: $R \cap S = R - (R - S)$.
Естественото съединение и тита-съединението могат да се изразят чрез декартово произведение, проекция и селекция по следния начин:
$R \bowtie_C S = \sigma_C (R \times S)$, $R \bowtie S = \pi_L (\sigma_C (R \times S))$. Тук условието C има вида:
$R.A_1 = S.A_1$ AND $R.A_2 = S.A_2$ AND …AND $R.A_n = S.A_n$, където A_1, A_2, …, A_n са общите атрибути на R и S. Списъкът от атрибути L започва с атрибутите на R, последвани от тези атрибути на S, които не са атрибути на R.
Шестте останали операции - обединение, разлика, декартово произведение, селекция, проекция и преименуване образуват независимо множество, т.е. никоя от тях не може да се изрази чрез останалите.

Линейна нотация

Ще разгледаме още едно представяне на сложните заявки с повече от една операция, което наричаме **линейна нотация**. При него на

междинните релации, които отговарят на вътрешните възли на дървото се поставят имена. Самата сложна заявка се представя като последователност от присвоявания. Те трябва да са разположени в такъв ред, че за всеки връх на дървото N преди да изчисляваме релация, отговаряща на N трябва да сме изчислили всички релации, отговарящи на децата на N. Ще използваме следната нотация за присвояванията:
$R(A_1, A_2, \ldots, A_n)$:= израз_от_релационната_алгебра.
В лявата страна стои името и схемата на междинната релация, в дясната страна стои израз от релационната алгебра, при това операндите в него са или дадени релации или получени релации в по-горни присвоявания. В релацията с име Answer получаваме резултатът - тя отговаря на корена на дървото.
Обикновено в изразите в дясна част се използва само една операция, т.е. за всеки междинен връх на дървото има отделно присвояване.
Възможно е, обаче, в изразите в дясна част да се комбинират няколко операции - например с цел повишаване на ефективността.
Като пример ще опишем първата заявка от по-горе в линейна нотация:
$R(t, y, l, t, s) := \sigma_{length>=100}(Movie1)$
$S(t, y, l, t, s) := \sigma_{studioName='Fox'}(Movie1)$
$T(t, y, l, t, s) := R \cap S$
$Answer(title, year) := \pi_{t, y}(T)$
Възможно е последните две присвоявания да се комбинират в едно:
$R(t, y, l, t, s) := \sigma_{length>=100}(Movie1)$
$S(t, y, l, t, s) := \sigma_{studioName='Fox'}(Movie1)$
$Answer(title, year) := \pi_{t, y}(R \cap S)$

Първите СУБД са били реализирани чрез линейна нотация.

Операции върху мултимножества

В съвременните СУБД екземплярите на релациите са мултимножества от кортежи, а не множества от кортежи, както е в класическата релационна алгебра. С други думи, допуска се в една релация да се дублират кортежи. Причината е, че много от операциите се реализират по-ефективно, когато се прилагат към мултимножества.
Например, при обединение на релации R, S с екземпляри мултимножества кортежите на S просто се добавят към кортежите на R без да се премахват дубликатите. Също, при проекция на една релация R може два различни кортежа да се проектират в еднакви кортежи.
Ако изискваме резултатът от проекцията да е множество от кортежи, то трябва допълнително след извършване на проекцията да премахваме дублиращите се кортежи. При мултимножества няма такова изискване.

Обединение, сечение и разлика на мултимножества

Нека R и S са релации с екземпляри мултимножества.
Обединието $R \cup S$ се образува по следния начин:

всеки кортеж, който присъства n пъти в R и m пъти в S присъства m+n пъти в R ∪ S.
Сечението R ∩ S се образува по следния начин:
всеки кортеж, който присъства n пъти в R и m пъти в S присъства min (m, n) пъти в R ∩ S.
Разликата R - S се образува по следния начин:
всеки кортеж, който присъства n пъти в R и m пъти в S присъства max (0, n-m) пъти в R - S. С други думи, всяко едно срещане на кортежа в S елиминира едно срещане на кортежа в R.
Ще разгледаме някои примери.
Нека R е следната релация:

A	B
1	2
3	4
1	2
1	2

Нека S е следната релация:

A	B
1	2
3	4
3	4
5	6

Тогава R ∪ S е следната релация:

A	B
1	2
3	4
1	2
1	2
1	2
3	4
3	4
5	6

Релацията R ∩ S изглежда по следния начин:

A	B
1	2

3	4

Релацията R - S изглежда по следния начин:

A	B
1	2
1	2

Релацията S - R изглежда по следния начин:

A	B
3	4
5	6

Всяко множество естествено може да се разглежда като мултимножество. Лесно се съобразява, че операциите сечение и разлика на мултимножества, приложени върху множества са идентични на операциите сечение и разлика на множества. Това, обаче, не е в сила за операцията обединение. Например, ако R и S са релации с екземпляри множества от кортежи и един кортеж t се съдържа както в R, така и в S, то чрез обединението на мултимножества получаваме релация, в която кортежът t се среща поне два пъти, т.е. дори не получаваме множество.

Изрично ще отбележим, че някои от обичайните закони за операциите с множества престават да бъдат в сила за съответните операции с мултимножества. Например, законът $(R \cup S) - T = (R - T) \cup (S - T)$, който е в сила за множества не е в сила за мултимножества.
Действително, нека един кортеж t се среща точно по веднъж в R, S и T. Тогава t се среща точно веднъж в $(R \cup S) - T$, но не се среща в $(R - T) \cup (S - T)$. Други закони, обаче, остават в сила. Например, $R \cup S = S \cup R$, $R \cap S = S \cap R$, $(R \cup S) \cup T = R \cup (S \cup T)$, $(R \cap S) \cap T = R \cap (S \cap T)$.

Проекция на мултимножества

Проекцията на една релация R с екземпляр мултимножество се осъществява по аналогичен начин на проекцията при множества - всеки кортеж се проектира независимо. Разликата е, че в резултата не се елиминират дубликатите.
Ще разгледаме пример.
Нека R е следната релация:

A	B	C
1	2	5
3	4	6
1	2	7
1	2	8

Тогава релацията $\pi_{A,B}$ (R) изглежда по следния начин:

A	B
1	2
3	4
1	2
1	2

Ако прилагаме проекция на множества, кортежът (1, 2) ще се среща само един път в резултата.

Селекция на мултимножества

Селекцията на една релация R с екземпляр мултимножество се осъществява по аналогичен начин на селекцията при множества - към всеки кортеж на R се прилага независимо условието за селекция. Отново разликата е, че в резултата не се елиминират дублиращите се кортежи. Ще разгледаме пример.
Нека R е следната релация:

A	B	C
1	2	5
3	4	6
1	2	7
1	2	7

Тогава релацията $\sigma_{C>6}$ (R) има следния вид:

A	B	C
1	2	7
1	2	7

Декартово произведение на мултимножества

Нека R, S са релации с екземпляри мултимножества. Тогава декартовото произведение R x S се изчислява по естествения начин - всеки кортеж на R независимо се съединява с всеки кортеж на S.
С други думи, ако един кортеж t се среща m пъти в R и един кортеж s се среща n пъти в S, то съединението на кортежите t и s ще се среща m.n пъти в R x S. Ще разгледаме пример.
Нека R е следната релация:

A	B
1	2
1	2

Нека S е следната релация:

B	C
2	3
4	5
4	5

Тогава R x S е следната релация:

A	R.B	S.B	C
1	2	2	3
1	2	4	5
1	2	4	5
1	2	2	3
1	2	4	5
1	2	4	5

Съединение на мултимножества

Нека R, S са релации с екземпляри мултимножества.
Тогава при съединение на R и S всеки кортеж на R независимо се изпробва да се съедини с всеки кортеж на S. В резултата не се премахват дубликатите.
Ще разгледаме примери за естествено съединение и тита-съединение с релациите R и S, които въведохме непосредствено по-горе.

Релацията $R \bowtie S$ изглежда по следния начин:

A	B	C
1	2	3
1	2	3

Релацията $R \bowtie_{R.B < S.B} S$ изглежда по следния начин:

A	R.B	S.B	C
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

Разширени операции на релационната алгебра

Разгледаните операции върху множества и мултимножества са в основата на всеки съвременен език за заявки. Сега ще разгледаме други операции, които разширяват функционалността на класическата релационна алгебра. Тези операции са реализирани в SQL.

1. Операция δ за премахване на дублиращите се кортежи.
2. Агрегатни операции - SUM, AVG, MIN, MAX, COUNT. Тези операции не са от релационната алгебра, т.е. те не преобразуват релации в релации. Използват се при операцията за групиране.
3. Операция γ за групиране - групира кортежите на една релация по еднакви стойности на част от атрибутите.
   Върху останалите атрибути се прилагат агрегатни функции в рамките на всяка група.
4. Разширена проекция - чрез нея могат да се въвеждат нови атрибути, които се изчисляват на базата на съществуващите атрибути.
5. Операция τ за сортиране - превръща релацията в списък от кортежи, сортирани по един или повече атрибути.
6. Операция за външно съединение - вариант на съединение, чрез което се избягва загубата на информация от висящите кортежи.

Операция за премахване на дубликати

Нека R е релация с екземпляр мултимножество. **Операцията за премахване на дубликати** се означава по следния начин: δ (R). Резултатната релация има същата схема, както R и в нейния екземпляр всеки кортеж на R се среща точно по един път. С други думи, операцията δ преобразува екземпляр на релация от мултимножество в множество. Ще разгледаме пример. Нека R е следната релация:

A	B
1	2
3	4
1	2
1	2

Тогава δ (R) е следната релация:

A	B
1	2
3	4

Агрегатни операции

Агрегатните операции се прилагат върху множества или мултимножества от атомарни стойности в една колона на релация. Стандартно се поддържат следните агрегатни операции:

1. SUM - намира сумата от стойностите в колона с числа.
2. AVG - намира средното аритметично на стойностите в колона с числа.
3. MIN - намира най-малката стойност в колона с числа (относно стандартната наредба на числата) или с низове (относно лексикографската наредба на низовете).
4. MAX - намира най-голямата стойност в колона с числа (относно стандартната наредба на числата) или с низове (относно лексикографската наредба на низовете).
5. COUNT - намира броят на стойностите в произволна колона.

Ще разгледаме примери с горната релация R.
SUM (B) = 2+4+2+2 = 10
AVG (B) = (2+4+2+2)/4 = 2.5
MIN (A) = 1
MAX (B) = 4
COUNT (A) = COUNT (B) = 4

Операция за групиране

Много често се налага агрегатна функция да не се прилага върху цяла колона на една релация, а кортежите да се разбият на групи и агрегацията да се извършва независимо в рамките на всяка група. Нека R е релация. **Операцията за групиране** означаваме по следния начин: γ_L (R). L е списък от елементи от следните два вида:

1. Атрибут на R, по който се прилага групирането. Такъв елемент се нарича **групиращ атрибут**.
2. Агрегатна функция, приложена към атрибут на R, последвана от стрелка и име на атрибута в резултата. Такъв елемент се нарича **агрегиран атрибут**.

Схемата на резултатната релация се получава от имената на атрибутите в L.
Екземплярът на резултатната релация се получава по следния начин:

1. Разбиваме кортежите на R на групи. Всяка група се състои от всички кортежи, които покомпонентно се съгласуват по стойностите на групиращите атрибути. Ако няма групиращи атрибути, то всички кортежи в R образуват една група.
2. За всяка група в резултатната релация се генерира точно един кортеж, който се състои от стойностите на групиращите атрибути за тази група и агрегациите на кортежите в групата върху агрегираните атрибути.

Ще разгледаме два примера.
Разглеждаме релацията Movie (title, year, length, type, studioName).
Да предположим, че трябва да изпълним следната заявка:
“Да се намери общата дължина на филмите на всяко студио.”
Тогава съответният израз, който трябва да използваме е следния:
γ studioName, SUM (length)→totalLength (Movie).

Разглеждаме релацията StarsIn (title, year, starName).
Да предположим, че трябва да изпълним следната заявка:
“Да се намерят годините на най-ранните участия на тези звезди, които са играли в поне три филма.”
Едно възможно решение е да групираме по starName, като агрегираме title чрез COUNT и year чрез MIN, след което да извършим подходяща селекция и проекция. Съответният израз е следния:
$\pi_{starName,\ minYear}\ (\sigma_{ctTitle>=3}\ (\gamma\ starName,\ MIN\ (year)\rightarrow minYear,\ COUNT\ (title)\rightarrow ctTitle\ (StarsIn)))$.

Да отбележим, че операторът за премахване на дубликати е частен случай на оператора за групиране. Действително, ако A_1, A_2, …, A_n са атрибутите на релация R, то изразът δ (R) е еквивалентен на израза $\gamma_{A_1, A_2, \ldots, A_n}$ (R). Действително, при изчисляване на последния израз се групират еднаквите кортежи и в резултата се включва по един кортеж от всяка група.

Разширена проекция

Нека R е релация. При класическата проекция π_L (R), L е списък от атрибути на релацията R. При **разширената проекция**, която се означава по същия начин, L е списък от елементи от следните видове:

1. Атрибут на R.
2. Израз от вида Expr → R, където Expr е израз, включващ атрибути на R, константи, аритметични операции или операции за стрингове, а R е новото име на атрибута, който се получава като резултат от изчисляването на израза Expr.

Схемата на резултатната релация се получава от имената на атрибутите в L. В екземпляра на резултатната релация има по един кортеж за всеки кортеж от екземпляра на R. Този кортеж получаваме, като заместим в L стойностите на съответните атрибути и извършим съответните операции. При това, в резултата могат да се съдържат повтарящи се кортежи, дори в първоначалната релация да няма дубликати. Ще разгледаме примери.
Нека R е следната релация:

A	B	C
0	1	2
0	1	2
3	4	5

Тогава релацията $\pi_{A,\ B+C \to X}$ (R) има следния вид:

A	X
0	3
0	3
3	9

Релацията $\pi_{B\ -\ A \to X,\ C\ -\ B \to Y}$ (R) има следния вид:

X	Y
1	1
1	1
1	1

Операция за сортиране

В много случаи при извеждане на кортежите на една релация е удобно те да бъдат сортирани. Нека R е релация. **Операцията за сортиране** бележим по следния начин: τ_L (R). L е списък от атрибути на R, спрямо който се осъществява сортирането. Резултатната релация има същата схема като R и същите кортежи като R, но сортирани по атрибутите в L, т.е. първо по най-левия атрибут в L, при равенство по следващия атрибут в L и т.н. Операцията τ е единствената разглеждана операция, която като резултат дава списък. Затова има смисъл τ да се прилага в края на серия от операции, тъй като операциите в релационната алгебра се прилагат върху множества и мултимножества, а не върху списъци. Ако след τ се изпълни друга операция на релационната алгебра, то резултатът от τ се третира като множество (мултимножество).
В някои случаи, обаче, е по-ефективно да се изпълни дадена операция върху сортирана релация, така че понякога с цел ефективност е смислено да използваме τ като междинна операция.

Външно съединение

Едно свойство на операцията съединение е, че е възможно някои от кортежите на една релация да са висящи, т.е. да не успеят да се съединят с никой кортеж от другата релация. Информацията за висящите кортежи се губи в резултата, което в някои случаи не е желателно. Поради тази причина се въвежда **външно съединение**.
Нека R, S са релации.
Външно естествено съединение на R и S се означава така: $R \ ^{\circ}\!\bowtie S$.
Резултатната релация се получава по следния начин: изчисляваме $R \bowtie S$ и добавяме висящите кортежи на R и на S, като запълваме с NULL-стойности (⊥) всички атрибути, които не са техни, но присъстват в резултата от съединението.

Ляво външно естествено съединение на R и S се означава по следния начин: R $^{\circ}_{L}\bowtie$ S. То е аналогично на обикновеното външно съединение, но в резултатната релация се добавят само висящите кортежи на R (допълнени с NULL-стойности).
Дясно външно естествено съединение на R и S се означава по следния начин: R $^{\circ}_{R}\bowtie$ S. То е аналогично на обикновеното външно съединение, но в резултатната релация се добавят само висящите кортежи на S (допълнени с NULL-стойности).
Външно тита-съединение на R и S се означава така: R $^{\circ}\bowtie_{C}$ S.
Тук C е условен израз, в който могат да участват атрибути на R и на S.
Резултатната релация се получаваме по следния начин:
изчисляваме R $\bowtie_{C}$ S и добавяме висящите кортежи на R и на S, като запълваме с NULL-стойности всички атрибути, които не са техни, но присъстват в резултата от съединението.
Ляво външно тита-съединение на R и S се означава
по следния начин: R $^{\circ}_{L}\bowtie_{C}$ S. То е аналогично на обикновеното външно тита-съединение, но в резултатната релация се добавят само висящите кортежи на R (допълнени с NULL-стойности).
Дясно външно тита-съединение на R и S се означава по следния начин: R $^{\circ}_{R}\bowtie_{C}$ S. То е аналогично на обикновеното външно тита-съединение, но в резултатната релация се добавят само висящите кортежи на S (допълнени с NULL-стойности).
Ще разгледаме примери.
Нека U е следната релация:

A	B	C
1	2	3
4	5	6
7	8	9

Нека V е следната релация:

B	C	D
2	3	10
2	3	11
6	7	12

Тогава U $^{\circ}\bowtie$ V, U $^{\circ}_{L}\bowtie$ V, U $^{\circ}_{R}\bowtie$ V имат съответно следния вид:

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	$\perp$
7	8	9	$\perp$
$\perp$	6	7	12

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	$\perp$
7	8	9	$\perp$

A	B	C	D
1	2	3	10
1	2	3	11
$\perp$	6	7	12

Релацията $U \circ\bowtie_{A>V.C} V$ изглежда така:

A	U.B	U.C	V.B	V.C	D
4	5	6	2	3	10
4	5	6	2	3	11
7	8	9	2	3	10
7	8	9	2	3	11
1	2	3	⊥	⊥	⊥
⊥	⊥	⊥	6	7	12

Естествено, $U \circ_L\bowtie_{A>V.C} V$ се получава като премахнем последния ред на тази релация, а $U \circ_R\bowtie_{A>V.C} V$ се получава като премахнем нейният предпоследен ред.

Ограничения върху релациите

Релационната алгебра предоставя начин, по който да се изразяват ограничения върху релациите - например, ограничения за референтна цялосност, функционални зависимости и др.

Ограниченията в релационната алгебра се изразяват по два начина:

1. Ако R е израз от релационната алгебра, то $R = \emptyset$ е ограничението, което казва, че няма кортежи в релацията, която е резултат от изчислението на R.
2. Ако R, S са изрази от релационната алгебра, то $R \subseteq S$ е ограничението, което казва, че всеки кортеж в релацията, която е резултат от изчислението на R е кортеж в релацията, която е резултат от изчислението на S.

Двата начина за изразяване на ограничения са еквивалентни.
Действително, ограничението $R \subseteq S$ може да бъде записано по следния еквивалентен начин: $R - S = \emptyset$.
От друга страна, ограничението $R = \emptyset$ може да бъде записано по следния еквивалентен начин: $R \subseteq \emptyset$. Формално погледнато, $\emptyset$ не е израз на релационната алгебра, така че трябва да използваме израз, чиято стойност е $\emptyset$, например R - R.
Еквивалентността на двете ограничения е в сила дори когато R, S са мултимножества, стига $R \subseteq S$ да се интепретира по следния начин: всеки кортеж се среща поне толкова пъти в S, колкото в R.

В SQL най-често се използва първият начин за изразяване на ограничения.

Ограничения за референтна цялостност

В релационния модел ограничение за референтна цялостност се дефинира по следния начин: ако v е стойност на кортеж в релация R, то тази стойност v трябва да присъства в кортеж на друга релация S.
Ще разгледаме примери. Нека разгледаме следните релации
Movie (title, year, length, inColor, studioName, producerC#)
MovieExec (name, address, cert#, netWorth).
При тези две релации можем да наложим следното ограничение за референтна цялостност: за всеки филм, producerC# трябва да е номер на продуцент, описан в таблицата MovieExec. То се изразява в релационната алгебра по следните еквивалентни начини:
$\pi_{producerC\#}$ (Movie) $\subseteq$ $\pi_{cert\#}$ (MovieExec),
$\pi_{producerC\#}$ (Movie) - $\pi_{cert\#}$ (MovieExec) = $\varnothing$.

Нека разгледаме и релацията
StarsIn (movieTitle, movieYear, starName).
Можем да наложим следното ограничение за референтна цялостност: всеки филм, който участва в StarsIn трябва да е описан в Movie.
Да напомним, че филмите се идентифицират и с двата атрибута заглавие, година. Ограничението се изразява в релационната алгебра по следните еквивалентни начини:
$\pi_{movieTitle,\ movieYear}$ (StarsIn) $\subseteq$ $\pi_{title,\ year}$ (Movie),
$\pi_{movieTitle,\ movieYear}$ (StarsIn) - $\pi_{title,\ year}$ (Movie) = $\varnothing$.

Други ограничения

Всяка функционална зависимост може да се изрази като ограничение в релационната алгебра. Например, за релацията
MovieStar (name, address, gender, birthdate) да предположим, че е в сила функционалната зависимост name → address.
Тогава тя се изразява по следния начин в релационната алгебра:
$\sigma_{MS1.name = MS2.name\ AND\ MS1.address \neq MS2.address}$ (ρ_{MS1} (MS) x ρ_{MS2} (MS)) = $\varnothing$.
Да отбележим, че многозначните зависимости също се изразяват като ограничения в релационната алгебра.

Ограниченията по домен изискват стойността на даден атрибут да е от определен тип данни. Ако допустимите стойности на един атрибут са краен брой и те могат да се изразят на езика на условните изрази в релационната алгебра, то ограничението по домен за този атрибут се изразява като ограничение в релационната алгебра.
Ще разгледаме пример.
За релацията MovieStar имаме следното ограничение по домен: атрибутът gender може да приема само две стойности - 'M' или 'F'.
Това ограничение се изразява в релационната алгебра по следния начин:
$\sigma_{gender \neq 'F'\ AND\ gender \neq 'M'}$ (MovieStar) = $\varnothing$.

Има ограничения, които не попадат в разгледаните категории.
Това са общите ограничения. Релационната алгебра предоставя възможност за изразяване на най-различни общи ограничения върху базата от данни. Като пример да разгледаме двете релации

MovieExec (name, address, cert#, netWorth)
Studio (name, address, presC#).
Поставяме общо ограничение президент на студио да притежава поне 10000000$. То се изразява по следните еквивалентни начини:
$\sigma_{netWorth<10000000}$ (Studio $\bowtie_{presC\# = cert\#}$ MovieExec) = $\varnothing$;
$\pi_{presC\#}$ (Studio) $\subseteq$ $\pi_{cert\#}$ ($\sigma_{netWorth\geq10000000}$ (MovieExec)).

Други модели на данните

Моделът същност-връзки и релационният модел са само два от моделите на данни, които се използват в съвременните бази от данни.

Първо ще разгледаме обектно-ориентираният модел на данните. Един начин за въвеждане на обектно-ориентираните концепции в базите от данни е да се разширят обектно-ориентираните езици, като C++ и JAVA в посока устойчивост на данните. При програмиране на тези езици се предполага, че обектите изчезват след приключване на програмата, докато при базите от данни се изисква обектите да се съхраняват произволно дълго, докато потребителят сам не реши да ги да мофицира или изтрие. Ще разгледаме чист обектно-ориентиран модел на данните, който се нарича **ODL** (object definition language) и е разработен от ODMG (object data management group).

По-нататък ще разгледаме обектно-релационният модел на данните. Този модел е част от най-съвременния SQL стандарт и се нарича **SQL99**. При него релационният модел е разширен с обектно-ориентираните концепции.

След това ще разгледаме модел на **полуструктурираните данни**.
Този модел решава редица проблеми при базите от данни, свързани с интеграция на информацията, предоставена от различни източници. Моделът предоставя по-голяма гъвкавост при структуриране на схемата на една база от данни. Реализацията на тази концепция е езикът **XML** (extensible mark-up language). Основното предназначение на XML е да представя документите като набор от вгнездени елементи от данни.
В наше време се счита, че данните, представени с XML са най-добри за обмяна между различни приложения. В бъдеще се предполага, че XML ще може да се използва и за съхранение на данни.

Концепции в обектно-ориентирания подход

Обектно-ориентираното програмиране е инструмент за по-добра организация и по-надеждна реализация на програмите. Първият обектно-ориентиран език за програмиране е Smalltalk и неговите концепции се пренасят в езика C++, който вече е обектно-ориентиран (за разлика от C). В наши дни езикът JAVA, който поддържа значително по-добра преносимост на програмите от C++ също е обектно-ориентиран. Обектно-ориентираните идеи указват голямо влияние при проектиране на базите от данни.
Основните концепции при обектно-ориентирания подход са следните:

1. Мощна система за **типизация** – всяка система, която използва обектно-ориентирания подход трябва да може да създава нови типове.
2. **Класове** - те всъщност са типове. Всеки клас е асоцииран с разширение, което представлява множество от **обекти**, принадлежащи към класа. Една важна особеност на класовете е, че

към тях могат да се дефинират **методи**, приложими към обектите на класа.
3. Идентификация на обект - всеки обект трябва да се идентифицира уникално независимо от неговото съдържание.
4. **Наследственост** - класовете се организират в йерархии, като всеки клас наследява свойствата на по-горните класове в йерархията.

Система за типизация

Всяка система за типизация предоставя **атомарни типове** - като цяло число, реално число, булев тип, символен низ и средства за създаване на нови типове - **конструктори**. Конструкторите са приложими както към атомарни, така и към новосъздадени типове. Обикновено се поддържат следните конструктори:

1. Конструктор за **структури от записи** - по дадени типове $T_1, T_2, \ldots, T_n$ и по съответни имена на полета $f_1, f_2, \ldots, f_n$ се конструира нов тип запис, който се състои от n компоненти. В този тип i-тата компонента е от тип T_i и достъп до нея се осъществява чрез нейното името f_i, i = 1, 2, …, n. Структурите записи са точно тези типове, които в C и C++ се дефинират със “struct”.
2. Конструктори за **тип набор** - по даден тип T се конструира нов тип, който е набор от този тип. Различните езици за програмиране използват различни набори, но най-често са срещаните набори са масиви, списъци и множества. Например, ако T е атомарният тип цяло число, то чрез конструкторите за тип набор може да създадем масив от цели числа.
3. Конструктор на **референтен тип** (reference) - по даден тип T се конструира нов тип, чиито стойности са подходящи за намиране на стойност от тип T. В C и C++ това са указатели към стойности, т.е. виртуални адреси на стойности.

Естествено, конструкторите могат да се прилагат последователно и по този начин се създават типове с произволна сложност.

Класове и обекти

Един клас се състои от тип и една или повече функции, наречени методи, които могат да се изпълняват върху обектите от този клас. Обектите биват два вида - **неизменяеми** и **изменяеми**. Неизменяемите обекти са стойности от тип този клас – например, { 1, 2, 5} е неизменяем обект от тип множество от цели числа. Изменяемите обекти са променливи от тип този клас и тяхното съдържание може да се променя.

Идентификация на обектите

Всеки обект има идентификатор **OID** (object identifier). Не е възможно два различни обекта да имат едно и също OID, нито пък един обект да има два различни OID. За разлика от модела ER, където всяка същност трябва да има ключ, който е уникален, в обектно-ориентирания модел

може да има два различни обекта с еднакво съдържание - те ще се отличават по OID.

Методи

Във всеки клас има асоциирани функции, наречени методи. Всеки метод има поне един аргумент, който е обект на класа. Методът може да има и други аргументи - обекти от други класове или от същия клас.
Например, клас множество от цели числа може да съдържа съдържа методи за сумиране на елементите на множество, за обединение на две множества, за проверка за празнота на множество и др.
В някои ситуации класовете се наричат абстрактни типове данни.
Това означава, че данните са капсулирани, т.е. достъпът до обектите на класа е ограничен - единствено методите на класа са тези, които могат да изменят обектите на този клас. Капсулацията гарантира, че обектите на класа могат да бъдат изменяни само по начин, предвиден от създателя на класа. Поради тази причина, капсулацията е ключ към създаване на надежден софтуер.

Йерархии от класове

Възможно е да декларираме един клас като подклас на друг. Тогава подкласът наследява всички свойства на суперкласа - типът и методите. Освен това, в подкласа могат да се добавят нови методи и да се разшири типа на класа.
Ще разгледаме пример.
Можем да опишем неформално клас Account по следния начин:

```
CLASS Account = { accountNo : integer;
                  balance : real;
                  owner : REF Customer;
                }
```

С други думи, типът на Account е структура с три полета - номер на сметка, баланс на сметката и собственик на сметката, който е reference към обект от клас Customer (няма да го дефинираме).
Можем да дефинираме методи на класа Account, например:
deposit (a : Account, m : real) - метод за внасяне на пари в сметка,
withdraw (a : Account, m : real) - метод за теглене на пари от сметка.
Също можем да дефинираме подклас на Account - например, клас TimeDeposit, който има допълнително поле dueDate - датата, в която собственикът може да изтегли парите. Към този клас може да има допълнителен метод penalty (a : TimeDeposit), който изчислява глоба за предсрочно теглене като функция на dueDate и текущата дата.

Въведение в ODL

ODL е език за представяне на структурата на базите от данни в обектно-ориентиран стил. Той е разширение на IDL (interface description language), който е част от CORBA (common object resource broker architecture) - архитектура за разпределени изчисления върху множество

компютри. Обектите в CORBA са машинно независими и тяхното описание се осъществява на IDL.

При обектно-ориентираното проектиране, светът е съставен от обекти. От концептуална гледна точка обектите са аналогични на същностите в ER-модела, въпреки че с тях могат да се асоциират методи.
Обектите трябва да притежават OID, за да могат да се идентифицират.
При организация на информацията обектите с подобни свойства се групират в класове. В контекста на ODL групирането на обектите се извършва по следните два критерия:

1. Концепциите от реалния свят, които се представят чрез обектите трябва да са подобни - например, клиентите на дадена банка могат да бъдат групирани в един клас.
2. Свойствата на обектите трябва да са едни и същи. Много често обектите се разглеждат като записи, съставени от полета.
   Във всяко поле се записва стойност или reference към друг обект.

В ODL свойствата на обектите са три вида:

- **атрибути** - те представят стойности, свързани с обекта;
- **връзки** - те свързват обекта с един или няколко обекта от други класове;
- **методи** - това са функции, които могат да се изпълняват върху обекта.

Декларация на клас в ODL

Декларацията на клас в ODL има следния синтаксис:

```
class име_на_класа {
        списък_от_свойства
} ;
```

Свойствата са атрибути, връзки или методи, записани в произволен ред. За разделител в списъка се използва **;**.

Атрибути в ODL

Най-простите свойства са атрибутите. Те описват някакъв аспект на обекта, като асоциират с него стойност от фиксиран тип. За разлика от модела ER, не е задължително стойностите на атрибутите да са от атомарни типове. Например, клас, който описва хора може да съдържа атрибут дата на раждане, който е от тип структура с три полета - ден, месец, година. Ще разгледаме примери.

```
class Movie {
        attribute string title;
        attribute integer year;
        attribute integer length;
        attribute enum Film { color, blackAndWhite} type;
 } ;


class Star {
```

```
        attribute string name;
        attribute struct Addr
                        { string street,  string city} address;
} ;
```

Чрез ключовата дума enum в ODL се дефинира изброен тип - задават се явно допустимите стойности в този тип.
Чрез ключовата дума struct в ODL се дефинира структура, подобно на C. Изброеният тип в класа Movie и структурата в класа Star имат имена, въпреки че това изглежда излишно. Това позволява тези типове да се използват извън класа чрез операцията :: за разрешаване на достъп. Например, в клас Camera можем да използваме атрибут uses, дефиниран по следния начин: attribute Movie::Film uses;.

Връзки в ODL

Както вече споменахме, връзките са свойства на обектите, чрез които един обект се свързва с един или повече обекти от друг клас или от същия клас. Например, ако искаме да свържем всеки филм, който е обект от клас Movie с множеството от звездите, участващи в него, които са обекти от клас Star в декларацията на Movie трябва да добавим следния ред (на произволно място): relationship Set<Star> stars;.
По този начин с всеки обект от клас Movie се свързва с множество от reference към обекти от клас Stars. Името на това множество е stars. По-нататък, ако искаме да свържем всяка звезда, която е обект от клас Star с множеството от филмите - обекти от клас Movie, в които тя участва в декларацията на Star трябва да добавим следния ред: relationship Set<Movie> starredIn;.
По този начин, обаче, изпускаме една важна особеност на връзките. Връзката между звездите и филмите е двупосочна, което означава, че ако една звезда S участва във филм M, то S трябва да присъства в множеството за M и M трябва да присъства в множеството за S. Такава двупосочност се предполага в ER модела и в релационния модел. За да означим, че връзките starredIn от класа Star и star от класа Movie са свързани по този начин, трябва да добавим в декларациите и на двете ключовата дума inverse, последвана от името на обратната връзка. В класа Movie промяната изглежа по следния начин:
relationship Set<Star> stars inverse Star::starredIn;.
В класа Star промяната изглежда по следния начин:
relationship Set<Movie> starredIn inverse Movie::stars;.
Ще отбележим, че имената на обратните връзките се предшестват от името на класа, в който са декларирани и операцията за разрешаване на достъп. Естествено, това не е необходимо, ако връзката се осъществява между обекти от един и същи клас.

Като пример ще дефинираме класовете Movie, Star, Studio.

```
class Movie {
        attribute string title;
        attribute integer year;
        attribute integer length;
        attribute enum Film { color, blackAndWhite} type;
        relationship Set<Star> stars inverse Star::starredIn;
        relationship Studio ownedBy inverse Studio::owns;
} ;
class Star {
        attribute string name;
        attribute struct Addr { string street, string city } address;
        relationship Set<Movie> starredIn inverse Movie::stars;
} ;
class Studio {
        attribute string name;
        attribute Star::Addr address;
        relationship Set<Movie> owns inverse Movie::ownedBy;
} ;
```

Да отбележим, че всеки филм се притежава от единствено студио и затова ключовата дума Set не е използвана при задаване на типа на връзката ownedBy. Ще разгледаме по-подробно този момент.

Мултипликативност на връзките

Подобно на ER модела, връзките в ODL могат да бъдат класифицирани в следните групи - връзки много към много, връзки много към един, връзки един към един. Различните типове връзки в ODL се отличават по декларациите им:

1. При връзка много към много между класовете C и D, типът на връзката в класа C е Set<D>, а типът на връзката в класа D е Set<C>. Например, такава е връзката между Movie и Star.
2. При връзка много към един от класа C към класа D, типът на връзката в класа C е просто D, а типът на връзката в класа D е Set<C>. Например, такава е връзката между Movie и Studio.
3. При връзка един към един между класовете C и D, типът на връзката в класа C е просто D, а типът на връзката в класа D е просто C.

Всъщност, наборният конструктор Set може да бъде заменен с друг наборен конструктор - например Bag или List. Ще разгледаме наборните конструктори по-нататък.

Методи в ODL

Третото свойство в ODL са методите. Подобно на всички езици за обектно-ориентирано програмиране, методите на един клас са изпълним код, който може да се прилага към обектите на този клас.
В ODL могат да се декларират имената на методите, асоциирани с класа, типът на резултата на методите и типовете на входно-изходните

параметри на методите. Такива декларации в ODL се наричат **сигнатури**. Самият код, т.е. дефиницията на метода се програмира на базовия език, а не в ODL. Причината е, че проверката за коректност в една база от данни достига само до ниво проверка на типовете. Сигнатурите на методи на даден клас се записват заедно с декларациите на атрибутите и връзките в тялото на този клас.
Обектът, за който се извиква един метод е скрит параметър за този метод. Уникалност на имената на методите се изисква само в рамките на класа, т.е. възможно е два различни класа да притежават методи с еднакви имена.
Синтаксисът на декларацията на методите е подобен на синтаксиса на декларацията на функции в C със следните особености:

1. Всеки параметър на метода се предшества от ключова дума **in**, **out** или **inout**, което означава, че той се използва като входен, изходен или входно-изходен параметър. Методът може да модифицира изходните и входно-изходните параметри, но не може да модифицира входните. Изходните и входно-изходните параметри се предават по reference, а входните параметри се предават по стойност. Такава функционалност задължително трябва да се поддържа от базовия език.
2. Методите могат да възбуждат **изключения** (**exceptions**). Това са специални ситуации, които не са част от механизма за предаване на параметрите или връщане на резултата, чрез който методите комуникират. Изключението означава, настъпване на ненормално събитие. Обработката на изключенията не е част от стандарта на ODL. В декларацията на един метод, обаче, можем да да използваме ключовата дума **raises**, след която описваме изключенията, които методът може да възбуди.

Ще разгледаме няколко примера. В класа Movie можем да добавим на произволно място декларацията на следните три метода:
float lengthInHours () raises (noLengthFound);
void starName (out Set<String>);
void otherMovie (in Star, out Set<Movie>) raises (noSuchStar).

Първият метод lengthInHours би могъл да връща като резултат дължината на филма-обект, за който е извикан в часове. Този метод може да възбуди изключение noLengthFound, ако дължината на филма не е дефинирана (има стойност NULL).
Вторият метод starName има едни изходен параметър множество от низове. В този параметър методът би могъл да връща множеството от имената на звездите, които участват във филма-обект, за който този метод е извикан.
Третият метод otherMovie има един входен параметър - звезда-обект от клас Star и един изходен параметър множество от филми-обекти от клас Movie. Също така, методът може да възбуди изключение noSuchStar. Този метод би могъл да действа по следния начин. Първо се проверява дали звездата участва във филма и ако това не е така методът възбужда изключението noSuchStar. В противен случай, методът връща

множеството от всички други филми-обекти от клас Movie, в които участва звездата.

Типове в ODL

Системата за типизация в ODL е подобна на тази в C++. Както всяка система за типизация тя се изгражда от базови типове и правила по които се строят по-сложни типове от по-прости.
Базовите типове в ODL са следните:

1. Атомарни типове - integer (цяло число), float (реално число), char (символ), string (символен низ), boolean (булев тип), enum (изброен тип). Изброеният тип е краен списък от имена на абстрактни стойности - например, типът Film в класа Movie.
2. Имена на класове - например, Movie и Star.

Тези базови типове се комбинират в по-сложни типове с помощта на следните конструктори:

1. Конструктор **Set**. Ако T е тип, то типът Set<T> има за стойности крайни множества от елементи от тип T.
2. Конструктор **Bag**. Ако T е тип, то типът Bag<T> има за стойности крайни мултимножества от елементи от тип T. Да напомним, че за разлика от множеството, в мултимножеството един елемент може да присъства повече от един път.
3. Конструктор **List**. Ако T е тип, то типът List<T> има за стойности крайни списъци от елементи от тип T. Да напомним, че за разлика от множествата и мултимножествата в списъка има наредба на елементите.
4. Конструктор **Array**. Ако T е тип, n е естествено число, то типът Array<T, n> има за стойности масиви с дължина n с елементи от тип T.
5. Конструктор **Dictionary**. Ако T и S са типове, то типът Dictionary<T, S> има за стойности крайни множества от наредени двойки с първи елемент от тип T и втори елемент от тип S. Типът T се нарича **ключов тип**, а типът S се нарича **област от стойности**. Изисква се да няма две наредени двойки с един и същи ключ. Предполага се, че този тип се реализира по такъв начин, че да се осигурява ефективно търсене по ключ.
6. Конструктор **Struct**. Ако T_1, T_2, …, T_n са типове и f_1, f_2, …, f_n са имена, типът Struct { T1 f_1, T_2 f_2, …, T_n f_n } има за стойности структури от n компоненти, i-тата компонента е достъпна чрез името си f_i и е от тип T_i, i = 1, 2, …, n. Пример е типът Addr, който се използва в класа Star.

Първите пет конструктори са наричат **наборни конструктори**, последният конструктор се нарича **конструктор на структура**.

По отношение на типовете, които могат да се използват в декларацията на класовете трябва да отбележим следното:

1. Върху типовете на атрибутите няма никакви ограничения.

2. Типът на една връзка трябва да е име на клас или наборен конструктор, приложен еднократно към име на клас.

Ще дадем примери за допустими типове.
integer
Struct N { string field1, string field2 }
List<real>
Array<Struct N { string field1, integer field2 }, 10>
Следните типове не са допустими за тип на връзка.
Struct N { Movie field1, Star field2 }
Set<integer>
Set<Array<Star, 10>>

Представяне на небинарни връзки в ODL

В ODL се поддържат само бинарни връзки. За да представим небинарна връзка в ODL трябва да разбием тази връзка на бинарни връзки, аналогично на конструкцията, която разгледахме по-горе в ER модела. Нека имаме небинарна връзка R между класовете C_1, C_2, …, C_n.
За да представим R образуваме нов клас C и връзки много към един от C към C_i, i = 1, 2, …, n. Всеки обект t от класа C е конкретна връзка между обектите от C_1, C_2, …, C_n, с които е свързан t.

Като пример ще представим връзката Contracts между филмите, звездите и студията. За целта създаваме клас Contract със следната декларация:

```
class Contract {
        attribute integer salary;
        relationship Movie theMovie inverse Movie::contractsFor;
        relationship Star theStar inverse Star::contractsFor;
        relationship Studio theStudio inverse Studio::contractsFor
} ;
```

Използваме един атрибут salary - това е атрибутът на връзката. Съответно, в класовете Movie, Star, Studio трябва да добавим следните декларации.
В Movie:
relationship Set<Contract> contractsFor inverse Contract::theMovie;.
В Star:
relationship Set<Contract> contractsFor inverse Contract::theStar;.
В Studio:
relationship Set<Contract> contractsFor inverse Contract::theStudio;.

Наследяване и подкласове в ODL

Подобно на модела ER, където могат да се строят isa-йерархии, в ODL се допуска един клас C да наследи друг клас D. За целта, в декларацията на C след неговото име се записва ключовата дума **extends**, последвана от името на класа D.

Ще разгледаме примера с Cartoons и MurderMysteries.
Подкласът Cartoon трябва да наследи класа Movie и към него да се добави допълнителна връзка voices с класа Star, указваща звездите, които озвучават анимациония филм.
Подкласът MurderMystery трябва да наследи класа Movie и към него да се добави допълнителен атрибут weapon, указващ оръжието, използвано в убийството.

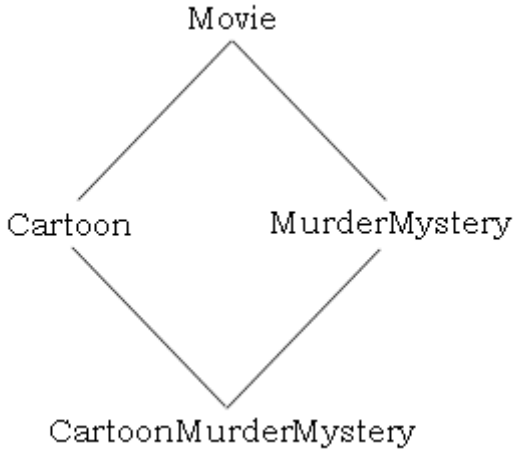
```
class Cartoon extends Movie {
        relationship Set<Star> voices inverse Star::voiceOf;
} ;
class MurderMystery extends Movie {
        attribute string weapon;
} ;
```

Естествено, в класа Star трябва да добавим следната декларация:
relationship Set<Cartoon> voiceOf inverse Cartoon::voices;.

Всички свойства на суперкласа се наследяват от подкласа - неговите атрибути, връзки, методи.

Множествено наследяване в ODL

Понякога се налага един клас да наследи два или повече класа. Например, може да има филми, които са едновременно анимационни и в които има убийства. В модела ER такива филми могат да се включат едновременно в Cartoons и MurderMysteries. При обектно-ориентирания подход, обаче, основен принцип е всеки обект да принадлежи на точно определен клас. Поради тази причина се нуждаем от нов клас, който представя указаните филми. Класът CartoonMurderMystery трябва да наследи двата класа Cartoon и MurderMystery. По този начин получаваме следната йерархия на наследяване:



При множественото наследяване в ODL, ключовата дума extends е последвана от имената на наследените класове, разделени с **:**.
В горния пример, декларацията на новия клас е следната:
class CartoonMurderMystery extends MurderMystery : Cartoon;.
В случая новият клас няма собствени свойства, а само наследени.

Технически, вторият и останалите наследени класове трябва да са **интерфейси**, т.е. класове, с които не се асоциират обекти.
По-нататък ще разгледаме по-подробно множеството от обекти, което се асоциира с един клас.

Когато един клас C наследява няколко класа, възниква опасност от дублиране на имената на свойства. Два или повече от суперкласовете на C може да имат свойства с еднакви имена.
В горния пример нямаше такъв проблем, но да разгледаме друг пример. Да предположим, че Movie има подкласове Romance и CourtRoom. Нека Romance има атрибут ending от изброен тип { happy, sad} и CourtRoom има атрибут със същото име ending от изброен тип { guilty, notGuilty }. Ако създадем нов подклас RomanceCourtRoom, който наследява двата класа Romance, CourtRoom, то атрибутът ending се дублира в двата суперкласа и неговият статут в новия клас е неясен. Решението на подобен конфликт не е част от стандарта на ODL.
Някои от подходите за решаване на проблема са следните:

1. Забранява се множественото наследяване - естествено, това е прекалено ограничаващ подход.
2. Изрично се указва, коя от декларациите на дублираните свойства да се използва.
3. Преименува се някое от дублираните свойства - например, в класа CourtRoom можем да преименуваме ending на verdict.

Разширения на класове

Множеството от обектите на даден клас се нарича **разширение на класа**.

Подобно на релационния модел, където се прави разлика между схемата на една релация и нейният екземпляр, в ODL се прави разлика между декларацията на един клас и неговото разширение.
Това се постига, като се зададат различни имена за класа и за неговото разширение. Името на разширението на един клас се задава в скоби след името на класа, предшествано от ключовата дума **extent**.
Общоприетата конвенция е имената на класа и на неговото разширение да са едни и същи, но името на класа да е в единствено число, а името на разширението да е в множествено число.
Например, за класа Movie може да се зададе име на разширението по следния начин:

```
class Movie (extent Movies) {
        attribute string title;
        …
} ;
```

Заявките към една база от данни, описана в ODL се извършват посредством имената на разширенията на класовете, а не посредством техните собствени имена.

Клас без зададено разширение наричаме **интерфейс**. С такъв клас не се асоциират обекти. Както вече споменахме, интерфейсите се използват при организиране на множествено наследяване в ODL.
Те са полезни когато няколко класа трябва да имат различни разширения, но едни и същи свойства. В такъв случай е достатъчно да се създаде един интерфейс I и всеки от класовете да наследи I.

Ключове в ODL

За разлика от моделите ER и релационния модел, в ODL ключовете не са задължителни. Причината е, че обектите се идентифицират уникално посредством своите OID. Напълно е възможно в ODL два обекта от един и същи клас да притежават еднакви свойства - системата ще ги различава по техните OID.

Въпреки това, в ODL могат да се дефинират ключове. Един или повече от атрибутите на даден клас могат да се декларират като ключ на класа с помощта на ключовата дума **key** или **keys**, последвана от заграден в скоби списък от имената на тези атрибути, разделени със запетаи. Самата декларация на ключа се поставя след името на разширението на този клас.
Ще разгледаме някои примери.
За класа Movie можем да зададем ключ по следния начин:

```
class Movie (extent Movies key (title, year))
{       attribute string title;
        …
} ;
```

Може да се използва коя да е от двете ключови думи key, keys при деклариране на ключ.
За класа Star можем да зададем ключ по следния начин:

```
class Star (extent Stars key name)
{       attribute string name;
        …
} ;
```

Възможно е да се зададе повече от един ключ за даден клас.
За целта, отделните ключове след key (keys) разделяме със запетаи.
Ще разгледаме пример, в който е подходящо използването на два ключа.

```
class Employee (extent Employees key empID, ssNo)
{ …
} ;
```

Тук empID е атрибут, чрез който се идентифицират работниците, атрибутът ssNo е техният номер на социална осигуровка.

Изрично отбелязваме, че тази декларация е различна от следната:

```
class Employee (extent Employees key (empID, ssNo))
{ …
} ;
```

При нея атрибутите empID, ssNo едновременно образуват ключ, т.е. възможно е, например, два обекта да имат еднакъв empID или ssNo. При горната декларация два обекта не могат да имат еднакви empID или еднакви ssNo.

Стандарта на ODL позволява в ключовете да се използват други свойства, а не само атрибути. Например, ако даден метод на клас се използва за негов ключ, това означава, че методът не може да връща еднакъв резултат за два различни обекта на класа.
Също е възможно в ключ на клас да се използва връзка.
Чрез използването на връзка един към много в ключ могат да се представят слабите множества същности. Като пример ще дефинираме клас Crew, който съответства на множеството същности Crews, което разгледахме по-горе при ER модела.

```
class Crew (extent Crews key (number, unitOf))
 {      attribute integer number;
        relationship Studio unitOf inverse Star::crewsOf;
 } ;
```

Естествено, в класа Star добавяме следната декларация:
relationship Set<Crew> crewsOf inverse Crew::unitOf;.
Ограничението, което поставя ключа е никои два екипа-обекти от клас Crew да нямат еднакви номера и да са свързани с еднакви студия. Това е същото ограничение, което произтичаше от факта, че Crews е слабо множество същности.

Преобразуване на ODL-проект към схема на релационна база данни

ODL е разработен като език за спецификация за обектно-ориентирани СУБД. Днешните СУБД са обектно-релационни, но ODL не губи значението си. Ще покажем как един ODL-проект може да се конвертира към схема на релационна база от данни. Този процес е подобен на преобразуването на ER-диаграма, но възникват следните проблеми:

1. Множествата същности задължително имат ключове за разлика от ODL-класовете.
2. Атрибутите в модела ER задължително са от атомарни типове, докато в ODL няма такова ограничение.
3. В ODL могат да се задават методи, но няма как те директно да се преобразуват към част от релационната схема. Засега ще считаме, че в ODL-класовете, които преобразуваме няма методи.

Преобразуване на атрибути

Най-общо всеки клас се преобразува към релация и всяко свойство на класа се преобразува към атрибут на тази релация. Това преобразуване може да се направи в случай, че всички свойства на класа са атрибути и всеки от тези атрибути има атомарен тип.

Ще разгледаме пример с опростен вариант на класа Movie.

```
class Movie (extent Movies) {
```

```
        attribute string title;
        attribute integer year;
        attribute integer length;
        attribute enum Film { color, blackAndWhite } type;
} ;
```
Този клас се преобразува към следната релация:
Movies (title, year, length, type).
Името на релацията съвпада с името на разширението на класа.
Всеки обект от разширението на класа се представя с един кортеж в екземпляра на съответната релация.

Преобразуване на атрибути от неатомарни типове

Дори свойствата на един клас да са само атрибути, може да възникнат проблеми при преобразуването на класа към релация по гореописания начин. Причината е, че в ODL атрибутите могат да са от неатомарни типове.

Най-простият случай е когато един атрибут има тип структура с полета от атомарни типове. Тогава за всяко поле на структурата в релацията добавяме по един атрибут. Единственият проблем, който може да възникне е дублиране на имена на атрибути. В такъв случай трябва да се извърши подходящо преименуване.

Ще разгледаме пример с класа Star.
```
class Star (extent Stars) {
        attribute string name;
        attribute struct Addr { string street, string city } address;
} ;
```
Този клас се преобразува към следната релация:
Stars (name, street, city).

Сега ще разгледаме как се преобразуват наборните конструктори Set, Bag, List, Array, Dictionary. Първо ще разгледаме по-подробно преобразуването на наборния конструктор Set.

Нека атрибутът A има тип множество от стойности. Един начин да преобразуваме този атрибут е следния: за всяка стойност от множеството на атрибута A да има отделен кортеж в екземпляра на съответната релация. При такова преобразуване схемата на релацията се образува както по-горе, т.е. атрибутът A се преобразува към атрибут на релацията без да се отчита, че A е от тип множество от стойности. Проблемът е, че такова преобразуване може да доведе до ненормализирани релации и се налага декомпозиране.
Ще разгледаме пример с разширен вариант на класа Star.
```
class Star (extent Stars) {
        attribute string name;
        attribute Set<struct Addr { string street, string city }> address;
        attribute Date birthdate;
} ;
```

Този клас се преобразува към следната релация:
Stars (name, street, city, birthdate).
За всеки отделен адрес на една звезда има отделен кортеж в екземпляра на релацията Stars. Поради тази причина name, street, city образуват ключ на тази релация. От друга страна, name е ключ на класа Star и тогава name → birthdate е нетривиална функционална зависимост с лява част, която не е суперключ. Това означава нарушаване на BCNF.

В общия случай при преобразуване на един атрибут от тип множество, заедно с няколко атрибути от атомарен тип, които не са част от ключ довеждат до нарушаване на BCNF. Два или повече атрибута от тип множество при преобразуване довеждат до нарушаване на 4NF.

Преобразуването на атрибут от тип Bag се извършва както при атрибут от тип Set. В чистия релационен модел, обаче, не се допуска дублиране на кортежи в релациите. Поради тази причина, трябва да добавим отделен атрибут count в релацията, който показва колко пъти съответният елемент се среща в мултимножеството. Например, ако в класа по-горе типът на атрибута address е
Bag<struct Addr { string street, string city }>,
то класът се преобразува към следната релация:
Stars (name, street, city, birthdate, count).
За всеки обект и всяка различна стойност от мултимножеството address на този обект има по един кортеж в релацията Stars. В този кортеж стойността на count е броят на срещанията на съответната стойност в мултимножеството address, съответно на обекта.

Преобразуване на атрибут от тип List се извършва подобно на атрибут от тип Set. За да представим наредбата между стойностите в списъка на този атрибут можем да добавим допълнителен атрибут position в релацията, който показва позицията на съответната стойност в списъка. Например, ако в класа по-горе типът на атрибута address е
List<struct Addr { string street, string city }>,
то класът се преобразува към следната релация:
Stars (name, street, city, birthdate, position).
За всеки обект и всяка стойност от списъка address на този обект има по един кортеж в релацията Stars. В този кортеж стойността на атрибута position е позицията на съответната стойност в списъка address, съответен на обекта.

Преобразуване на атрибут от тип Array може да се извърши по следния начин: всеки елемент на масива се представя чрез отделен атрибут в релацията. Например, ако в класа по-горе типът на атрибута address е
Array<struct Addr { string street, string city}, 3>,
то класът се преобразува към следната релация:
Stars (name, street1, city1, street2, city2, street3, city3, birthdate).
За всеки обект има по един кортеж в релацията Stars. В този кортеж стойностите на streetK, cityK, K = 1, 2, 3 са съответните стойности от масива address, съответен на обекта.

Преобразуване на атрибут от тип Dictionary може да се извърши по следния начин: за всяка стойност от речника на атрибута да има отделен кортеж в релацията. В схемата на релацията атрибутът се представя чрез два атрибута - един атрибут за ключа на речника и един атрибут за стойността, съответна на ключа. Например, ако в класа по-горе типът на атрибута address е
Dictionary<struct Addr { string street, string city}, string>,
то класът се преобразува към следната релация:
Stars (name, street, city, attr, birthdate).
За всеки обект и всяка стойност от речника address, съответен на този обект в релацията Stars има по един кортеж. В този кортеж стойностите на атрибутите street, city, attr се извличат от съответната стойност от речника address на обекта.

Преобразуване на ODL връзки

Както в модела ER за всяка връзка между два ODL-класа се образува по една релация, която свързва ключовете на двата класа. Естествено, двойките инверсни връзки се представят чрез една релация.

Като пример ще разгледаме следният вариант на
класовете Movie и Studio.

```
class Movie (extent Movies key (title, year)) {
        attribute string title;
        attribute integer year;
        attribute integer length;
        attribute enum Film { color, blackAndWhite} type;
        relationship Studio ownedBy inverse Studio::owns;
} ;
class Studio (extent Studios key name) {
        attribute string name;
        attribute string address;
        relationship Set<Movie> owns inverse Movie::ownedBy;
} ;
```

За представяне на връзката между Movie и Studio се създава нова релация StudioOf със следната схема: StudioOf (title, year, studioName).
В случая преобразуваната връзка е много към един от Movie към Studio. Поради тази причина релацията StudioOf може да се комбинира с релацията Movies, която съответства на класа Movie - този клас стои в множествената част на връзката. По този начин получаваме следната релация: Movies (title, year, length, type, studioName).

Както вече отбелязахме когато разглеждахме релационния модел, подобно комбиниране не е удачно да се извършва с класа в единичната част на връзка много към един или въобще за връзки много към много - това води до нарушаване на BCNF.

Липса на ключове в ODL

Тъй като ключовете не са задължителни в ODL можем да попаднем в ситуация, при която атрибутите на един клас не могат да се използват за идентифициране на обектите от този клас. Такава ситуация води до проблеми, особено ако класът участва в една или повече връзки с други класове. Решението на този проблем е добавяне на допълнителен атрибут, който да служи за ключ. Този атрибут се преобразува към отделен атрибут в релацията, съответна на класа и чрез него обектите на класа се представят в релациите, съответни на връзки, в които участва този клас. Например, да предположим, че атрибутът name не може да е ключ на класа Star. Тогава добавяме допълнителен атрибут cert#, който се свързва с всяка звезда и чрез който всяка звезда се идентифицира уникално. При това положение класът Star се преобразува към следната релация: Stars (cert#, name, street, city). Връзката много към много между класовете Star и Movie тогава се представя чрез следната релация: StarsIn (title, year, cert#).

Обектно-релационен модел на данните

В началото на 90-те години се появиха обектно-ориентирани СУБД.
Те не успяха да се наложат и отпаднаха към средата на 90-те.
Така преходът от релационни към обектно-ориентирани СУБД не се реализира, но обектно-ориентираният подход оказа голямо влияние върху релационните СУБД. В резултат на това се появи обектно-релационният модел. Този модел е в основата на един от най-съвременните стандарти на SQL - SQL99.

От релации към обектни релации

В обектно-релационния модел релацията отново е фундаментална концепция. За разлика от релационния модел, в обектно-релационния модел са добавени следните пет нови характеристики:

1. Въвеждат се **структурирани типове** за атрибутите - системата за типизация в обектно-релационните СУБД е подобна на тази в ODL. Атрибутите могат да са както от атомарни типове, така и от по-сложни типове, образувани чрез наборни конструктори и конструктори за структури. Интересен ефект от това е, че типът множество (мултимножество) от структури може да се разглежда като релация. По този начин един компонент на един кортеж може да бъде цяла релация.
2. Въвеждат се **методи**, които се обвързват с релациите и могат да се прилагат към кортежите на релациите.
3. Въвеждат се **идентификатори на кортежи** - в обектно-релационните системи кортежите играят ролята на обекти.
   В определени ситуации е полезно кортежите да се идентифицират уникално, подобно на обектите в ODL. По принцип идентификаторите на кортежи са невидими за потребителя, но понякога той може да има достъп до тях.

4. Въвеждат се **reference** към кортежи, които се използват по различни начини в обектно-релационните системи.

Вгнездени релации

Релации, които са разширени с първата характеристика от по-горе често се наричат **вгнездени релации**. В модел, допускащ вгнездени релации атрибутите могат да имат неатомарен тип. Релационната схема също се разглежда като възможен тип за атрибутите. Като резултат получаваме следната индуктивна дефиниция за тип на атрибут и схема на релация.
База: всеки атрибут може да има атомарен тип.
Стъпка: схемата на една релация се състои от имена за всеки един от атрибутите, като всеки атрибут може да има произволен тип, включително схема на релация.

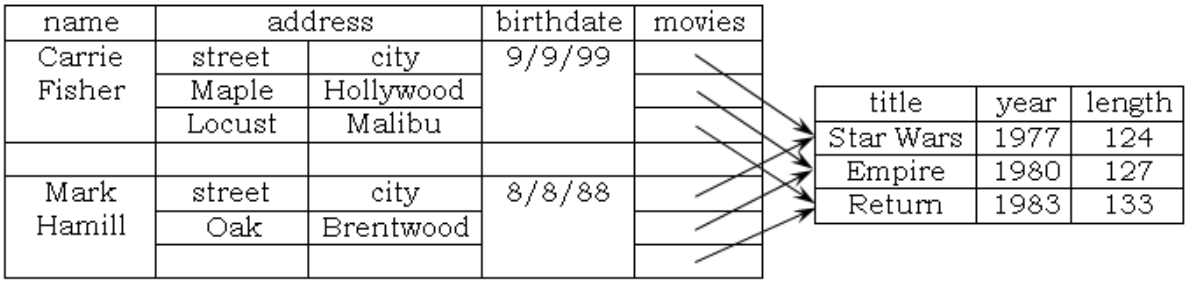
Когато разглеждахме релационния модел, не указвахме изрично атомарните типове на атрибутите в схемата на релациите, тъй като те съвсем не влияеха на релационните концепции. Тук ще продължим по същия начин, с тази разлика, че ако тип на атрибут е релационна схема, имената на атрибутите на тази схема се задават в скоби след името на атрибута. Тези атрибути от своя страна също могат да имат тип релационна схема и по този начин да получим вгнездена релационна схема на произволно ниски нива.
Като пример ще разгледаме вгнездена релация, подобна на релацията за звездите на филми от по-горе. Тя има следната схема:
Stars (name, address (street, city), birthdate, movies (title, year, length)).
Чрез атрибута movies, който има тип релационна схема се задават филмите, в които участва звездата. Чрез атрибута address, който има тип релационна схема се задават адресите на звездата.
Един примерен екземпляр на Stars изглежда по следния начин:

name	address		birthdate	movies		
Carrie Fisher	street	city	9/9/99	title	year	length
	Maple	Hollywood		Star Wars	1977	124
	Locust	Malibu		Empire	1980	127
				Return	1983	133
Mark Hamill	street	city	8/8/88	title	year	length
	Oak	Brentwood		Star Wars	1977	124
				Empire	1980	127
				Return	1983	133

В този екземпляр има два кортежа. Релациите, които са стойности на компонентите на кортежите, съответни на атрибута movie изглеждат по един и същи начин, но те са различни.

В този пример наблюдаваме излишество - информацията за филмите, които попадат в повече от една релация, която е стойност на атрибута movie се дублира. Този проблем се решава с въвеждане на **reference** към кортежи. Всеки атрибут може да има тип reference към кортеж с дадена схема или reference към множество от кортежи с дадена схема.
Ако атрибутът A има тип reference към кортеж със схема R, то в схемата, в която участва A отбелязваме това по следния начин: A (*R).
В ODL аналог на това понятие е връзка от тип R.
Ако атрибутът A има тип reference към множество от кортежи със схема R, то в схемата, в която участва A отбелязваме това по следния начин: A ({ *R}).
В ODL аналог на това понятие е връзка от тип Set<R>.

За да елиминираме излишеството в примера трябва да използваме две релации със следните схеми:
Movies (title, year, length)
Stars (name, address (street, city), birthdate, movies ({ *Movies})).
Преобразуваният екземпляр изглежда по следния начин:



По този начин елиминирахме излишеството в горната релация - за всеки филм има единствен кортеж, въпреки че към този кортеж може да има много reference.

Сравнение на обектно-ориентирания с обектно-релационния модел

Обектно-ориентираният модел, представен с ODL и обектно-релационният модел, който разглеждаме много си приличат. Ще коментираме някои от приликите и разликите между двата модела.

Обекти и кортежи

Стойността на обект всъщност е структура, съдържаща компоненти за атрибутите и връзките на обекта. В стандарта на ODL не е указано как се представят връзките, но можем да предполагаме, че те са реализирани чрез указатели. Кортежът също е структура, но при обикновените релации тя има компоненти само за атрибутите.
В чистия релационен модел връзките се представят чрез релации.

В обектно-релационния модел, обаче, връзките могат да се представят и чрез указатели към кортежи, както видяхме по-горе.

Разширения на класове и екземпляри на релации

В ODL всички обекти от даден клас съществуват в разширението на класа. В обектно-релационния модел може да има няколко релации с идентични схеми, но с различни екземпляри. В ODL това може да се постигне чрез наследяване на интерфейс, както видяхме по-горе.

Методи

По-горе не дискутирахме използването на методи като част от обектно-релационния модел. На практика в стандарта SQL99, както и при другите обектно-релационни системи позволяват аналогични на ODL възможности за дефиниране и използване на методи.

Система за типизация

Системите за типизация в ODL и в обектно-релационния модел са подобни. И двете системи поддържат атомарни типове и конструктори за създаване на нови типове. Конструкторите се различават в реализациите, но навсякъде се поддържат множества и мултимножества. Освен това, типът множество (мултимножество) от структури играе централна роля и в двата модела. В ODL това е типът на всеки клас, а в обектно-релационния модел това е типът на всяка релация.

Reference и идентификатори на обекти

В чистия обектно-ориентиран модел идентификаторите на обекти (OID) са скрити от потребителя и той няма достъп до тях.
В обектно-релационния модел reference към кортежи се задават като част от типа на атрибут и при определени условия те подлежат на манипулация. Тази възможност позволява значителна гъвкавост, но е сериозен източник на грешки.

Проблем за съвместимост

От коментара се вижда, че разликите между обектно-ориентирания и обектно-релационния модел са много малки. Въпреки това обектно-релационният модел надделя, тъй като обектно-ориентираните системи се появиха прекалено късно - след като релационните системи доминираха пазара през последните десет години. При въвеждането на обектно-релационни СУБД доставчиците гарантираха съвместимост с чисто релационните СУБД. От друга страна, миграцията към чисто обектно-ориентирано СУБД от релационно СУБД изисква пълно пренаписване на системата.

Преобразуване на ODL-проекти в обектно-релационни схеми

Проблемите, които се появиха при преобразуване на ODL-проект към чисто релационна схема произтичаха от това, че в ODL се допускат по-богати конструкции - атрибути от неатомарен тип, връзки, методи. С разширенията които добавихме в обектно-релационния модел, преобразуването на ODL-проект към обектно-релационна схема се улеснява значително. Неатомарните типове в ODL директно могат да се образуват към съответните им еквивалентни обектно-релационни типове. В някои случаи, обаче, конкретната обектно-релационна система може да не поддържа определен неатомарен тип и тогава се налага да се използва подходящо моделиране, подобно на това при преобразуване на ODL-проект към чисто релационна схема.
Връзките в обектно-релационния модел могат да се представят чрез релации, основавайки се на ключове, но за тяхното представяне могат да се използват и reference към кортежи.
И накрая, при преобразуване на ODL-проект към чисто релационна схема изключихме от разглежданията методите, но това ограничение отпада в обектно-релационния модел, където се поддържат методи.

Модел на полуструктурираните данни

Моделът на **полуструктурираните данни** има специална роля в СУБД.

1. Този модел е подходящ за интеграция на бази от данни, т.е. за описание на подобни данни, които се съдържат в две или повече бази от данни с различни схеми.
2. Моделът се използва като модел за документация с нотации от типа на XML, които се използват при поделяне на информацията чрез мрежа.

Моделът на полуструктурираните данни притежава по-голяма гъвкавост от досега разгледаните модели.

В модела ER има два фундаментални типове данни - множества същности и връзки. В релационния модел има само един тип данни - релацията. Моделът ER е по-богат на концепции от релационния модел и позволява по-ефикасно моделиране на реалния свят. От друга страна, фактът че има само една структура от данни в релационния модел позволява по-ефективна реализация на заявките.
В обектно-ориентирания модел има две концепции - класове и връзки между класовете. В обектно-релационния модел също има две концепции - типове на атрибути и релации.

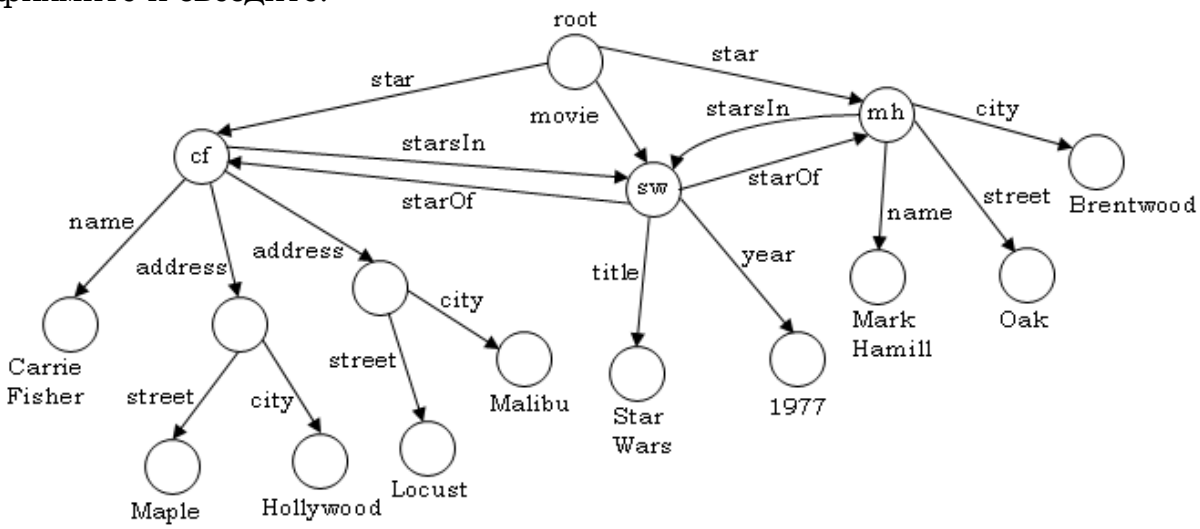
В модела на полуструктурираните данни концепциите класове и връзки от обектно-ориентирания модел се смесват, както в релационния модел се смесват концепциите множества същности и връзки от ER модела.
В релационния модел мотивацията за това е по-ефективната реализация, докато в модела на полуструктурираните данни чрез смесването на двете концепции се търси по-голяма гъвкавост.
Във всички разгледани досега модели за данните задължително предварително трябва да се дефинира схема. При модела на

полуструктурираните данните нямат схема. По-точно самите данни носят информация за схемата. Това позволява схемата да варира във времето и в рамките на една релация.

Представяне на полуструктурираните данни

Базата от полуструктурирани данни е набор от **възли**. Всеки възел е **вътрешен възел** или **листо**. Всеки лист е асоцииран с данни от атомарен тип. От всеки вътрешен възел излизат една или повече **дъги**. Всяка дъга има етикет, който описва по какъв начин възелът в главата на дъгата се свързва с възела в опашката на дъгата. Един от вътрешните възли се нарича **корен** и в него не влизат дъги. Коренът представя цялата база от данни. Всеки възел трябва да е достижим от корена, но не е задължително графът на базата от данни да е дърво.

Като пример ще разгледаме полуструктурирана база от данни за филмите и звездите.



Коренът (root) е входна точка в базата от данни. Основните обекти (същностите), които в случая са звездите и филмите се представят чрез преките наследници на корена. Листата имат етикети, които описват данните. Три от вътрешните възли имат етикети - cf, sw, mh.
Те представят съответно звездата Carrie Fisher, филмът Star Wars и звездата Mark Hamill. Етикетите на вътрешните възли не са част от модела, но придават по-голяма яснота в примера.

Етикетите на дъгите имат две роли. Да предположим, че една дъга, която излиза от възел N и влиза във възел M има етикет L.

1. Ако възелът N представя обект (структура), а възелът M представя атрибут на обекта (поле на структурата), то етикетът L е името на този атрибут (поле).
2. Ако възлите N и M представят обекти, то етикетът L е името на връзката от N към M.

В примера по-горе да разгледаме вътрешният възел cf, който представя звездата Carrie Fisher. От него излиза дъга с етикет name, която

представя атрибутът name на тази звезда. Също така, от cf излиза дъга с етикет starsIn към възела sw, който представя филмът Star Wars. Тази дъга представя връзката starsIn между звездата и филма, в който тя участва.

Интеграция на информацията чрез полуструктурирани данни

За разлика от другите модели, данните в полуструктурирания модел се самоописват, т.е. схемата на данните е прикачена към самите данни. Във всеки възел (освен корена) влизат дъги, чиито етикети описват ролята на възела по отношение на свързаните с него възли, посредством тези дъги. Във всички други модели данните имат фиксирана схема, която е отделена от тях. При тях ролята на данните се подразбира от тази схема.

Фиксирана схема на базата от данни се използва когато базата от данни е много голяма, тъй като с тази схема могат да се използват подходящи структури за по-ефективно търсене в данните. При малки бази от данни е за предпочитане да се използва полуструктурирания модел, т.е. самоописващи се данни. Такъв подход се използва при продукта Lotus Notes. При този продукт не е необходимо предварително да се задава схема на данните.

Едно от приложенията на гъвкавия полуструктуриран модел на данните е използването му при интеграция на информацията.
Базите данни в днешно време се използват масово. Често възниква необходимост две или повече бази от данни да бъдат достъпни като една. Например, при сливане на компании базите от данни за техния персонал трябва да се слеят. Ако тези бази от данни имат еднакви схеми, то обединяването ще е лесно. Такава ситуация, обаче, много рядко е налице. В общия случай независимо разработените бази от данни имат различни схеми, дори могат да използват различни модели на данните - една база от данни да е релационна, друга обектно-ориентирана. Нещата са още по-сложни. В течение на времето една база от данни се използва от все повече и повече приложения и в един момент става практически невъзможно тя да бъде модифицирана. Този проблем е известен като проблем на **наследените бази от данни**.
Едно възможно решение на проблема е чрез използване на модела на полуструктурираните данни. Мястото на този модел е в интерфейса, чрез който потребителят осъществява достъп до наследените бази от данни. Този интерфейс трябва да разполага с обвивка, която преобразува данните от източниците в полуструктурирани данни.
Възможно е в интерфейса въобще да не се включват полуструктурирани данни. В този случай потребителят извършва заявки към полуструктурирани данни, въпреки че тези данни не съществуват, а интерфейсът преобразува тези заявки към заявки за съответните бази от данни, съобразени с техните схеми.
XML и неговият модел на данни

XML (extensible mark-up language) е нотация за маркиране на документи чрез етикети, подобна на HTML или SGML. Документът не е нищо друго, освен символен файл. За разлика от HTML, където маркировката се използва най-вече за визуализация на данните в документа, маркировката в XML се използва за описване на смисъла на информацията в документа.
Ще разгледаме как се използва XML за представяне на графа на базата от полуструктурирани данни в линейна форма. По-конкретно, ролята на етикетите на дъгите ще се изпълнява от маркерите в XML-документа. След това ще въведем **DTD** (document type definition) - това е гъвкава схема на данните, която може да се прилага върху XML-документи.

Маркери в XML

Маркерите в XML са образувани от текст, ограден в триъгълни скоби по следния начин: <текст>. Маркерите винаги се използват по двойки - на всеки отварящ маркер <текст> съответства затварящ маркер </текст>, в който се записва същият текст, предшестван от /. В HTML, за разлика от XML, е възможно един отварящ маркер да няма съответен затварящ маркер. Маркировката задължително трябва да е вгнездена, т.е. ако между една двойка маркери има отварящ маркер, то съответният затварящ маркер трябва да присъства между същата двойка маркери. Например, не е възможна подобна маркировка:
<TAG1> … <TAG2> … </TAG1> …</TAG2>.

XML-документите се използват в два режима:

1. **Добре структуриран XML** - при него могат да се използват произволни маркери. Този режим е много близък до полуструктурираните данни, в смисъл че структурата на XML-документа не е предварително фиксирана.
2. **Валиден XML** - при него се използва DTD, където е описано какви маркери могат да се използват и как те могат да се вгнездяват. Този режим е нещо средно между полуструктурираните данни и данните с фиксирана схема. Схемата в DTD е доста по-гъвкава от схемите в релационния и обектно-ориентирания модел. Както ще видим по-нататък, DTD позволява изборни полета, липсващи полета и др.

Добре структурани XML-документи

Минималното изискване за един добре структуриран XML-документ е наличие на декларация в началото на документа, която показва че той е XML и наличие на **коренен маркер**, който обгражда цялата останала част от документа. Общата структура на добре структурираните XML-документи е следната:

<? XML VERSION = “1.0” STANDALONE = “yes” ?>
<BODY>
…

</BODY>

Първият ред индикира, че документът е XML. В него е записана версията на XML, която е използвана. Параметърът STANDALONE със стойност “yes” означава, че документът е добре структуриран, т.е. с него не е свързано DTD. Целият първи ред трябва да е ограден в специален маркер <? …?>, който е различен от обикновените маркери.
Името на коренния маркер в случая е BODY, но може да се използва кое да е друго име.

Ще опишем добре структуриран XML-документ, който представя грубо полуструктурираните данни за филмите и звездите.

```
<? XML VERSION = “1.0” STANDALONE = “yes” >
<STAR-MOVIE-DATA>
    <STAR><NAME>Carrie Fisher</NAME>
        <ADDRESS><STREET>123 Maple Str.</STREET>
            <CITY>Hollywood</CITY></ADDRESS>
        <ADDRESS><STREET>5 Locust Ln.</STREET>
            <CITY>Malibu</CITY></ADDRESS>
    </STAR>
    <STAR><NAME>Mark Hamill</NAME>
        <STREET>456 Oak Rd.</STREET><CITY>Brentwood</CITY>
    </STAR>
    <MOVIE><TITLE>Star Wars</TITLE><YEAR>1977</YEAR>
    </MOVIE>
</STAR-MOVIE-DATA>
```

В този XML-документ не са представени връзките между звездите и филмите. Една възможност е да съхраняваме информация за всички филми, които играе дадена звезда в нейната съответна секция. Например, за звездата Mark Hamill това изглежда по следния начин:

```
    <STAR><NAME>Mark Hamill</NAME>
        <STREET>456 Oak Rd.</STREET><CITY>Brentwood</CITY>
        <MOVIE><TITLE>Star Wars</TITLE><YEAR>1977</YEAR>
        </MOVIE>
        <MOVIE><TITLE>Empire Strikes Back</TITLE>
            <YEAR>1980</YEAR></MOVIE>
    </STAR>
```

Този подход, обаче, води до излишество, тъй като цялата информация за всеки филм се повтаря по един път за всяка звезда, която участва във филма. Всъщност, в конкретния случай филмите се идентифицират със заглавие и година, но е възможно те да имат други атрибути.

Валидни XML-документи

За да може XML-документите да се обработват автоматично, нормално е те да се подчиняват на някаква схема. С други думи, трябва да е указано какви маркери могат да се използват и по какви правила тези маркери могат да се вгнездяват. Описанието на схемата на XML-документ е последователност от граматични правила, която се нарича **DTD** (document type definition).
Естествено, наличието на такива описания води до по-лесна обмяна на информация между компании или общности, които използват XML-документи.
Най-общо едно DTD изглежда по следния начин:

```
<!DOCTYPE име_на_коренен_маркер [
        <!ELEMENT име_на_елемент (компоненти)>
        <!ELEMENT име_на_елемент (компоненти)>
        …
        <!ELEMENT име_на_елемент (компоненти)>
] >
```

Зададеното име на коренния маркер трябва да се използва във всеки XML-документ, който се подчинява на това DTD.
Всеки **елемент** се описва с име, което представлява името на маркера, който се използва за ограждане на порцията от XML-документа, съответна на този елемент. Освен това всеки елемент има компоненти, които се изброяват след името на елемента и се разделят със запетаи. Тези компоненти са имена на маркери, които може трябва да присъстват в порцията от XML-документа, съответна на елемента.
След малко ще видим как се налагат изисквания върху компонентите на един елемент.
Ако след името на елемент се използва списък от компоненти (#PCDATA), това означава, че в порцията на XML-документа, съответна на този елемент няма маркери, т.е. има само текст.
Да отбележим, че имената на маркерите в XML (както в HTML) са case-insensitive, т.е. не се прави разлика между малки и главни букви.

Ще разгледаме едно примерно DTD за звездите.

```
<!DOCTYPE Stars [
        <!ELEMENT STARS (STAR*)>
        <!ELEMENT STAR (NAME, ADDRESS+, MOVIES)>
        <!ELEMENT NAME (#PCDATA)>
        <!ELEMENT ADDRESS (STREET, CITY)>
        <!ELEMENT STREET (#PCDATA)>
        <!ELEMENT CITY (#PCDATA)>
        <!ELEMENT MOVIES (MOVIE*)>
        <!ELEMENT MOVIE (TITLE, YEAR)>
        <!ELEMENT TITLE (#PCDATA)>
        <!ELEMENT YEAR (#PCDATA)>
] >
```

В общия случай компонентите на един елемент с име E са имена на други елементи. Те трябва да присъстват между маркерите <E> и </E> в XML-документа и то в указания ред. С всеки компонент може да се използва един от следните оператори, който се записва след името на компонента:

- оператор * - той указва, че съответният компонент трябва да присъства нула или повече пъти в XML-документа;
- оператор + - той указва, че съответният компонент трябва да присъства един или повече пъти в XML-документа;
- оператор ? - той указва, че съответният компонент трябва да не присъства или да присъства един път в XML-документа.

В списъка от компоненти на един елемент може да се използва символът |, който се появява между компоненти или списъци от компоненти, оградени в скоби. Неговата семантика е на изключващо или - в XML-документа трябва да присъства или компонента (списъка от компоненти) отляво на | или компонента (списъка от компоненти) отдясно на |, но не и двете едновременно. Например, списък от компоненти (#PCDATA | (STREET, CITY)) за елемента ADDRESS означава, че в XML-документа порцията, съответна на този елемент е или текст без маркери или текст с два маркера със съответни имена.

Ще разгледаме един примерен XML-документ, който отговаря на описаното по-горе DTD.

```
<STARS>
	<STAR><NAME>Carrie Fisher</NAME>
		<ADDRESS><STREET>123 Maple Str.</STREET>
			<CITY>Hollywood</CITY></ADDRESS>
	<MOVIES><MOVIE><TITLE>Star Wars</TITLE>
		<YEAR>1977</YEAR></MOVIE>
		<MOVIE><TITLE>Empire Strikes Back</TITLE>
		<YEAR>1980</YEAR></MOVIE>
		<MOVIE><TITLE>Return of the Jedi</TITLE>
		<YEAR>1983</YEAR></MOVIE>
	</MOVIES>
	</STAR>
	<STAR><NAME>Mark Hamill</NAME>
		<ADDRESS><STREET>456 Oak Rd.</STREET>
			<CITY>Brentwood</CITY></ADDRESS>
		<MOVIES><MOVIE><TITLE>Star Wars</TITLE>
			<YEAR>1977</YEAR></MOVIE>
			<MOVIE><TITLE>Empire Strikes Back</TITLE>
			<YEAR>1980</YEAR></MOVIE>
		<MOVIE><TITLE>Return of the Jedi</TITLE>
			<YEAR>1983</YEAR></MOVIE>
	</MOVIES>
	</STAR>
</STARS>
```

Ако един XML-документ е валиден, т.е. свързан с DTD, това се указва по един от следните два начина:

1. Включваме пълното описанието на DTD в началото на XML-документа.
2. Указваме име на файл, който съдържа съответното DTD. В този случай, приложението което обработва XML-документа трябва да има достъп до файловата система, в която се съхранява DTD.

И в двата случая параметърът STANDALONE в първия ред на XML-документа има стойност “no”.
В първия случай пълното описание на DTD се осъществява непосредствено след първия ред на валидния XML-документ.
Във втория случай вторият ред на валидния XML-документ трябва да има следния вид:
<!DOCTYPE име_на_коренен_маркер SYSTEM “име_на_файл”>
Тук пълното описание на DTD е съдържание на съответния файл.
Този вариант е по-удобен, тъй като едно описание на DTD може да се използва с много XML-документи.

Например, описаният валиден XML-документ по-горе трябва да започва със следните два реда:
<?XML VERSION = “1.0” STANDALONE = “no” ?>
<!DOCTYPE Stars SYSTEM “star.dtd”>
Тук star.dtd е файлът, който съдържа описанието на DTD за звездите.

Атрибути в XML

XML-документите и полуструктурираните данни са доста силно свързани. Всеки XML-документ може да се представи чрез граф на полуструктурирани данни. За целта, за всяка двойка маркери <T> и </T> създаваме по един възел. Ако една двойка маркери <S> и </S> е директно вгнездена в двойката маркери <T> и </T> (т.е. няма двойка маркери, която огражда <S>, </S> и е оградена от <T>, </T>), то от възела, съответен на <T> и </T> прекарваме дъга с етикет S към възела, съответен на <S> и </S>.
Обратното не е вярно - получените графи от XML-документи са дървета, тъй като всеки връх има единствен предшественик.
Поради тази причина чрез досега описаните възможности на XML връзките не могат да се представят чрез двойки противоположни дъги.
Ще въведем още един елемент от XML, чрез който това става възможно - **атрибутите** на маркери.
Всеки отварящ маркер може да имат атрибути, за които се задават стойности след името на маркера. Атрибутите имат различни предназначения, но ние в конкретния случай ще ги използваме за представяне на връзки.
За нашите цели разглеждаме два възможни типове атрибути - **ID** и **IDREF** (**IDREFS**).
Ако един атрибут на елемент с име E има тип ID, това означава, че той ще приема уникални стойности за всяка порция от XML-документа, оградена с маркери <E> и </E>. В термините на полуструктурираните

данни това означава, че този атрибут осигурява уникална идентификация на съответния възел.
Ако един атрибут на елемент с име E има тип IDREF (IDREFS), това означава, че той ще приема стойност (списък от стойности) на атрибути ID, асоциирани с други елементи в XML-документа. В термините на полуструктурираните данни това може да се интепретира по следния начин: от съответния възел излизат дъги към възлите, чиито ID присъстват в стойността на атрибута от тип IDREF (IDREFS) на този възел, при това името на дъгата съвпада с името на атрибута от тип IDREF (IDREFS).

Следващият пример илюстрира синтаксисът за деклариране на атрибути от тип ID и IDREF (IDREFS) за елементи в DTD, както и използването на атрибутите в XML-документ за представяне на връзките в полуструктурираните данни.

```
<!DOCTYPE Stars-Movie [
        <!ELEMENT STARS-MOVIES (STAR*, MOVIE*)>
        <!ELEMENT STAR (NAME, ADDRESS+)>
                <!ATTLIST STAR
                        starID ID
                        starsIn IDREFS>
        <!ELEMENT NAME (#PCDATA)>
        <!ELEMENT ADDRESS (STREET, CITY)>
        <!ELEMENT STREET (#PCDATA)>
        <!ELEMENT CITY (#PCDATA)>
        <!ELEMENT MOVIE (TITLE, YEAR)>
                <!ATTLIST MOVIE
                        movieID ID
                        starOF IDREFS>
        <!ELEMENT TITLE (#PCDATA)>
        <!ELEMENT YEAR (#PCDATA)>
] >
```

Следният XML-документ отговаря на описаното DTD.

```
<STARS-MOVIES>
        <STAR starId = “cf” starsIn = “sw, esb, rj”>
                <NAME>Carrie Fisher</NAME>
                <ADDRESS><STREET>123 Maple Str.</STREET>
                        <CITY>Hollywood</CITY></ADDRESS>
                <ADDRESS><STREET>5 Locust Ln.</STREET>
                        <CITY>Malibu</CITY></ADDRESS>
        </STAR>
        <STAR starId = “mh” starsIn = “sw, esb, rj”>
                <NAME>Mark Hamill</NAME>
                <ADDRESS><STREET>456 Oak Rd.</STREET>
                        <CITY>Brentwood</CITY></ADDRESS>
        </STAR>

        <MOVIE movieId = “sw” starOf = “cf, mh”>
```

```
        <TITLE>Star Wars</TITLE>
        <YEAR>1977</YEAR>
    </MOVIE>
    <MOVIE movieId = “esb” starOf = “cf, mh”>
        <TITLE>Empire Strikes Back</TITLE>
        <YEAR>1980</YEAR>
    </MOVIE>
    <MOVIE movieID = “rj” starOf = “cf, mh”>
        <TITLE>Return of the Jedi</TITLE>
        <YEAR>1983</YEAR>
    </MOVIE>
</STARS-MOVIES>
```

Логически езици за заявки

Някои езици за заявки се основават на логиката повече отколкото на релационната алгебра. Те са значително по-трудни за усвояване от програмистите. Ще разгледаме логически език за заявки, наречен **Datalog** (database logic). Той се основава на релационния модел на данни и е близък до Prolog. Нерекурсивната версия на езика има мощта на класическата релационна алгебра. В рекурсивната версия на Datalog могат да се опишат заявки, които не могат да се опишат в SQL.

Логика за релациите

В Datalog заявките се състоят от правила, подобни на правилата в Prolog.
Всяко едно правило изразява идеята, че от определени комбинации на кортежи на дадени релации може да се извлече кортеж на друга релация.

Предикати и атоми

Основна роля в Datalog играят **предикатите** и **атомите**. Релациите в Datalog се представят чрез предикати. Всеки предикат получава фиксиран брой аргументи. **Атом** наричаме предикат, последван от аргументите си. Атомите се описват по начина по който се описват извиквания към функции в конвенционалните езици за програмиране. Например P (x_1, x_2, …, x_n) е атом, който се състои от предиката P с аргументи x_1, x_2, …, x_n.
Всъщност, предикатът може да се разглежда като функция, която връща булеви стойности. Ако R е име на релация с n атрибута в някакъв фиксиран ред, то ще използваме R като име на предикат, който съответства на тази релация. Атомът R (a_1, a_2, …, a_n) има стойност true, ако (a_1, a_2, …, a_n) е кортеж на релацията R и стойност false, ако (a_1, a_2, …, a_n) не е кортеж на релацията R.

Аргументите на един предикат могат да бъдат променливи или константи. Ако в един атом има поне една променлива, този атом се разглежда като булева функция на променливите на атома.

Атомите, които съответстват на релации наричаме **релационни атоми**. В Datalog могат се използват и **аритметични атоми**. Те представляват сравнение на два аритметични израза: например $x < y$, $x+1 \geq y + 4*z$.

Да отбележим, че както релационните, така и аритметичните атоми приемат като аргументи стойностите на променливите си и връщат булева стойност. В този смисъл можем да разглеждаме аритметичните атоми като релационни - например, аритметичното сравнение < може да се разглежда като име на релация, която съдържа всички двойки числа с първи компонент по-малък от втория компонент.
Такава релация, обаче, е безкрайна. Релациите съответни на релационни атоми са крайни и дори те могат да се променят във времето.
Поради тази причина аритметичните атоми се отделят от релационните.

Правила в Datalog

Операции, подобни на операциите от релационната алгебра могат да се опишат в Datalog с помощта на **правила**.
Едно правило се състои от три части:

1. Релационен атом, който се нарича **глава** на правилото.
2. Символът ←, който четем като “ако”.
3. **Тяло** на правилото, което се състои от един или повече релационни или аритметични атоми, наречени **подцели**. Подцелите са свързани с логическо и (AND), като е възможно някоя от тях да бъде отречена, т.е. предшествана от логическо отрицание (NOT).

Едно примерно правило е следното:
LongMovie (t, y) ← Movie (t, y, l, c, s, p) AND l >= 100
То се отнася за базата от данни със схема
Movie (title, year, length, inColor, studioName, producerC#).
Семантиката на това правило е следната: LongMovie (t, y) е истина точно когато в релацията Movie има кортеж със следните свойства:
стойност t за title, стойност y за year, стойност по-голяма от 100 за length и произволни стойности за останалите атрибути.
Да отбележим, че в релационната алгебра това правило е еквивалентно на следното присвояване: LongMovie := $\pi_{title,\ year}$ ($\sigma_{length>=100}$ (Movie)).

В правилата често се срещат **анонимни променливи**, които се появяват само веднъж. Имената на тези променливи са несъществени - те не свързват подцели или подцел с резултата. Поради тази причина приемаме общата конвенция, че underscore (_) като аргумент на атом означава променлива, която се появява само веднъж в съответното правило. Ако в едно правило се срещат няколко символа _, те означават различни анонимни променливи. Например горното правило може да се запише по следния начин:

LongMovie (t, y) ← Movie (t, y, l, _, _, _) AND l >= 100.

Една **заявка** в Datalog е съвкупност от едно или повече правила.
Ако в главата на всички правила от заявката присъства една и съща релация, то тази релация се приема за отговор на заявката. Естествено, предвид смисълът на релационните атоми, релацията резултат е обединение на всички релации, които се получават от правилата.
В горния пример отговорът на заявката е релацията LongMovie.
Ако в главите на правилата от заявката присъства повече от една релация, то точно една от тези релации е отговор на заявката, а останалите се разглеждат като обслужващи. Най-често за резултатната релация се използва име Answer.

Семантика на правилата в Datalog

Всяка променлива от едно правило има някаква област от стойности. Тогава когато при конкретни стойности на променливите всички подцели се оценяват до истина, само тогава се добавя съответен кортеж към релацията в главата на правилото.
Върху начина по който се използват променливи в правилата трябва да се въведат ограничения за да е сигурно, че резултатната релация ще бъде крайна и че аритметичните и отречените подцели ще носят интуитивния си смисъл. Едно достатъчно условие за това е така нареченото **условие за безопасност**, според което всяка променлива, която се появява в правило трябва да присъства и в някоя неотречена релационна подцел на това правило. По-точно променливите, които се появяват в главата, в отречените подцели и в аритметичните подцели на едно правило трябва да се появяват и в неотречена релационна подцел на това правило.
Например, правилото LongMovie (t, y) ← Movie (t, y, l, c, s, p) AND l >= 100 е безопасно, тъй като релационната подцел съдържа всички променливи, които участват в правилото.
В правилото P (x, y) ← Q (x, z) AND NOT R (x, w, z) AND x < y има три нарушения на условието за безопасност:

1. Променливата y присъства в главата, но не присъства в неотречена релационна подцел.
2. Променливата w присъства в отречена подцел, но не присъства в неотречена релационна подцел.
3. Променливата y присъства в аритметична подцел, но не присъства в неотречена релационна подцел.

Има и друг начин за дефиниране на семантиката на правилата. Вместо да разглеждаме всички възможни стойности на променливите, ограничаваме множеството от тези стойности до кортежите на неотречените релационни подцели. Когато някое присъединяване на кортежи е съгласувано, в смисъл че то присъединява една и съща стойност за всяка поява на една променлива в правилото, тогава разглеждаме резултатното присъединяване на стойности на всички променливи. Тъй като правилото е съгласувано, то всяка променлива ще

е получила стойност. За всяко съгласувано присъединяване при което отречените релационни подцели и аритметичните подцели се оценяват с истина, добавяме кортеж към резултатната релация в главата на правилото.

Ще разгледаме пример. Нека е дадено следното правило
P (x, y) ← Q (x, z) AND R (z, y) AND NOT Q (x, y).
В правилото има две неотречени релационни подцели - Q (x, z) и R (z, y).
Нека релацията Q има два кортежа (1, 2), (1, 3) и релацията R има два кортежа (2, 3), (3, 1).
Всевъзможните присъединявания на тези кортежи са следните:

Q (x, z)	R (z, y)	съгласувано
(1, 2)	(2, 3)	да
(1, 2)	(3, 1)	не
(1, 3)	(2, 3)	не
(1, 3)	(3, 1)	да

Разглеждаме само съгласуваните присъединявания. Освен неотречените релационни подцели, в правилото има още само една отречена релационна подцел. Тя се оценява до истина само при последното присъединяване. Така резултатната релация P има един кортеж (1, 1).

Екстенсионални и интенсионални предикати

Добре е да се прави разлика между:

1. **Екстенсионални предикати** - те съответстват на релации, които се съхраняват в базата от данни.
2. **Интенсионални предикати** - те съответстват на релации, които са резултат от изчисление на едно или повече правила.

В този смисъл в контекста на Datalog, една релация наричаме екстенсионална (**EDB**), ако тя съответства на екстенсионален предикат и интенсионална (**IDB**), ако тя съответства на интенсионален предикат.
В релационната алгебра екстенсионалните релации са операндите на изразите, а интенсионалните релации са междинните резултати и крайният резултат. В примера с филмите, релацията Movie е екстенсионална, а релацията LongMovie е интенсионална.

Една EDB не може да присъства в глава на правило, но може да присъства в тяло на правило. За IDB няма ограничения - може да присъства както в главата, така и в тялото на правило. Една IDB може да се дефинира с няколко правила, в които главата е един и същ интенсионален предикат. Използвайки IDB можем да дефинираме по-сложни релации. Този процес е подобен на изграждането на израз в релационната алгебра.

Прилагане на правила от Datalog към мултимножества

Семантиката на правилата в Datalog дефинирахме предполагайки, че релациите са множества от кортежи. В случай че в правилата не присъстват отречени релационни подцели, същата идея може да се приложи и за релации, които са мултимножества от кортежи.
Концептуално по-лесно е да се използва вторият подход - ако набор от кортежи за всяка релационна подцел, която не е отречена дава съгласувани стойности за всяка променлива в правилото и всички аритметични подцели се оценяват като истина, то добавяме съответен кортеж към релацията в главата. При това, тъй като в случая релациите са мултимножества, не премахваме дубликатите от резултата.
Също така, ако един кортеж присъства много пъти в релация, съответна на релационна подцел, то всяко срещане на кортежа се разглежда независимо когато оценяваме променливите в правилото.
Изрично отбелязваме, че в модела с мултимножества не може да се дефинира ясно смисълът на правила, в които участват отречени релационни подцели.
Ще разгледаме пример. Нека е дадено следното правило:
H (x, z) ← R (x, y) AND S (y, z).
Нека релацията R има следните кортежи: (1, 2), (1, 2), а релацията S има следните кортежи: (2, 3), (4, 5), (4, 5). Всевъзможните присъединявания на тези кортежи са следните:

R (x, y)	S (y, z)	съгласувано
(1, 2)	(2, 3)	да
(1, 2)	(4, 5)	не
(1, 2)	(4, 5)	не
(1, 2)	(2, 3)	да
(1, 2)	(4, 5)	не
(1, 2)	(4, 5)	не

Резултатната релация H има два кортежа: (1, 3), (1, 3). По-общо, ако в релацията R имаше n кортежа (1, 2), а в релацията m кортежа (2, 3), то в релацията H щяха да присъстват n.m кортежа (1, 3).

Ако една релация е дефинирана чрез повече от едно правила, то тя се получава като мултимножествено обединение на резултатните релации.
Ще разгледаме пример. Нека е дадена следната заявка:
H (x, y) ← S (x, y) AND x > 1,
H (x, y) ← S (x, y) AND y < 5.
Тук релацията S има три кортежа: (2, 3), (4, 5), (4, 5).
Тогава резултатната релация H ще има следните кортежи:
(2, 3), (2, 3), (4, 5), (4, 5).

Преобразуване на изразите от релационната алгебра в Datalog правила

Всяка една от операциите в релационната алгебра може да се моделира чрез правила. Първо ще покажем как се моделират основните операции, а след това ще видим как се моделират изрази от релационната алгебра с произволна сложност.

Сечение

Сечението на две релации се моделира с едно правило, което съдържа по една подцел за всяка от двете релации. При това в двете подцели се използват еднакви променливи в съответните аргументи.
Например, ако R и S са релации, които имат по два атрибута, то тяхното сечение се пресмята от следното Datalog правило:
Intersection (x, y) ← R (x, y) AND S (x, y).

Обединение

Обединение на две релации се моделира с две правила. Те имат по една релационна подцел, отговаряща на всяка от релациите. Аргументите на главата на всяко от правилата съвпадат с аргументите на техните подцели. Например, ако R и S са релации, които имат по два атрибута, то тяхното обединение се пресмята от следната заявка:
Union (x, y) ← R (x, y),
Union (x, y) ← S (x, y).
Да отбележим, че за разлика от моделирането на сечението, което е правилно само ако релациите са множества, при обединението моделирането е коректно дори ако релациите са мултимножества.
По принцип в тези разглеждания предполагаме, че релациите се интерпретират като множества, освен ако изрично не е указано друго.
Също да отбележим, че няма връзка между променливите в различни правила. Причината е, че всяко правило се оценява независимо от другите правила. Например, по-горе можем да заменим второто правило със следното: Union (u, v) ← S (u, v). Естествено, когато преименуваме една променлива, трябва да внимаваме да не променим смисъла на правилото - новата променлива не трябва да се среща в правилото и освен това трябва да се преименуват всички срещания на старата променлива.

Разлика

Разликата на две релации се моделира с едно правило с една отречена релационна подцел, съответна на релацията в лявата част на разликата и една неотречена релационна подцел, съответна на релацията в дясната част на разликата. Тези подцели, както и главата имат едни и същи аргументи. Например, ако R и S са релации, които имат по два атрибута, разликата R - S се пресмята от следното правило:
Difference (x, y) ← R (x, y) AND NOT S (x, y).

Проекция

Проекцията на една релация се пресмята с едно правило с единствена релационна подцел, която съответства на релацията. За аргументи на подцелта се използват различни променливи, по една за всеки атрибут на релацията. За аргументи на главата се използват променливите от аргументите на подцела, съответстващи на атрибутите по които се извършва проекцията. Например, ако R има три атрибута, то проекцията на R по първите два атрибута се пресмята от следното правило: Projection (x, y) ← R (x, y, z).

Селекция

Селекцията на една релация относно условие по принцип се изразява по-трудно в Datalog. Най-простият случай е когато условието е конюнкция на едно или повече аритметични сравнения.
В този случай образуваме едно правило, което има:

1. Една релационна подцел за релацията, върху която се извършва селекция, при това в тази подцел се използват различни променливи, по една за всеки атрибут на релацията.
2. По една аритметична подцел за всяко от аритметичните сравнения, съвпадаща с това сравнение. Естествено, аргументите на подцелите съответстват на аргументите, които се използват в релационната подцел.

Ще разгледаме пример с базата от данни Movie.
Нека е дадена следната селекция: $\sigma_{\text{length>=100 AND studioName =' Fox'}}$ (Movie).
В Datalog тя се моделира по следния начин:
Selection (t, y, l, c, s, p) ← Movie (t, y, l, c, s, p) AND l >= 100 AND s = “Fox”.

Ако условието на селекцията е дизюнкция на няколко условия, то трябва да се използва факта, че селекция по дизюнкция от условия е еквивалентна на обединение на селекциите по всяко от условията и да се използва техниката при моделиране на обединение.
Например, нека е дадена следната селекция:
$\sigma_{\text{length>=100 OR studioName =' Fox'}}$ (Movie).
В Datalog тя се моделира със следната заявка:
Selection (t, y, l, c, s, p) ← Movie (t, y, l, c, s, p) AND l >= 100,
Selection (t, y, l, c, s, p) ← Movie (t, y, l, c, s, p) AND s = “Fox”.

Сега да разгледаме общият случай. Произволно условие може да се приведе в дизюнктивна нормална форма - дизюнкция на конюнкции на аритметични сравнения или техни отрицания. Да отбележим, че отрицание на аритметично сравнение може да се преобразува в аритметично сравнение - например, NOT A ≤ B е еквивалентно на A > B, NOT A = B е еквивалентно на A ≠ B и т.н. Всяка конюнкция може да се представи чрез едно правило, както по-горе описахме и след това дизюнкцията представяме чрез няколко правила, по едно за всяка конюнкция.

Декартово произведение

Декартовото произведение на две релации се изразява с едно правило с две подцели, по една за всяка от релациите. Всяка от тези подцели има различни променливи, по една за всеки от атрибутите на релациите. Главата на правилото има за аргументи всички променливи и от двете подцели, при това променливите на релацията в лявата част на декартовото произведение предшестват променливите на релацията в дясната част. Например, ако R и S са релации с два атрибута декартовото произведение R x S се пресмята чрез следното правило: Product (x, y, u, v) ← R (x, y) AND S (u, v).

Съединения

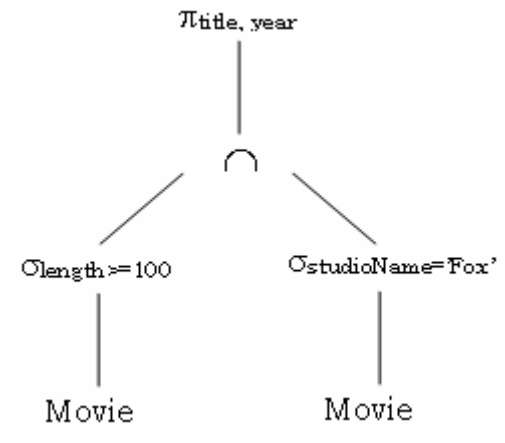
Моделирането на естествено съединение в Datalog е подобно на моделирането на декартово произведение, но с тази разлика, че трябва да се използват еднакви променливи за общите атрибути на двете релации. Например, ако имаме две релации със следните схеми: R (A, B), S (B, C, D), то тяхното естествено съединение се описва със следното правило: NaturalJoin (a, b, c, d) ← R (a, b) AND S (b, c, d).

Естествено, тита-съединенията също могат да се изразят в Datalog. Ако условието е конюнкция на аритметични сравнения, то използваме едно правило, което описва декартово произведение на двете участващи релации, като добавяме по една аритметична подцел за всяко от аритметичните сравнения.
Като пример разглеждаме две релации със следните схеми: U (A, B, C) и V (B, C, D). Нека имаме следното тита-съединение: $U \bowtie_{A < D \text{ AND } U.B \neq V.B} S$. То се моделира със следното правило:
Thetajoin (a, ub, uc, vb, vc, d) ← U (a, ub, uc) AND V (vb, vc, d) AND a < d AND ub ≠ vb.
Ако условието в тита-съединението е произволно, то го привеждаме в дизюнктивна нормална форма. За всяка конюнкция образуваме по едно правило по описания начин. Естествено, главите на правилата, съответни на конюнкциите трябва да са едни и същи.
Например, за горните релации U и V да разгледаме следното тита-съединение: $U \bowtie_{A < D \text{ OR } U.B \neq V.B} S$. То се описва със следната заявка:
Thetajoin (a, ub, uc, vb, vc, d) ← U (a, ub, uc) AND V (vb, vc, d) AND a < d,
Thetajoin (a, ub, uc, vb, vc, d) ← U (a, ub, uc) AND V (vb, vc, d) AND ub ≠ vb.

Моделиране на произволни релационни изрази

Идеята при това моделиране е за всеки вътрешен връх в дървото, съответно на релационния израз да се използва по един интенсионален предикат. Правилото или правилата, които описват всеки от тези предикати се получават в зависимост от операцията, която се намира

във възела. Естествено, операндите в листата на дървото се представят чрез съответните им екстенсионални предикати.
Например, нека имаме следният релационен израз:
$\pi_{title,\ year}$ ($\sigma_{length>=100}$ (Movie) ∩ $\sigma_{studioName='Fox'}$ (Movie)).
Той се представя чрез следното дърво:



За да моделираме този израз в Datalog можем да използваме следната заявка:
W (t, y, l, c, s, p) ← Movie (t, y, l, c, s, p) AND l >= 100
X (t, y, l, c, s, p) ← Movie (t, y, l, c, s, p) AND s = "Fox"
Y (t, y, l, c, s, p) ← W (t, y, l, c, s, p) AND X (t, y, l, c, s, p)
Z (t, y) ← Y (t, y, l, c, s, p)

Рекурсия в Datalog

Съществуват заявки, които не могат да се изразят с помощта на релационната алгебра. Пример за такива заявки са безкрайни, рекурсивни поредици от операции.

Ще разгледаме пример, в контекста на базата от данни за филмите. Един филм може да има продължение, продължението от своя страна също може да има продължение и т.н. Да си мислим, че имаме релация със следната схема SequelOf (movie, sequel). Нейните кортежи съдържат двойки от филм и негово непосредствено продължение.
Примерен екземпляр на релацията е следния:

movie	sequel
Naked Gun	Naked Gun 2
Naked Gun 2	Naked Gun 3

Да предположим, че имаме следната заявка: да се състави релация, която съдържа всички двойки от филм и негово продължение.
Под продължение разбираме продължение от произволна дълбочина.
Не е ясно как да се опише подобна заявка в релационната алгебра.
Можем да опишем продължение на филм на две нива, т.е. непосредствено продължение на непосредствено продължение по следния начин: $\pi_{first,\ third}$ ($\rho_{R\ (first,\ second)}$ (SequelOf) ⋈ $\rho_{R\ (second,\ third)}$ (SequelOf)).

По аналогичен начин, чрез суперпозиция на естествени съединения можем да получим продължения на произволно фиксирано ниво i.
Това, което не можем в релационната алгебра е да образуваме безкрайно обединение на релациите за i = 1, 2, ….

В Datalog въпросната заявка може да се опише, ако се използва един и същи интенсионален предикат в главата и в тялото на правило.
Това става по следния начин:
FollowOn (x, y) ← SequelOf (x, y)
FollowOn (x, y) ← SequelOf (x, z) AND FollowOn (z, y).

Възниква въпросът как да се определи семантиката на рекурсивните заявки в Datalog. Това се постига чрез изчисляване на най-малка неподвижна точка за интенсионалните предикати. Също така, възникват интересни проблеми при използване на отрицание в рекусивните заявки.