**Лекция 14:**

**Физическая модель хранилища данных: учет влияния транзакций, денормализация таблиц**

**Аннотация:**В настоящей лекции рассматриваются вопросы учета влияния транзакции при проектировании физической структуры хранилища данных и принципы денормализации на уровне моделирования логической и физической моделей данн

**Цель лекции**

Изучив материал настоящей лекции, вы будете знать:

* основные типы приложений баз данных;
* что понимается под *транзакцией* базой данных;
* как можно описать *транзакции* базы данных;
* основные типы *денормализации* таблиц базы данных;
* что такое *вертикальное* и *горизонтальное разбиение* таблиц;
* о проблеме длинной строки в *таблице хеширования* ;
* что такое методы *денормализации* таблиц;
* что такое *денормализация колонки* таблицы;

и научитесь:

* выполнять *разделение таблиц* ;
* выполнять объединение таблиц;
* выполнять *денормализацию колонки* таблицы.

**Литература**: [[37](https://intuit.ru/studies/courses/599/455/literature#literature.37)], [[48](https://intuit.ru/studies/courses/599/455/literature#literature.48)], [[54](https://intuit.ru/studies/courses/599/455/literature#literature.54)].

**Определение транзакций базы данных**

**Учет типа приложений базы данных**

Прежде чем обсуждать основные типы приложений баз данных, уточним термины *транзакция* (transaction) и *запрос*. В теории БД, вообще говоря, под ***транзакцией*** **понимают одну из команд SQL** — SELECT, INSERT, UPDATE, DELETE. Однако в зависимости от типа приложений термин " *транзакция* " трактуется более свободно, как **элементарная логически завершенная единица работы** (так называемая **бизнес-транзакция** ), которая может включать несколько команд вставки, удаления или модификации. В зависимости от того, какие команды SQL используются, *транзакции* разделяют на **транзакции только для записи** (write-only), **только для модификации** (modify-only), **только для чтения** (read-only), **только для удаления** (delete-only). **Транзакции только для чтения называют** ***запросом***.

Исходя из такого деления *транзакций* по типам обработки с учетом частоты *транзакций* каждого типа, можно выделить три основных классических типа приложений БД.

1. OLTP-системы (On-Line Transaction Processing). OLTP-система – это такое приложение, которое содержит в основном *транзакции* вставки, обновления и удаления, с высокой частотой преимущественно *транзакций* обновления. Классическим примером таких систем являются системы резервирования авиабилетов или обслуживания гостиниц. Для них характерен высокий **уровень параллелизма** (high concurrency), который в данном случае означает, что много пользователей используют базу данных одинаковым образом.
2. DSS-системы (*Decision Support System*). DSS-система – это такое приложение, которое работает с очень большой базой данных в режиме "только чтение". Обычно используются набор фиксированных простых *запросов* или нерегламентированные *запросы* пользователей. Хорошим примером такой системы является корпоративная информационная система организации.
3. BATCH-системы. BATCH-системы – это такое приложение, которое работает с базой данных в неинтерактивном режиме. Обычно оно использует много *транзакций* вставки, удаления и обновления и имеет низкий уровень параллелизма, что означает небольшое число пользователей, использующих базу данных одинаковым образом. Существенным фактором является отношение *запросов* к *транзакциям* обновления. Классическим примером таких систем является обслуживание БД продукции организации.

Можно выделить еще несколько типов приложений, появившихся в последние два десятилетия.

* OLAP-системы (On-Line Analytical Processing). OLAP-система – это приложение, которое обеспечивает аналитическую обработку данных, включающую математический, статистический или иной анализ данных. Такие системы нельзя отнести полностью либо к OLTP-, либо DSS-системам. Они располагаются где-то между ними. В рамках OLAP-систем выделяют так называемые *ROLAP*-системы (Relational OLAP), т.е. OLAP-системы, использующие реляционные базы данных. Типичные OLAP-системы разрабатываются обычно под многомерные модели данных.
* VCDB-системы (Variable *Cardinality* Database). VCDB-система — это такое приложение обработки данных, для которого база данных растет или сжимается в размерах периодически, в зависимости от характера обработки данных. Обычно размер этих баз данных постоянно растет. Типичным примером такой системы является БД по обеспечению безопасности (security authorization database), для которой характерна короткая по времени активность записей в таблицы.

Для анализа *транзакций* необходимо оценить тип приложений, для которых разрабатывается база данных. Это позволит оценить:

* тип *транзакций* (какие);
* частоту *транзакций* каждого типа (сколько);
* количество одновременно работающих с БД пользователей.

**Спецификация транзакций**

Напомним, *транзакция* БД – это логическая единица работы, которая переводит БД из одного завершенного состояния в другое завершенное состояние. Под завершенным состоянием БД здесь понимается такое состояние, которое не нарушает целостности этих данных, когда все данные в таблицах базы данных правильны, а ссылки между таблицами корректны. В дополнение *транзакция* группирует операции над данными таким образом, чтобы все обращения к БД были успешно завершены или, в случае сбоя, БД возвратилась в предыдущее завершенное состояние (откат *транзакции* ).

Поскольку любая операция изменения данных в БД несет в себе потенциальную возможность нарушения целостности данных, необходимо строго определять *транзакции* и идентифицировать информацию, которая обычно включается в определение *транзакции*.

Определение *транзакции* может иметь различные формы. Иногда для определения *транзакций* используется репозиторий данных CASE-средств проектирования базы данных. Очень часть определение *транзакций* выполняется посредством текстовых описаний. Независимо от выбранного подхода, любое хорошее определение *транзакции* включает несколько важных элементов. К таким элементам относятся:

* имя *транзакции* ;
* номер *транзакции* ;
* описание *транзакции* ;
* характер *транзакции* и ее сложность;
* объем *транзакции* ;
* требования к *производительности транзакции* ;
* относительный приоритет;
* время выполнения *транзакции*.

Первым шагом в определении *транзакций* является уникальная идентификация каждой *транзакции* БД. Это можно сделать назначением имени и номера каждой *транзакции* БД. Имена *транзакций* должны позволять пользователям отличать их друг от друга. Описание включает перечень операций предметной области, которые выполняются *транзакцией*. Оно должно быть выполнено в терминах предметной области, понятных пользователю. Здесь нужно иметь в виду следующее:

* описание *транзакции* должно указывать, **что** *транзакция* делает для пользователя, а не **как** она выполняется;
* описание должно быть понятно пользователю, что не исключает использование технологического жаргона.

**Пример 19.1**. Описание *транзакции*.

*Имя транзакции*: назначить работу служащему.

*Описание транзакции*. *Транзакция* проверяет, не назначена ли уже данная работа данному служащему, затем проверяет, располагает ли служащий рабочим временем для выполнения данной работы. Если результаты проверки положительны (данная работа не назначена данному служащему, и служащий располагает временем для ее выполнения), то происходит назначение данной работы данному служащему.

В OLTP-системах большинство *транзакций* известны заранее, поэтому между *спецификацией транзакции* и *транзакцией* БД существует взаимно однозначное соответствие. В DSS-системах *транзакции* часто неизвестны заранее, и, следовательно, невозможно в принципе описать их все. В этом случае спецификация лишь в общих чертах описывает *транзакции* БД, поэтому важно уметь предсказать тип *транзакций*, которые пользователь, вероятнее всего, будет выполнять в БД.

Для каждой *транзакции* может быть определен **характер транзакции** (онлайновая *транзакция* или пакетная *транзакция* ), а также указана ее **сложность**. Обычно сложность указывается в терминах "высокая", "средняя", "низкая". Эта информация нужна для оценки *транзакций* базы данных в целом. Количество *транзакций* той или иной сложности влияет на время проектирования физической модели базы данных: чем больше в базе данных *транзакций* высокой сложности, тем больше время проектирования физической модели. Сложность *транзакции* является условной мерой трудоемкости при достижении требований производительности базы данных.

Высокая сложность приписывается обычно *транзакции*, которая имеет две из следующих характеристик:

* содержит от 8-ми до 10-ти команд SQL;
* содержит предложение WHERE с большим количеством предикатов;
* содержит предложение WHERE с более чем тремя соединениями или подзапросами;
* *транзакция* обрабатывает более чем 100 строк.

Низкая сложность приписывается *транзакции* с следующими характеристиками:

* содержит до 3-х команд SQL;
* содержит предложение WHERE с одним или двумя предикатами;
* *транзакция* обрабатывает менее чем 25 строк.

*Транзакция* со средней сложностью имеет характеристики между низкой и высокой сложностью.

**Пример 19.2**. Определение характеристик *транзакции*.

*Характер транзакции*: онлайновая *транзакция*.

**Сложность**: средняя.

Информация о частотах *транзакций* включает обычно два параметра — среднюю частоту *транзакции* (например, 50 тр/ч) и пиковую частоту *транзакции* (например, 70 тр/ч). Оценка частотных характеристик БД очень важна для проектирования *физической модели данных* ХД: настройка физической структуры БД для *транзакций* с высокой частотой существенно отличается от настройки ее для *транзакции* с низкой частотой использования.

Пример 19.3.

*Средняя частота транзакции*: до 10 в день.

*Пиковая частота*: 10 в час.

Если для модернизации существующих систем информация о частотах *транзакций* может быть представлена во входной документации на основе анализа статистики эксплуатации системы, то для создаваемых ХД такую информацию получить практически невозможно, и приходится пользоваться приблизительными оценками.

*Спецификация транзакции* должна включать требования по производительности. Производительность БД в целом складывается из *производительности* каждой *транзакции* в отдельности и их распределении во времени. Это очень важная информация для решения задачи настройки физической структуры ХД. Требования по производительности обычно задаются в виде параметра **время реакции**, т.е. количества секунд, требуемых для выполнения *транзакции*. Например, во многих банковских системах *транзакция* для снятия денег со счета клиента не должна занимать более 5 секунд.

Другой формой задания требования на *производительность транзакций* БД является указание их характера и сложности. Например,

* онлайновые *транзакции* высокой сложности должны выполняться не более 15 с;
* онлайновые *транзакции* *средней сложности* должны выполняться не более 7 с;
* онлайновые *транзакции* низкой сложности должны выполняться не более 4 с;
* пакетные *транзакции* высокой сложности должны выполняться не более 1 часа;
* пакетные *транзакции* *средней сложности* должны выполняться не более 0.5 часа;
* пакетные *транзакции* низкой сложности должны выполняться не более 15 мин.

Получение требований на *производительность транзакций* является одним из самых узких мест как в исходной документации, так и в проектировании физической модели ХД. На практике получить временные характеристики выполнения *транзакций* часто удается только на этапе опытной эксплуатации, а то и промышленной эксплуатации ХД.

Желательно в *спецификации транзакции* указать ее **относительный приоритет**, который показывает, насколько важна настоящая *транзакция* для предметной области по сравнению с другими. Относительный приоритет позволяет сгруппировать *транзакции* по этому параметру, что дает возможность последовательно оценить все *транзакции* БД и их вклад в производительность БД.

Выполнение какой-либо *транзакции* в БД с большим приоритетом отрицательно сказывается на *производительности* других *транзакций*. Ранжирование по относительному приоритету позволяет сбалансировать влияние *транзакций* друг на друга в целом.

Задание приоритета *транзакций* может иметь различные формы. Обычно такое действие сводится к субъективной оценке в виде числа от 1 до 10.

Каждая *спецификация транзакции* должна содержать команды SQL, которые задают операции с БД. Указание команд SQL в контексте создания физической модели ХД позволяет оценить **время выполнения транзакций** (*execution time*), т.е. фактическое количество секунд, необходимое для завершения *транзакции* в режиме эксплуатации ХД. Проектировщику ХД этот параметр важен еще и с точки зрения составления *спецификаций модулей* приложений ХД для разработчиков приложений.

Помимо собственно команды языка манипулирования данными, желательно включить некоторый комментарий к каждой команде, в котором указать: а) что команда делает, б) почему это требуется и в) количество строк в ХД, которое захватывается командой. Время выполнения команды SQL непосредственно зависит от числа обрабатываемых командой строк. Обычно время выполнения *транзакций* можно оценить на стадии опытной эксплуатации и тестирования базы данных.

**Пример 19.4**.

|  |  |
| --- | --- |
| **Команда** | **Комментарий** |
| Select works from project where empno=:1 and works=:2 | Возвращает информацию о назначении данной работы данному служащему. По крайней мере, одна строка возвращается. Число строк, которые могут обрабатываться командой, равно текущему размеру таблицы PROJECT |
| Select works from project where empno=:1 | Возвращает список работ данного служащего, чтобы оценить его загруженность. Число строк, которые могут обрабатываться командой, равно текущему размеру таблицы PROJECT |
| Insert into project empno, works values(:1, :2) | Назначает данного служащего на данную работу, если это необходимо |

Описание *транзакций* позволяет принимать или откладывать решение об изменении физической схемы ХД с целью исполнения требований по производительности базы данных.

Механизмы, с помощью которых можно обеспечить выполнение требований производительности, обсуждаются ниже, в предположении, что для создания ХД выбрана СУБД семейства MS SQL Server, за исключением одного примера, в котором используется диалект SQL СУБД семейства Oracle.

**Денормализация**

**Понятие о денормализации**

Начиная с этого раздела, мы будем рассматривать методики настройки физической структуры реляционной БД, используемой для реализации ХД, с целью удовлетворения требования к производительности ХД. Эти методики представляют собой набор рекомендаций и эвристических правил по изменению физической структуры БД, которая была получена в результате первой итерации создания *физической модели данных* ХД. Ясно, что применение этих методик носит рекомендательный характер.

В этом разделе будут описаны различные типы *денормализации* и методы ее реализации. Кроме того, мы рассмотрим, как при выполнении *денормализации* обеспечить целостность данных, не прибегая к созданию дополнительного кода.

Под ***денормализацией*** **понимают процесс достижения компромиссов в нормализованных таблицах посредством намеренного введения избыточности в целях увеличения производительности**.

В большинстве случаев необходимость *денормализации* становится очевидной лишь на этапе проектирования приложений ХД или его эксплуатации. Другими словами, нельзя принять решение о *денормализации* на основании одной только модели данных. Обычно стараются найти в приложениях ХД *критичные процессы* и принимать решения о *денормализации* в основном в пользу этих процессов. Критичные процессы, как правило, определяют по высокой частоте, большому объему, высокой изменчивости или явному приоритету. Качественное описание *транзакций* БД позволяет определить наличие таких критических процессов.

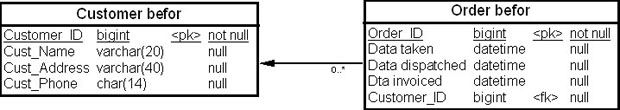
Заметим, что применять *денормализацию* только для упрощения SQL-запросов при обращении к ХД является неправильным решением. Если вы хотите упростить SQL-запросы на уровне приложения или пользователя, то, наверное, лучше использовать представления, а не вводить избыточность. Чтобы повысить производительность *запроса*, можно ввести индексы.

Как правило, *денормализация* уменьшает время *запроса* за счет DML-операций. *Денормализацию* следует рассматривать как расширение нормализованной модели данных, которое повышает производительность *запросов*. При приятии решения о *денормализации* следует определить, что является наиболее важным для приложения – избыточность данных или высокая производительность. Если ведется журнал проектирования (некоторый внутренний документ произвольной формы, в котором фиксируются все принятые в процессе проектирования ХД решения), то в него необходимо занести обоснованное решение о *денормализации*. Необходимо помнить, что кроме *денормализации* существуют и другие пути повышения производительности. *Денормализацию* таблиц можно выполнять как на уровне логической модели данных, так и на уровне физической модели.

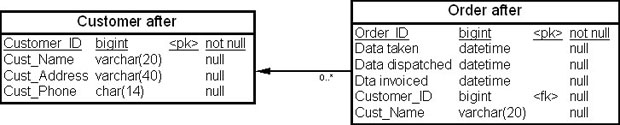
**Нисходящая денормализация**

Рассмотрим принципы *денормализации* на уровне *физической модели данных*. ***Нисходящая денормализация*** предлагает перенос колонки из одной (родительской) таблицы в подчиненную (дочернюю) таблицу.

На [рис. 19.1](https://intuit.ru/studies/courses/599/455/lecture/10180?page=2#image.19.1) показаны две таблицы – "Покупатель" (Customer befor) и "Заказ" (Order befor) – *физической модели данных* до проведения *денормализации*, а на [рис. 19.2](https://intuit.ru/studies/courses/599/455/lecture/10180?page=2#image.19.2) — эти же таблицы, "Покупатель" (Customer after) и "Заказ" (Order after), после выполнения *нисходящей денормализации*.

[](https://intuit.ru/EDI/20_07_20_2/1595197216-9970/tutorial/632/objects/19/files/13_01.jpg)

[увеличить изображение](https://intuit.ru/EDI/20_07_20_2/1595197216-9970/tutorial/632/objects/19/files/13_01.jpg)  
**Рис. 19.1.**Таблицы "Покупатель" (Customer befor) и "Заказ" (Order befor) до денормализации

[](https://intuit.ru/EDI/20_07_20_2/1595197216-9970/tutorial/632/objects/19/files/13_02.jpg)

[увеличить изображение](https://intuit.ru/EDI/20_07_20_2/1595197216-9970/tutorial/632/objects/19/files/13_02.jpg)  
**Рис. 19.2.**Таблицы "Покупатель" (Customer after) и "Заказ" (Order after) после денормализации

Из рисунков видно, что в денормализованной модели мы переместили колонку "Фамилия покупателя" (Cust\_Name) из таблицы "Покупатель" (Customer after) в таблицу "Заказ" (Order after).

Что дает введение избыточности (перенос колонки) в данном случае? Единственный выигрыш заключается в том, что мы исключаем операцию соединения, если захотим вместе с заказом увидеть фамилию клиента.

Таким образом, *нисходящая денормализация* *– это процесс введения избыточных колонок в подчиненных таблицах с целью устранения операций соединения*.

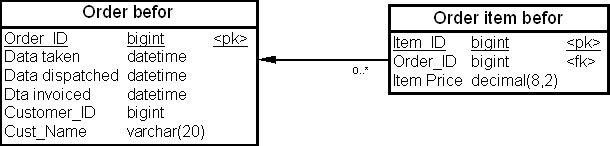
Однако устранение соединений посредством *нисходящей денормализации* редко оправдывает затраты на сопровождение дублирующей колонки в таблице "Заказ" (Order). Такие соединения, как правило, не являются глобальной проблемой, а выполнение *нисходящей денормализации* может привести к возникновению дорогостоящих *каскадных обновлений*. Например, если покупатель меняет фамилию, то приходится обновлять все заказы, чтобы отразить это изменение. А нужно ли это делать? Следует ли обновлять старые заказы, которые выполнены или закрыты? Если бы не была проведена *денормализация*, эти вопросы никогда и не возникли бы.

*Нисходящая денормализация* оправдана лишь в приложениях, где необходимо устранять операции соединения таблиц. Это имеет место в ХД большого объема. При этом проблем с каскадными обновлениями не возникает, потому что данные в ХД обладают свойством стабильности, т.е. не изменяются после занесения их в ХД.

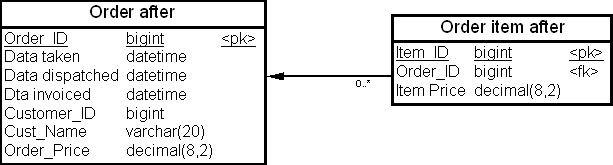
**Восходящая денормализация**

*Восходящая денормализация* предлагает перенос колонки из подчиненной (дочерней) таблицы в родительскую таблицу, обычно в форме итоговых данных.

На [рис. 19.3](https://intuit.ru/studies/courses/599/455/lecture/10180?page=2#image.19.3) показаны две таблицы – "Заказ" (Order befor) и "Позиция заказа" (Order item befor) – *физической модели данных* до проведения *денормализации*, а на [рис. 19.4](https://intuit.ru/studies/courses/599/455/lecture/10180?page=2#image.19.4) — эти же таблицы "Заказ" (Order after) и "Позиция заказа" (Order item after) после выполнения *восходящей денормализации*.



**Рис. 19.3.**Таблицы "Заказ" (Order befor) и "Позиция заказа" (Order item befor) до денормализации



**Рис. 19.4.**Таблицы "Заказ" (Order after) и "Позиция заказа" (Order item after) после денормализации

Например, если вычисление общей суммы заказа в системе обработки заказов (суммирование колонок "Цена позиции заказа" (Item\_Price) в таблице "Позиции заказа" (Order Item)) приводит к снижению производительности, то мы можем повысить производительность этой операции, поместив сумму заказа в избыточной колонке "Сумма заказа" (Order Price) таблицы "Заказ" (Order after).

В нашем примере в избыточном столбце хранится сумма значений, но *восходящая денормализация* применима к максимальным, минимальным и средним значениям, а также к другим агрегатным показателям.

Таким образом, ***восходящая денормализация*** **- это процесс введения избыточных колонок в родительских таблицах с целью устранения операций соединения с операциями агрегирования**.

Чтобы представить результат введения *денормализации*, рассмотрим процедуру сопровождения денормализованных таблиц "Заказ" (Order after) и "Позиция заказа" (Order item after), которые сводятся к поддержке следующих бизнес-правил.

* Когда в таблицу "Позиция заказа" (Order item after) добавляется новая строка, колонка "Сумма заказа" (Order\_Price) в таблице "Заказ" (Order after) увеличивается на значение колонки "Цена позиции заказа" (Item\_Price) таблицы "Позиция заказа" (Order item after) для новой позиции заказа.
* Когда строка удаляется из таблицы "Позиция заказа" (Order item after), значение колонки "Цена заказа" в таблице "Заказ" (Order after) уменьшается на значение колонки "Цена позиции заказа" (Item\_Price) в таблице "Позиция заказа" (Order item after) для удаляемой строки из позиции заказа.
* Когда изменяется значение колонки "Цена позиции заказа" (Item\_Price) в таблице "Позиция заказа" (Order item after), значение колонки "Сумма заказа" (Order\_Price) в таблице "Заказ" (Order after) должна быть откорректирована на разницу между старым и новым значением колонки "Цена позиции заказа" (Item\_Price).

Поддержка перечисленных выше бизнес-правил создает дополнительную нагрузку на процессы, выполняющие DML-операции в таблице "Позиция заказа" (Order item after). Это и есть цена, которую приходится платить за повышение производительности *запросов*.

**Внутритабличная денормализация**

*Внутритабличная денормализация* выполняется в пределах одной таблицы, т.е. это процесс введения избыточных колонок в одной таблице с целью увеличения производительности *запроса* строки по производному значению. Например, если строка содержит две числовых колонки, X и Y, то значение Z, равное произведению X и Y (Z=X\*Y), легко вычислить во время выполнения.

Предположим, что есть *запросы*, в которых необходимо осуществить поиск по Z (например, Z принадлежит диапазону от 10 до 20). Сохранив избыточные значения Z в столбце, можно построить индекс по Z, и *запросы* будут использовать этот индекс. Если индекс по Z строить не надо, то решение о его хранении в отдельном столбце зависит от того, что является более приемлемым — увеличение времени загрузки, вызванное необходимостью постоянно пересчитывать Z, или увеличение времени сканирования, обусловленное удлинением строк таблицы за счет хранения дополнительной колонки.

Приведем еще один часто встречающийся пример *внутритабличной нормализации*. Допустим, что одинаковый текст хранится в двух видах: с символами в верхнем и в нижнем регистре — для отображения в отчетах и на экране, и с символами в верхнем регистре — для обеспечения работы ускоренных *запросов* без учета регистра.

Отметим, что обеспечить приемлемую производительность для таблиц умеренного размера (до 10000 строк) в последнем случае можно и без *внутритабличной нормализации*, переработав *запрос* с применением встроенной функции UPPER.

*Внутритабличная нормализация* редко используется при проектировании ХД.

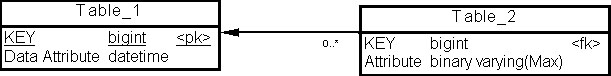
**Денормализация методом "разделяй и властвуй"**

***Денормализация методом "разделяй и властвуй"*** – это процесс разбиения нормализованной таблицы на две таблицы и более и создание между ними отношения "один к одному" с целью устранения дополнительных операций ввода-вывода или по техническим причинам.

Использование этого приема обычно носит причины технического характера. Как правило, этот метод применяется для *денормализации* таблиц, содержащих колонки с данными типа text или varbinary (max), размер которых может составлять 64К и более.

Иногда лучше вынести такую колонку в отдельную таблицу. Рассмотрим таблицу, строки которой содержат в начале ключевые колонки, потом неключевые колонки, а в конце – колонку типа varbinary (max). Предположим, что в большинстве строк колонка типа varbinary (max) содержит данные. Если нет индексов по неключевым столбцам, то при выполнении *запросов* по любому из этих столбцов СУБД обычно будет осуществлять полное сканирование таблицы. При этом из-за наличия в таблице колонки типа varbinary (max) понадобятся дополнительные операции ввода-вывода.

Чтобы устранить эту проблему, необходимо разделить таблицу так, как показано на [рис. 19.5](https://intuit.ru/studies/courses/599/455/lecture/10180?page=3#image.19.5).



**Рис. 19.5.**Выделение колонки типа varbinary (max) в отдельную таблицу

В некоторых СУБД таблица не может иметь более 254 столбцов, и если для представления данных нужна таблица с большим числом столбцов, то также возникнет причина для *разделения* такой таблицы на две. Как правило, такие таблицы возникают в следующих случаях;

* приложение полностью проектируется на основе БД унаследованной системы, и каждая таблица строится как точная копия файла унаследованной системы. При этом наследуется и структура, а все реляционные свойства в ней отсутствуют;
* выполняется слияние двух таблиц путем формирования в одной из них повторяющейся группы;
* проектируется ХД, для которого принято решение выполнить массовую *нисходящую денормализацию*. В этом случае следует создавать таблицы с максимальным для СУБД числом столбцов, так как любое другое решение, вероятно, обусловит необходимость массовых соединений "один к одному".

Согласно мнению известного специалиста в области проектирования реляционных БД Д. Энсора, "хорошей мерой степени нормализации является число столбцов на таблицу. Эмпирическое правило гласит, что "очень немногие первичные ключи имеют более двадцати действительно зависимых от них атрибутов".

**Денормализация методом слияния таблиц**

***Денормализация методом слияния таблиц*** – это процесс объединения двух или более нормализованных таблиц с целью устранения операций соединений или уменьшения в некоторых случаях числа операций вставки.

Возникает резонный вопрос: зачем после того, как вложено столько сил в нормализацию структур данных и разбивку сущностей, предлагается начать слияние таблиц? Фактически, слияние оправдано в очень немногих случаях.

Один из примеров обоснованного применения слияния — наличие повторяющейся группы, которая гарантированно состоит из фиксированного числа элементов. Хорошими кандидатами на такое объединение являются таблицы со строкой для каждого месяца года или каждого дня недели. Единственный случай, когда фиксированные группы надежны,— это когда они соответствуют абсолютно постоянным вещам, например, дням недели.

Будет ли какое-либо преимущество от такого слияния, заранее сказать трудно. Подход к *денормализации* должен определяться требованиями к обработке данных.

Альтернатива данному способу *денормализации* – физическое размещение таблиц в кластере БД (как например, в СУБД семейства Oracle). Это позволяет хранить рядом строки логически связанных отдельных таблиц.

Таким образом, на практике *денормализация* представляет собой набор приемов преобразования таблиц с целью повышения производительности обработки *запросов*. С точки зрения реляционной теории, мы как бы не принимаем в рассмотрение вынесение некоторых функциональных зависимостей в отдельные сущности-таблицы. Основным критерием проведения *денормализации* нормализованных таблиц являются требования к обработке данных. В реляционных БД следует избегать неоправданной *денормализации* таблиц.

**Методы разбиения таблиц**

**Разбиение таблиц базы данных**

**Разбиение таблиц** (splitting partition) является одним из общих методов *денормализации*, который применяется в физическом проектировании ХД. *Разбиение таблиц* бывает двух видов – *вертикальное разбиение* и *горизонтальное разбиение*.

***Вертикальное разбиение*** (vertically partition) **является процессом перемещения некоторых колонок таблицы в другую новую таблицу, которая имеет тот же первичный ключ, что и исходная таблица**.

***Горизонтальное разбиение*** (horizontally partition) **является процессом перемещения некоторых строк одной таблицы в другую новую таблицу, которая имеет такую же внутреннюю структуру, что и исходная таблица**.

Основные причины *разбиения таблицы*: либо существуют определенные проблемы с производительностью *запросов* на этой таблице, либо непересекающиеся подмножества строк таблицы имеют значительное различие в производительности обработки, т.е. подмножества строк таблицы редко встречаются в одном и том же *запросе*.

**Вертикальное разбиение таблиц**

Проблемы с производительностью выполнения *запросов* на длинных строках таблицы являются наиболее частой причиной для выполнения *вертикального разбиения* таблицы. Критериями того, что строка считается длинной, может быть следующее:

* длина строки больше, чем длина физической страницы базы данных (>1 КБ);
* использование так называемого индекса хеширования (cluster hashed index).

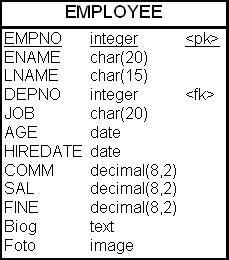
Рассмотрим случай, когда длина строки больше размера физической страницы БД. Здесь либо таблица имеет слишком много колонок, либо некоторые колонки имеют большую длину. В этом случае СУБД при вставке строки в таблицу будет использовать одну или более дополнительных физических страниц для сохранения строки. Следовательно, при выборке строки из таблицы потребуется больше операций ввода-вывода на число дополнительных физических страниц. Производительность *запроса* при этом будет ухудшаться. Для увеличения производительности выборки можно разбить таблицу на одну или несколько таблиц с длиной строки подходящего размера.

Метод *вертикального разбиения* принципиально прост, если вспомнить, что разбиение эквивалентно *реляционной операции* проекции на таблице. Ясно, что некоторые колонки просто переносятся в новую таблицу так, чтобы длина оставшейся строки была подходящей (< 1 КБ). Разбиение не должно нарушать функциональных зависимостей между колонками. Поскольку мы предполагаем, что исходная таблица нормализована (в частности, все неключевые колонки функционально полно зависят от первичного ключа), первичный ключ новой таблицы является точной копией первичного ключа исходной таблицы.

Критичным вопросом является вопрос, какие колонки следует переносить в новую таблицу. При *разбиении таблицы* увеличение производительности будет достигаться только при раздельном доступе к полученным таблицам. При соединении полученных таблиц производительность такого *запроса* может быть ниже, чем с тем же *запросом* к исходной таблице, из-за дополнительного ввода-вывода. Поэтому, до того как принять решение о *разбиении таблицы*, необходимо проанализировать все *транзакции* к исходной таблице с высоким приоритетом и при разнесении колонок по таблицам руководствоваться принципом: частота совместного использования колонок в *запросах*, размещенных в каждой таблице разбиения, должна быть достаточно высока.

**Пример 19.5**.

Предположим, что в таблице "Служащие" (EMPLOYEE) необходимо дополнительно сохранять фотографию сотрудника и его автобиографию ([рис. 19.6](https://intuit.ru/studies/courses/599/455/lecture/10180?page=3#image.19.6)). Эти два новых поля имеют достаточно большой размер, и длина строки таблицы заведомо превысит 1 КБ.



**Рис. 19.6.**Таблица "Служащие" (EMPLOYEE)

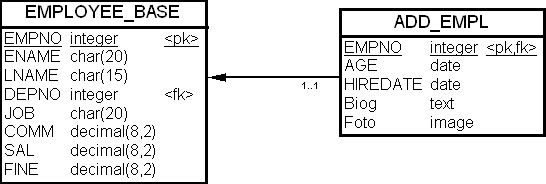
Далее предположим, что существует 60 *транзакций*, которые обращаются к этой таблице. Только четыре из них обращаются ко всем колонкам: при вводе данных о сотруднике при приеме на работу, при внесении изменений, при удалении информации о сотруднике в связи с его увольнением и *запрос* руководителя, который имеет высокий приоритет. Все *транзакции*, кроме одной, указанной выше, имеют средний и низкий приоритеты. Частота *транзакции* с высоким приоритетом ожидается не превышающей 2 раз в неделю. Поэтому разбиение таблицы на две не сильно повлияет на *производительность транзакций* с высоким приоритетом в базе данных в целом.

Частота использования полей в *транзакциях* приведена в [табл. 19.1](https://intuit.ru/studies/courses/599/455/lecture/10180?page=3#table.19.1).

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 19.1. Таблица частот использования полей таблицы "Служащие" (EMPLOYEE) | | | |
| **№** | **Наименование атрибута** | **Наименование колонки** | **Частота использования полей в транзакциях** |
| 1 | Номер личной карточки | EMPNO (PK) | 60 |
| 2 | Фамилия | ENAME | 60 |
| 3 | Имя | LNAME | 50 |
| 4 | Номер подразделения | DEPNO | 50 |
| 5 | Должность | JOB | 20 |
| 6 | Дата рождения | AGE | 4 |
| 7 | Стаж | HIREDATE | 4 |
| 8 | Доплаты | COMM | 50 |
| 9 | Зарплата | SAL | 50 |
| 10 | Штрафы | FINE | 50 |
| 11 | Автобиография | Biog | 4 |
| 12 | Фотография | Foto | 4 |

[Табл. 19.1](https://intuit.ru/studies/courses/599/455/lecture/10180?page=3#table.19.1) не содержит частоты совместного использования колонок в *транзакциях*, но из частот использования полей в *транзакциях* можно сделать вывод о совместном использовании колонок. Вероятнее всего, колонки, имеющие близкие значения частот использования, используются и совместно.

Таким образом, имеется основание для принятия решения о *разбиении таблицы* "Служащие" (EMPLOYEE) на две — скажем, "Служащие" (EMPLOYEE\_BASE) и "Дополнительные данные" (ADD\_EMPL), как показано на [рис. 19.7](https://intuit.ru/studies/courses/599/455/lecture/10180?page=3#image.19.7).



**Рис. 19.7.**Вертикальное разбиение таблицы "Служащие" (EMPLOYEE) на две таблицы: "Служащие" (EMPLOYEE\_BASE) и "Дополнительные данные" (ADD\_EMPL)

Последовательность команд SQL для создания *вертикального разбиения* таблицы "Служащие" приведена ниже.

create table ADD\_EMPL (

EMPNO integer not null,

AGE date null,

HIREDATE date not null with default,

Biog text null,

Foto image null,

constraint PK\_ADD\_EMPL primary key (EMPNO)

)

go

create table EMPLOYEE\_BASE (

EMPNO integer not null,

ENAME char(20) null,

LNAME char(15) null,

DEPNO integer null,

JOB char(20) null,

COMM decimal(8,2) null,

SAL decimal(8,2) null,

FINE decimal(8,2) null,

constraint PK\_EMPLOYEE\_BASE primary key (EMPNO)

)

go

alter table ADD\_EMPL

add constraint FK\_ADD\_EMPL\_REFERENCE\_EMPLOYEE foreign key (EMPNO)

references EMPLOYEE\_BASE (EMPNO)

go

alter table EMPLOYEE\_BASE

add constraint FK\_EMPLOYEE\_REFERENCE\_DEPARTAMENT (DEPNO)

references DEPARTAMENT (DEPNO)

go

Мы определили ограничение ссылочной целостности между таблицами "Служащие" (EMPLOYEE\_BASE) и "Дополнительные данные" (ADD\_EMPL) с помощью команды ALTER TABLE, поэтому она будет поддерживаться встроенными механизмами СУБД MS SQL Server 2008.

Помимо повышения производительности операций выборки в качестве обоснования *вертикального разбиения* таблиц следует указать на повышение производительности при операциях вставки и удаления, а также при операциях резервного копирования и восстановления БД или ХД.

**Длинные строки в таблицах хеширования**

Во многих реляционных СУБД поддерживаются так называемые **хеш-кластерные индексы** (clustered hashed index). Такие объекты правильнее называть *таблицами хеширования*, а не индексами. ***Таблица хеширования*** представляет собой таблицу реляционной БД, доступ к строкам которой осуществляется с помощью преобразования ключа. Значения колонок, которые объявлены ключевыми, преобразуются в позиции строк таблицы (и при их вставке там и размещаются) – хешируются. Такую функцию называют хеш-функцией. Ключ таблицы, который подвергается преобразованию, называется хеш-ключом.

Данные, которые обрабатываются таким образом, размещаются в специальных таблицах, называемых еще *хеш-кластерами* или просто хеш-таблицами. В настоящей лекции предполагается, что проектировщику известны общие методы организации физического доступа к данным, поэтому мы не будем детально обсуждать вопрос, как устроены такие таблицы.

Отметим, что хеширование является очень эффективным методом доступа по первичному ключу к записи за один доступ. Если значения ключа равномерно распределены, то в среднем это будет так. В противном случае производительность доступа будет резко падать из-за коллизий — случаев, когда для двух различных значений первичного ключа хеш-функция дает одинаковые числа, т.е. позиции записи. В худшем случае придется просканировать всю таблицу, чтобы получить одну запись.

Несмотря на то, что построены динамические *таблицы хеширования* (изменяющие свой размер во время существования), в большинстве СУБД поддерживаются статические *таблицы хеширования*, размер которых определяется при их создании. В большинстве случаев *таблица хеширования* формируется случайным образом по отношению к порядку следования значений ключа, хотя известны функции преобразования ключа, которые поддерживают *лексикографический порядок* на значении ключа таблицы.

Хеш-индекс обычно применяется, если ключ полностью представлен в предложении WHERE и используется операция равенства для колонок ключа.

Нас интересует проблема увеличения производительности в хеш-таблицах, когда длина строки превышает размер физической страницы на жестком диске. Чтобы лучше понять проблему, рассмотрим, как определяется хеш-таблица в SQL.

Такая таблица создается при помощи команды, например (как в СУБД SQLBase):

CREATE CLUSTERED HASHES INDEX CHXNAME ON EMPLOYEE

(EMPNO) SIZE 2000 ROWS;

Предложение SIZE задает вероятное количество строк в индексе, а ROWS определяет число строк для хранения индекса. Размер можно задавать в блоках ( *BUCKETS* ). Таким образом, по значению первичного ключа адресуется блок, содержащий целое число строк, или строка, если ее размер сопоставим с размером физического блока. В последнем случае считается, что блок содержит одну строку.

Для *таблицы хеширования* определяется параметр "число строк на странице" (rows per page), или кластеризация страницы (page clustering)), или коэффициент блокировки, равный

\mbox{blocking\_factor}=\frac{\mbox{pagesize}}{\mbox{rowsize}}

Как видно из определения *таблицы хеширования*, размер строки и размер физического блока должны быть согласованы. Проблема длинной строки в *таблице хеширования* состоит в том, что если строка занимает несколько блоков, то возрастает частота коллизий и вместо одного физического доступа для получения строки требуется 4-6, что уже сопоставимо с использованием индексов другого типа.

Цель разбиения *таблицы хеширования* состоит в том, чтобы попытаться достигнуть такого значения коэффициента блокировки ( *blocking factor* ), при котором таблица содержала бы как можно больше строк, выбирающихся вместе в большинстве *транзакций* к этой таблице.

Поскольку в современных СУБД размер физического блока фиксирован для каждой операционной платформы, единственным способом влияния на величину этого параметра является подгонка размера строки. Если нельзя пересмотреть *спецификацию типов* колонок таблицы и свести их размеры до минимума, то единственной возможностью выбрать подходящий коэффициент блокировки является разбиение *таблицы хеширования*.

В СУБД семейства MS SQL Server *таблицы хеширования* не поддерживаются в явном виде. Однако они могут быть созданы с помощью функции *CHECKSUM*() и вычисляемой колонки в индексе. Индекс должен быть неуникальным, чтобы было возможно разрешать коллизии хеш-функции.

#### Горизонтальное разбиение таблиц

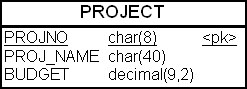
На практике *горизонтальное разбиение* (horizontally partition) применяется для изоляции одной группы строк таблицы от другой, когда использование этих групп строк в *транзакциях* почти не пересекается. Типичный пример — изоляция текущих данных от архивных данных.

Рассмотрим систему обработки заказов. Менеджеры и продавцы работают с текущими заказами. Обработка выполненных заказов (архивные данные) выполняется при подготовке разного рода отчетов (в частности, путем создания *киоска данных*). Даже если готовится ежедневный отчет с обращением к архивным данным, то в организациях среднего размера частота использования текущих данных все равно превышает частоту использования архивных данных на 2-3 порядка, а отношение объема текущих данных к архивным данным может составлять менее 0.001.

Одним из практических критериев в данном случае может служить классическое правило 80-20. Если активно работают с 20-ю процентами данных, то вероятнее всего, остальные 80% можно перенести в архивную таблицу.

**Пример 19.6**.

Таблицей – кандидатом на *горизонтальное разбиение* является таблица "Проект" (PROJECT), структура которой показана на [рис. 19.8](https://intuit.ru/studies/courses/599/455/lecture/10180?page=4#image.19.8). В этой таблице хранятся архивные данные – выполненные проекты.



**Рис. 19.8.**Таблица "Проект" (PROJECT)

Предположим, что число выполненных проектов в год в организации где-то около 1000. Данные в таблицы нужно хранить 10 лет (10000 записей). Средняя продолжительность проекта равна двум месяцам, т. е. число незавершенных проектов в данный момент времени не превышает 200. Через 5 лет отношение числа текущих проектов к архивным проектам достигнет 0.04.

Следовательно, можно рассмотреть вопрос о *горизонтальном разбиении* этой таблицы и выполнить его, как показано на [рис. 19.9](https://intuit.ru/studies/courses/599/455/lecture/10180?page=4#image.19.9).



**Рис. 19.9.**Горизонтальное разбиение таблицы "Проект" (PROJECT) на две таблицы: "Текущие проекты" (PROJECT\_CUR) и "Архивные проекты" (PROJECT\_OLD)

Мы разбили таблицу "Проект" (PROJECT) горизонтально на две таблицы — "Текущие проекты" (PROJECT\_CUR) и "Архивные проекты" (PROJECT\_OLD). Последовательность команд для создания таблиц разбиения приведена ниже.

create table PROJECT\_CUR (

PROJNO char(8) not null,

PROJ\_NAME char(40) not null,

BUDGET decimal(9,2) not null,

constraint PK\_PROJECT\_CUR primary key (PROJNO)

)

go

create table PROJECT\_OLD (

PROJNO char(8) not null,

PROJ\_NAME char(40) not null,

BUDGET decimal(9,2) not null,

constraint PK\_PROJECT\_OLD primary key (PROJNO)

)

go

Для совместного использования двух этих таблиц можно предусмотреть представление "Все проекты" ( ALL\_PROJECT ), которое показано на [рис. 19.10](https://intuit.ru/studies/courses/599/455/lecture/10180?page=4#image.19.10).



**Рис. 19.10.**Представление "Все проекты" (ALL\_PROJECT)

Команда SQL для создания представления "Все проекты" ( ALL\_PROJECT ) приведена ниже.

CREATE VIEW ALL\_PROJECT

AS

SELECT PROJNO, PROJ\_NAME, BUDGET FROM PROJECT\_CUR

UNION

SELECT PROJNO, PROJ\_NAME, BUDGET FROM PROJECT\_OLD;

Заметим, что далее по тексту под исходной таблицей понимается и сама таблица, и то, во что она превратилась после разбиения.

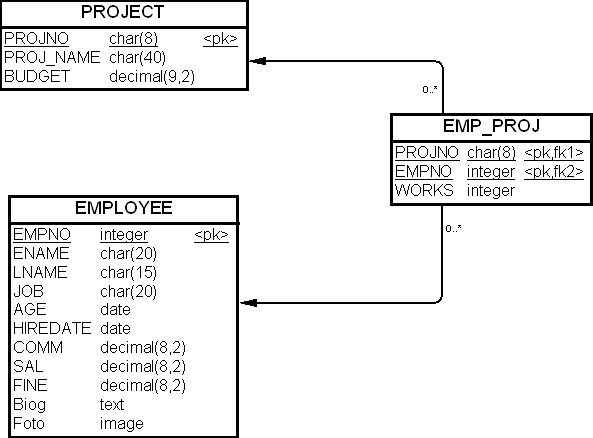
**Разбиение таблиц и ссылочная целостность**

Если до *разбиения таблица* была нормализована, то первичные ключи будут идентичны для таблиц, полученных в результате разбиения. Следовательно, взаимосвязи новых таблиц с другими таблицами базы данных могут остаться такими же, как и с первоначальной таблицей. Однако необходимо рассмотреть и действие правил удаления для исходной таблицы, и участие новой таблицы в существующих взаимоотношениях ссылочной целостности, когда будет решаться вопрос об определении *поддержки ссылочной целостности* в новых таблицах.

В некоторых случаях может быть проще не определять ссылочную целостность вовсе, так как фактическое удаление строк из новой таблицы будет поддерживаться параллельно с исходной таблицей каким-либо программным способом в приложении БД (например, с помощью триггера).

**Пример 19.7**.

Для разрешения отношения "многие ко многим" между таблицами "Служащий" (EMPLOYEE) и "Проект" (PROJECT) была введена связывающая таблица "Служащий Проект" (EMP\_PROJ), которая имеет ограничения ссылочной целостности, с таблицей "Проект" (PROJECT), как показано на [рис. 19.11](https://intuit.ru/studies/courses/599/455/lecture/10180?page=5#image.19.11).



**Рис. 19.11.**Разрешение отношения "многие ко многим" между таблицами "Служащий" (EMPLOYEE) и "Проект" (PROJECT)

Связывающая таблица "Служащий Проект" (EMP\_PROJ) была создана с помощью команды SQL, приведенной ниже.

create table EMP\_PROJ (

PROJNO char(8) not null,

EMPNO integer not null,

WORKS integer null,

constraint PK\_EMP\_PROJ primary key (PROJNO, EMPNO).

constraint FK\_EMP\_PROJ\_REFERENCE\_PROJECT foreign key (PROJNO)

references PROJECT (PROJNO),

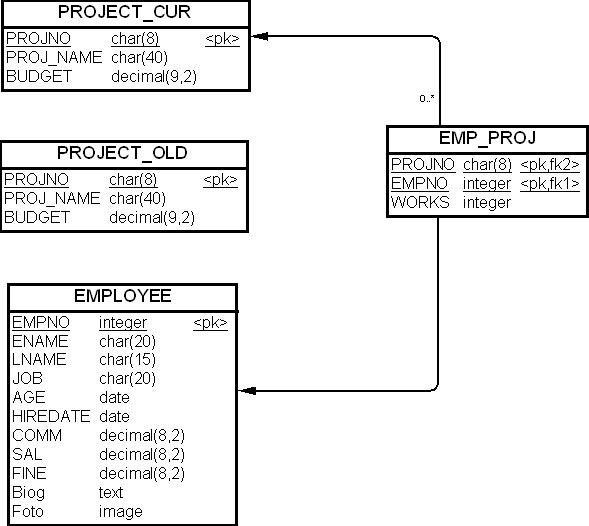
constraint FK\_EMP\_PROJ\_REFERENCE\_EMPLOYEE foreign key (EMPNO)

references EMPLOYEE (EMPNO)

)

go

Но у нас теперь, после *разбиения таблицы* "Проект" (PROJECT), появились две новые таблицы: "Текущие проекты" (PROJECT\_CUR) вместо таблицы "Проекты" (PROJECT) и "Архивные проекты" (PROJECT\_OLD), как показано на [рис. 19.12](https://intuit.ru/studies/courses/599/455/lecture/10180?page=5#image.19.12).

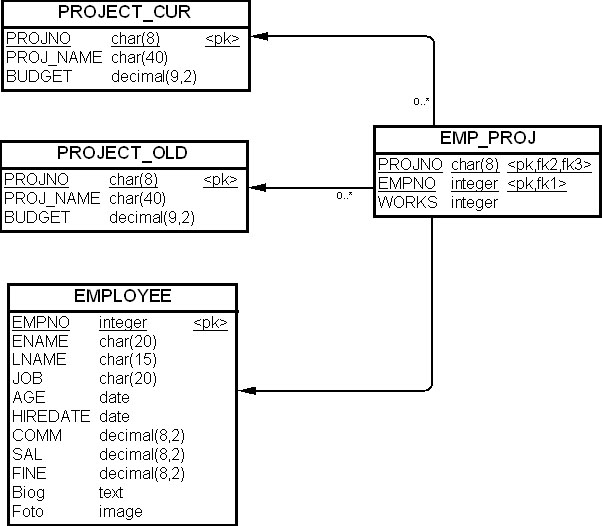


**Рис. 19.12.**Разрешение отношения "многие ко многим" после разбиения таблицы "Проекты" (PROJECT)

Предположим, что для реализации проекта организация нанимает сотрудников на временной основе (на время выполнения проекта), и вопрос: кто, какие проекты, когда выполнял – не интересует руководство организации. В этом случае взаимосвязь между исполнителями и проектами, которые уже завершены, не интересует руководство, и следовательно, в *физической модели данных* ничего менять не нужно.

Однако если взаимосвязь между исполнителями и завершенными проектами должна отслеживаться (например, руководство будет изучать вопрос: кто, когда, какой проект выполнял), ее следует распространить и на таблицу "Архивные проекты" (PROJECT\_OLD), как показано на [рис. 19.13](https://intuit.ru/studies/courses/599/455/lecture/10180?page=5#image.19.13).

Чтобы учесть в БД проделанную нами работу, достаточно внести ограничение внешнего ключа в таблицу "Служащий Проект" (EMP\_PROJ), как показано ниже:



**Рис. 19.13.**Разрешение отношения "многие ко многим" таблицы "Архивные проекты" (PROJECT\_OLD)

drop table EMP\_PROJ

Go

create table EMP\_PROJ (

PROJNO char(8) not null,

EMPNO integer not null,

WORKS integer null,

constraint PK\_EMP\_PROJ primary key (PROJNO, EMPNO).

constraint FK\_EMP\_PROJ\_REFERENCE\_PROJECT foreign key (PROJNO)

references PROJECT\_CUR (PROJNO),

constraint FK\_EMP\_PROJ\_REFERENCE\_PROJECT\_OLD foreign key (PROJNO)

references PROJECT\_OLD (PROJNO),

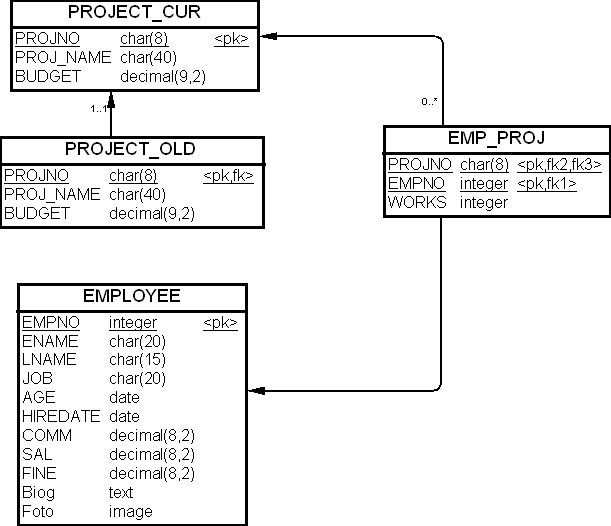
constraint FK\_EMP\_PROJ\_REFERENCE\_EMPLOYEE foreign key (EMPNO)

references EMPLOYEE (EMPNO)

)

go

При *разбиении таблицы*, для которой были определены ограничения ссылочной целостности, можно также рассмотреть возможность *поддержки ссылочной целостности* установлением взаимосвязи "один к одному" в сторону исходной таблицы от новой таблицы. В этом случае новая таблица будет иметь внешний ключ, идентичный первичному ключу таблицы. Например, связь "один к одному" между таблицами на [рис. 19.14](https://intuit.ru/studies/courses/599/455/lecture/10180?page=5#image.19.14).



**Рис. 19.14.**Взаимосвязь "один к одному" между таблицами "Текущие проекты" (PROJECT\_CUR) и "Архивные проекты" (PROJECT\_OLD)

Если определено каскадное правило удаления для этого внешнего ключа (см. ["Знакомство с CASE инструментом"](https://intuit.ru/studies/courses/599/455/lecture/10166)), то СУБД будет автоматически удалять строки новой таблицы, когда соответствующая строка исходной таблицы будет удалена, хотя управление вставкой строк придется перенести в приложение ХД. Команда SQL для этого случая приведена ниже.

create table PROJECT\_OLD (

PROJNO char(8) not null,

PROJ\_NAME char(40) not null,

BUDGET decimal(9,2) not null,

constraint PK\_PROJECT\_OLD primary key (PROJNO),

constraint FK\_EMP\_PROJ\_REFERENCE\_PROJECT\_OLD\_1 foreign key (PROJNO)

references PROJECT\_СUR (PROJNO)

)

go

При принятии решения о *разбиении таблицы* следует придерживаться следующего алгоритма:

* определить, какие колонки исходной таблицы в какие новые таблицы будут перемещены;
* создать новые таблицы с первичным ключом, идентичным первичному ключу исходной таблицы;
* если СУБД будет управлять декларативной ссылочной целостностью для новых таблиц таким же образом, как и для исходной таблицы, в случае если она является дочерней таблицей во взаимосвязи, следует добавить колонку внешнего ключа каждой родительской таблицы во взаимосвязи в новую таблицу, т.е. новая таблица должна содержать ограничение внешнего ключа, идентичное родительской таблице, для каждой взаимосвязи. Альтернативой этому решению является создание связи "один к одному" между новой таблицей и исходной таблицей, определение внешнего ключа обратно к исходной таблице тождественным первичному ключу;
* если СУБД будет управлять ссылочной целостностью для новых таблиц таким же образом, как и для исходной таблицы, в случае если она является родительской таблицей во взаимосвязи, то следует добавить внешний ключ в каждую дочернюю таблицу исходной таблицы, чтобы идентифицировать эту новую таблицу как дочернюю. Можно так же поступить, как в первой части предыдущего пункта, при этом новые таблицы не следует объявлять как родительские в других взаимоотношениях. Исходная таблица поддерживает все взаимосвязи, в которых она выступает родителем;
* следует прописать для разработчиков приложений все команды INSERT для полученных в результате *разбиения таблиц* или указать правила, которым должна следовать вставка строк в эти таблицы;
* следует изменить все представления, которые основывались на исходной таблице, и, если нужно, рассмотреть создание новых представлений для доступа к новым таблицам.

**Объединение таблиц базы данных**

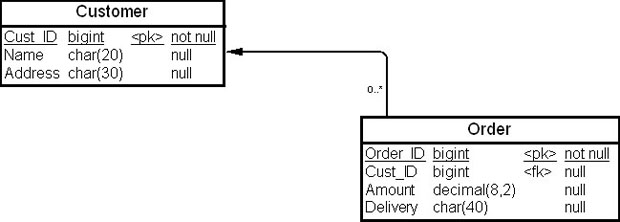
**Объединение таблиц (Table collapsing) является процессом перемещения строк нескольких таблиц в одну, новую таблицу для ограничения числа соединений таблиц БД и улучшения производительности запросов**.

Объединяющая таблица включает в себя колонки всех объединяемых таблиц. Если между таблицами были установлены ограничения ссылочной целостности, то при объединении колонки родительской таблицы дублируются, *ограничения внешнего ключа* таблиц-потомков удаляются из объединяющей таблицы и колонки, участвующие в соединении объединяемых таблиц, удаляются.

Рассмотрим примеры.

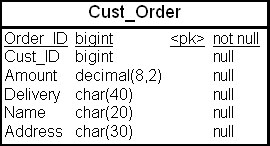
**Пример. 13.8**.

Предположим, что было принято решение об объединении таблиц "Покупатель" (Customer) и "Заказ" (Order), — их необходимо объединить, чтобы исключить операцию соединения в *запросах* к этим таблицам. *Физическая модель данных* до объединения приведена на [рис. 19.15](https://intuit.ru/studies/courses/599/455/lecture/10180?page=5#image.19.15).

[](https://intuit.ru/EDI/20_07_20_2/1595197216-9970/tutorial/632/objects/19/files/13_15.jpg)

[увеличить изображение](https://intuit.ru/EDI/20_07_20_2/1595197216-9970/tutorial/632/objects/19/files/13_15.jpg)  
**Рис. 19.15.**Таблицы "Покупатель" (Customer) и "Заказ" (Order) до их объединения

В результате объединения таблиц будет создана одна таблица "Покупатель Счет" (Cust\_Order), содержащая все колонки объединяемых таблиц ([рис. 19.16](https://intuit.ru/studies/courses/599/455/lecture/10180?page=5#image.19.16)).



**Рис. 19.16.**Таблица "Покупатель Счет" (Cust\_Order), объединяющая таблицы "Покупатель" (Customer) и "Заказ" (Order)

Команда SQL для создания объединяющей таблицы приведена ниже.

create table Cust\_Order (

Order\_ID bigint not null,

Cust\_ID bigint not null,

Amount decimal(8,2) null,

Delivery char(40) null,

Name char(20) null,

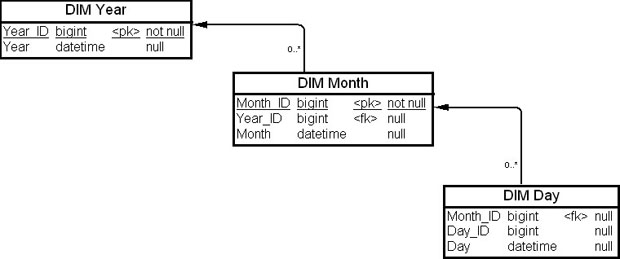
Address char(30) null,

constraint PK\_CUST\_ORDER primary key (Order\_ID)

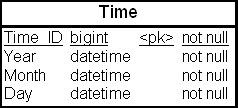
)

go

Для ХД целесообразно рассматривать объединение нормализованных таблиц иерархии — например, таблиц измерений, представляющих измерение "Время" ( **Time** ), как на [рис. 19.17](https://intuit.ru/studies/courses/599/455/lecture/10180?page=5#image.19.17). Денормализованная таблица измерений для иерархии "Время" приведена на [рис. 19.18](https://intuit.ru/studies/courses/599/455/lecture/10180?page=5#image.19.18).

[](https://intuit.ru/EDI/20_07_20_2/1595197216-9970/tutorial/632/objects/19/files/13_17.jpg)

[увеличить изображение](https://intuit.ru/EDI/20_07_20_2/1595197216-9970/tutorial/632/objects/19/files/13_17.jpg)  
**Рис. 19.17.**Иерархия таблиц измерения "Время"



**Рис. 19.18.**Денормализованная таблица иерархии "Время"

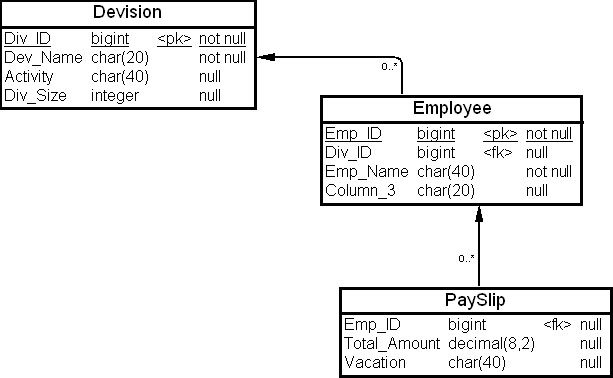
**Денормализация колонок**

***Денормализация колонок*** (*Column* *denormalization*) **является процессом для ограничения числа часто встречающихся соединений таблиц БД и улучшения производительности запросов**.

Рассмотрим следующий пример.

**Пример 19.9**.

Пусть задана *физическая модель данных* ([рис. 19.19](https://intuit.ru/studies/courses/599/455/lecture/10180?page=5#image.19.19)).



**Рис. 19.19.**Таблицы физической модели данных

Допустим, что в отчете необходимо напечатать колонки "Название отдела" (Div\_name) на платежной расписке из таблицы "Платежная расписка" (Pay *Slip*) каждого служащего. Для решения этой задачи можно рассмотреть вопрос о *денормализации колонок* таблицы, чтобы иметь колонку "Название отдела" (Div\_Name) в таблице "Платежная расписка" (Pay *Slip*). Перенесем для этого колонку "Название отдела" (Div\_Name) ( *нисходящая денормализация* ) в таблицу "Платежная расписка" (Pay *Slip*). *Физическая модель данных* таблицы "Платежная расписка" (Pay *Slip*) приведена на [рис. 19.20](https://intuit.ru/studies/courses/599/455/lecture/10180?page=5#image.19.20).



**Рис. 19.20.**Нисходящая денормализация таблицы "Платежная расписка" (Pay Slip)

Таким образом, мы сократим число соединений при совместном использовании в отчетах колонок таблицы "Платежная расписка" (Pay *Slip*) и колонки "Название отдела" (Div\_Name).

**Резюме**

В настоящей лекции мы рассмотрели процесс *денормализации* таблиц физической *БД* на уровне моделирования *физической модели данных*, как для *БД*, так и для ХД.

*Денормализация* представляет собой набор методов настройки физической модели *БД*, используемой для реализации ХД, с целью удовлетворения требований к производительности ХД. Эти методы представляют собой набор рекомендаций и эвристических правил по изменению физической структуры *БД*, которая была получена в результате первой итерации создания *физической модели данных* ХД. Применение этих методик носит рекомендательный характер.

В этой лекции были описаны различные типы *денормализации* и методы ее реализации. Кроме того, было рассмотрено, как при *денормализации* обеспечить *целостность* данных, не прибегая к созданию дополнительного кода.

Таким образом, под **денормализацией понимают процесс достижения компромиссов в нормализованных таблицах посредством намеренного введения избыточности в целях увеличения производительности**.

В большинстве случаев необходимость *денормализации* становится очевидной лишь на этапе проектирования приложений ХД или его эксплуатации. Другими словами, нельзя принять решение о *денормализации* на основании одной только модели данных. Обычно стараются найти в приложениях ХД критичные процессы и принимать решения о *денормализации* в основном в пользу этих процессов. Критичные процессы, как правило, определяют по высокой частоте, большому объему, высокой изменчивости или явному приоритету. Качественное описание *транзакций* *БД* позволяет определить наличие таких критических процессов.