Как решать задачи на SQL

Сергей Минюров *Москва, Июнь 2016*

SQL это фантастический по своей логике и возможностям язык, который идеально подходит для бизнес-программирования. В сравнении с другими языками программирования, например, C или Java, он кажется простым, но на нем можно решать самые сложные логические задачи, которые даже в голову не придут при программировании на других языках. При этом, решение на SQL будет проще и производительнее. Давайте разберемся, почему этот относительно простой язык программирования является таким мощным и компактным.

SQL как язык программирования

SQL является декларативным языком (4-го поколения) для обработки данных с полной комбинаторикой. **Декларативность** значит, что нам не приходится тратить свои ресурсы на технические детали решения, а мы сразу же можем логически решать задачу. На обычных языках программирования прежде чем мы доберемся до сути задачи нам нужно написать массу технического кода, не имеющего к ней непосредственного отношения. А это серьезно нагружает мышление программиста и не позволяет полностью сосредоточиться на самой задаче.

> Оказывается, при программировании именно данные являются главным компонентом в решении, а алгоритмы и архитектура зависят от данных (Программирование, управляемое данными). Программистов сначала учат как разрабатывать алгоритмы и проектировать классы. С опытом приходит понимание важности именно структур данных, поскольку именно от них зависит сложность и гибкость алгоритмов.

Полная комбинаторика означает, что любое простое выражение на SQL может быть преобразовано в более сложное в соответствии с логикой данных и логикой самой задачи. Например, когда указываем поле или скалярное значение в SELECT, WHERE, ORDER BY и т.д., то можно, как в математике, заменить значение на выражение, скалярный подзапрос или функцию. Аналогично, таблицу во FROM можно заменить на табличный запрос, представление или функцию.

На следующем рисунке показывается базовый синтаксис SELECT как основной инструкции SQL, который используется в задачах среднего уровня сложности.

```
(5) SELECT (6) DISTINCT (8) TOP { <Поле> | <Выражение> | <Скалярный запрос> |
<Скалярная функция> }
(1) FROM { <Таблица> | <Представление> | <Табличный запрос> | <Табличная функция> }
[ { INNER | { LEFT | RIGHT [OUTER] }} JOIN { <Таблица> | <Представление> |
<Табличный запрос> | <Табличная функция> }
ON <Логическое выражение – критерий соединения> ]
[ CROSS JOIN { <Таблица> | <Представление> | <Табличный запрос> |
<Табличная функция> } ]
[{ OUTER | CROSS } APPLY { <Табличный запрос> | <Табличная функция> }]
(2) WHERE <Логическое выражение – критерий выборки до группировки>
(3) GROUP BY { <Поле> | <Выражение> | <Скалярный запрос> | <Скалярная функция> }
(4) HAVING { <Логическое выражение – критерий выборки после группировки> }
(7) ORDER BY { <Поле> | <Выражение> | <Скалярный запрос> | <Скалярная функция> }
```

Комбинаторика языка позволяет нам управлять сложностью задачи, разбивая ее на подзадачи. **Декомпозиция задачи** является очень важной техникой решения, поскольку сложные задачи «в лоб» не решаются (поэтому они и называются сложными). При анализе данных и требований мы понимаем структуру задачи и можем сначала решить ее по частям, а потом объединить их в общий сложный запрос.

Для понимания SQL можно использовать метафору «**конвейер данных**«: сам запрос можно представить как «трубу», в которой есть данные на входе и на выходе. На входе данные фильтруются с помощью WHERE: это важная операция с точки зрения производительности. Также эти данные могут быть преобразованы в самой «трубе» с помощью GROUP BY и выражений. На выходе мы получаем требуемое представление данных с помощью SELECT. При необходимости упорядочиваем данные с помощью ORDER BY.

При использовании скалярных и табличных подзапросов получается составная «труба» состоящая из нескольких частей и соединений. При написании запросов важно абстрагироваться от таблиц данных, и мыслить множествами данных и их преобразованиями.

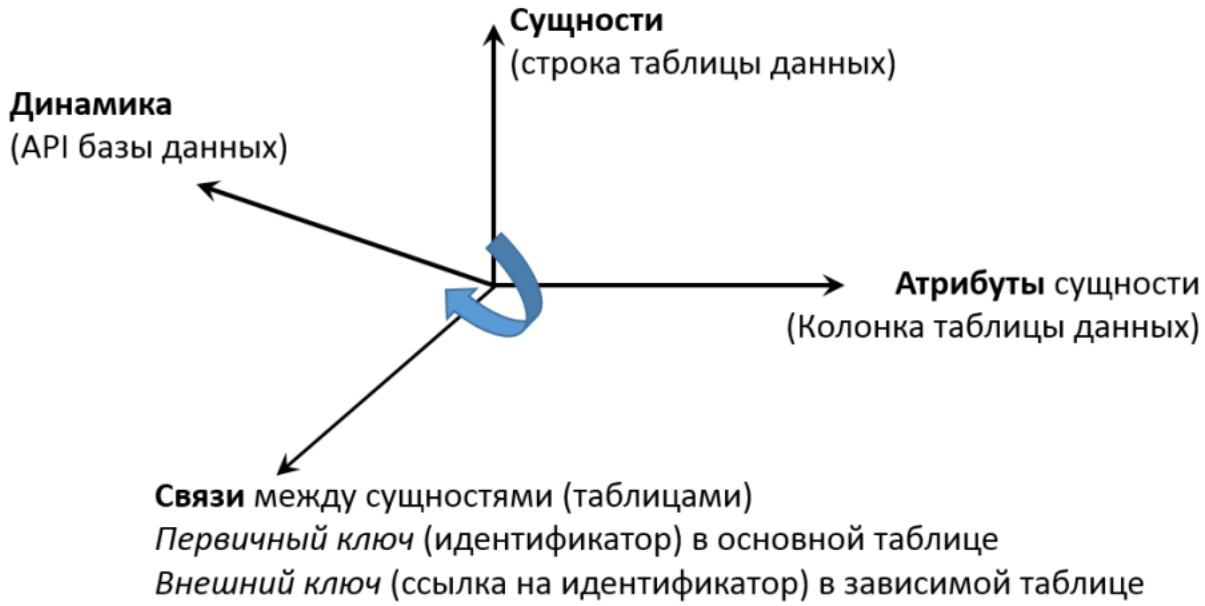
Понимание данных

Решение задачи состоит из двух аспектов: анализ данных и предметная логика самой задачи (независимая от базы данных). Недостаточное понимание любого из этих аспектов приводит к ошибкам или неоправданному усложнению решения.

Прежде всего важно понять, как спроектирована база данных: для этого используются концептуальные и физические диаграммы, на которых показываются таблицы, поля и связи. Также важно изучить API базы данных (прикладной программный интерфейс): представления, функции и хранимые процедуры — поскольку эти компоненты содержат базовые решения.

Качество проектирования базы данных является критическим фактором для решения задач и производительности системы. Опытный проектировщик сделает простую и понятную схему данных, в которой можно разобраться без документации. Это позволяет быстро решать задачи, избегая ненужных ошибок, создавая простой и производительный код на SQL.

Структурирование данных → Совместный доступ → Качество данных → Представления данных → Автоматизация обработки



Проектирование базы данных - анализ данных | Запросы к базе данных – синтез данных

Если база данных спроектирована в стиле «так получилось», то вместо четкой логической структуры данных мы имеем информационную «свалку» в виде большого количества несвязанных, дублируемых и непонятных данных. В таком случае решать задачи будет намного сложнее и неизбежно будут возникать проблемы с ошибками в самих данных и производительностью выполнения запросов.

С экономической точки зрения в этом случае необходимо сделать**рефакторинг** базы данных, иначе все разработчики и пользователи будут постоянно тратить значительное время на преодоление ненужных сложностей. И, самое главное, очень трудно в такой базе обеспечить **качество данных**.

Для решения задачи вначале находим необходимые данные и определяем, как мы их можем получить. В простых случаях соединяем данные с помощью JOIN (реляционное соединение). Если связи данных более сложные и динамические, то применяем APPLY (итеративное соединение). А если данные разбросаны по таблицам и базам данных, то используем UNION, чтобы собрать их в общее множество.

Какие данные нужны для решения задачи?

FROM – JOIN – APPLY - UNION

Как отфильтровать только необходимые данные?

SELECT – TOP – DISTINCT – WHERE – GROUP BY – HAVING – EXCEPT - INTERSECT

Как получить необходимое представление данных?

SELECT – CASE – OVER – PIVOT – UNPIVOT – ORDER BY

Разобраться с исходными данными и понять какие данные нужны для решения задачи относительно несложно. Следующая задача — получение только необходимых данных является самой важной и, зачастую, сложной задачей. Корректная выборка данных, с учетом размножения данных при соединении, является критическим фактором для получения правильного результата и нахождения производительного решения. В некоторых случаях для корректной фильтрации данных нужно использовать вместо JOIN подзапросы или APPLY.

> С помощью SELECT выбираем только нужные столбцы (фильтрация по колонкам), а с помощью WHERE выбираем только нужные данные (фильтрация по строкам). DISTINCT позволяет убирать дублирование строк (а не первой колонки!).
>
> TOP используется при рейтингах, которые строятся на основе упорядочивания данных с помощью ORDER BY и позволяет нам получить самые интересные данные по количеству строк или в процентах.
>
> С помощью GROUP BY одновременно происходит сжатие данных (аналогично DISTINCT) с разделением по группам, при этом для каждой группы с помощью агрегатных функций считаем необходимые показатели.
>
> В некоторых случаях можно отфильтровать ненужные данные очень элегантным и простым (а значит надежным) способом с помощью операторов множества EXCEPT и INTERSECT.

Для окончательного решения задачи важно сделать удобное для пользователя представление данных. В инструкции SELECT мы с помощью CASE можем реализовать сложные правила для отображения данных, сделать их более понятными для пользователя. CASE можно использовать внутри агрегатных функций.

> CASE можно использовать и в других частях запроса, например WHERE или ORDER BY. Часто с его помощью получаются красивые и мощные решения. Но важно обратить внимание на оптимизацию запроса, поскольку могут перестать работать индексы.

Для более удобного представления данных таблица может быть преобразована в матрицу с помощью PIVOT. Обратная операция выполняется с помощью UNPIVOT.

Если нужно группировать данные на разных уровнях или выполнить сложные вычисления, то используем оконные функции (OVER).

Часто пользователю удобно работать с упорядоченными данными, например, отсортированные по алфавиту или по дате. Если это не влияет на логику запроса, то лучше выполнять сортировку данных на стороне клиентского приложения. Поскольку на больших объемах данных сортировка будет одной из самых «дорогих» операций, в несколько раз замедляющих выполнение запроса.

> В более редких случаях, например, для рейтингов данных требуется обязательно использовать ORDER BY при обработке данных на стороне сервера.

Методология

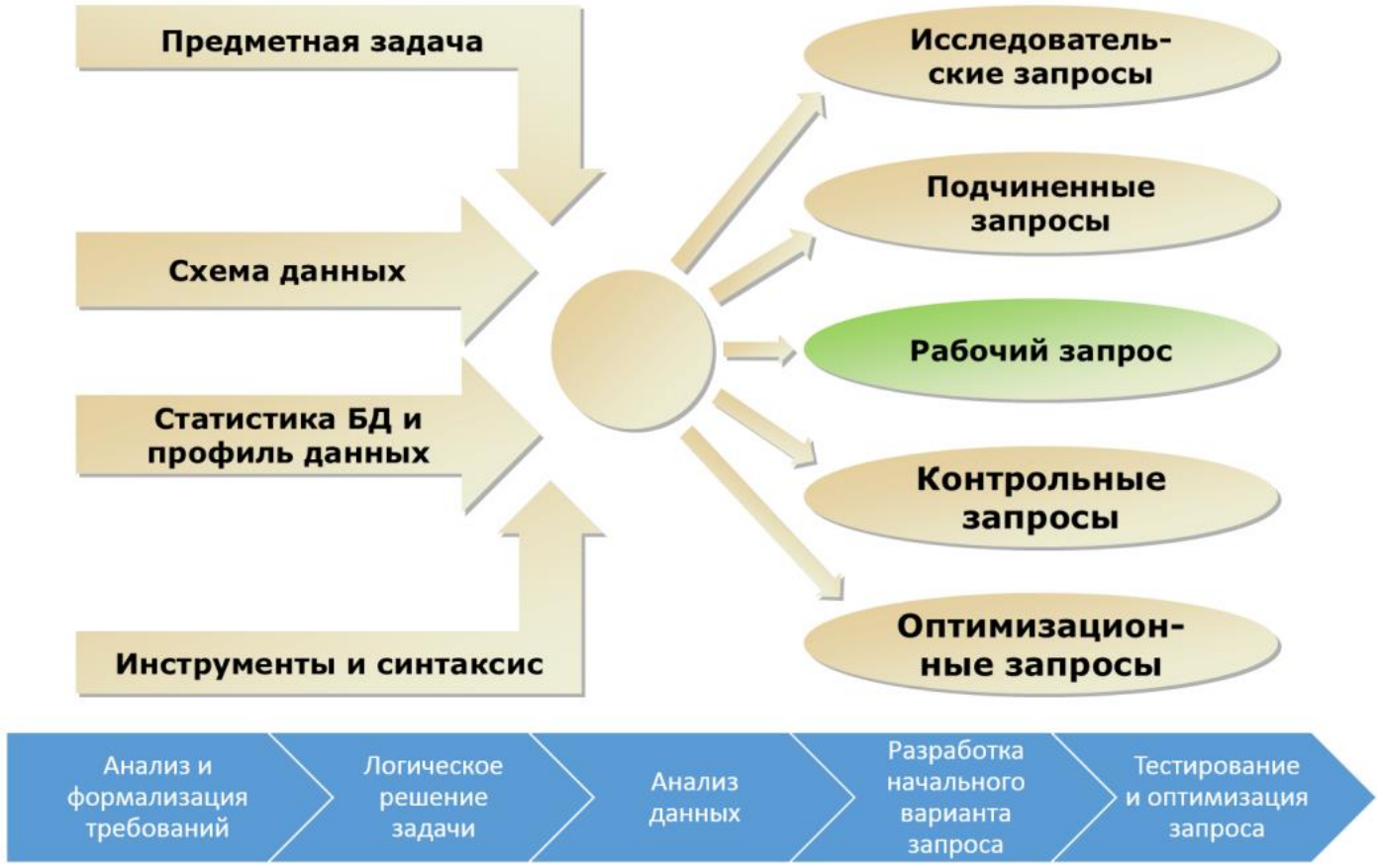
Сложность программирования на SQL заключается не в самом языке, в отличие от других языков программирования, а в необходимости постоянно заниматься анализом данных и требований в процессе решения задач.

Чем сложнее задача, тем имеется больше вариантов ее решения и тем более важным является анализ предметной логики. Не всегда по постановке задачи можно сразу же оценить ее сложность. Но если сразу же не получилось решить задачу, то стоит сделать паузу в написании SQL и выполнить ее логический анализ.

При анализе задачи лучше даже на время забыть про SQL и базу данных: важно формализовать требования и четко определить логические шаги с точки зрения предметной области, а не программирования. В простых случаях это будет просто список действий. В более сложных случаях приходится рисовать различные диаграммы в произвольном формате или, например, на UML (выбирайте удобный для себя способ).

Можно написать несколько запросов для понимания данных и как пробные варианты решения задачи. Именно в процессе решения задачи приходит ее настоящее понимание (не нужно застревать на анализе задачи).

Как в математике, вначале можно найти частное решение, которое решает задачу не полностью, но в ее главной части. А затем уже используя технику программирования преобразовать частное решение в полное конечное.



Во многих случаях конечное решение собирается из отдельных запросов, которые вначале разрабатываются отдельно как подзадачи, а потом собираются в общем запросе.

Для тестирования решения нужно написать контрольные запросы, которые позволяют проверить отдельные показатели или количество строк по отдельным наборам данных. Тестовые запросы должны быть максимально простыми (иначе высокая вероятность ошибок в самом тестовом запросе) и выполняться в другой технике программирования (иначе перенесутся ошибки из рабочего запроса).

Для «тяжелых» задач в высоконагруженных системах приходится писать несколько вариантов запросов. Поскольку сервер базы данных является динамической и открытой системой, и на нее воздействует масса факторов, влияющих на производительность, заранее очень трудно сразу же выбрать оптимальную технику. Поэтому оптимизация запросов выполняется как разработка различных вариантов и сравнительная оценки их эффективности. При этом обязательно требуется поддержка со стороны администратора баз данных.

Эвристика

Есть несколько простых приемов, как можно быстро решать задачи любой сложности, распутывая их как клубок — нужно только выбрать правильную ниточку и решать задачу последовательно, по частям.

Двигаемся от того, что знаем, к тому, чего мы пока еще не поняли — не забывайте, что глубокое понимание задачи приходит в процессе ее решения.

> Даже если мы еще не до конца поняли, как решить задачу, нужно написать запрос для понятной части задачи. Это помогает лучше понять логику данных и требования, особенности базы данных и сконцентрироваться на непонятном аспекте решения.

Базовая техника решений это использование JOIN и GROUP BY. С их помощью можно решить половину задач, которые являются простыми. Если решение сходу не получилось, то можно попробовать подзапрос. А если и это не помогло, то либо у нас редкий тип задачи, и нужно вспомнить более редкую технику вроде PIVOT, OVER или UNION. Либо нам попалась по-настоящему сложная задача, над которой придется посидеть и может потребоваться помощь более опытных коллег.

Для сложного компонента решения лучше сразу же написать частное решение, которое затем может быть использовано как подзапрос или функция.

> Также можно написать общую структуру запроса, и вначале заменить сложные подзапросы на фейковые (вроде SELECT NULL). А затем уже по мере решения подзадач последовательно их заменять на рабочие подзапросы.

Если «застряли», то лучше остановиться (выпить чай-кофе) и набросать предметную логику в виде списка шагов для решения задачи. В сложных случаях помогает рисование диаграмм. На уставшую голову писать SQL практически бесполезно — завтра его придется выбросить и написать заново.

> Для меня написание строки кода на SQL примерно в 3 раза «дороже», чем написание строки кода на C#.

Есть еще удивительно эффективный психологический прием «поговорить с кошкой» — попробовать объяснить своему коллеге задачу и варианты решения. При этом мы можем посмотреть на задачу со стороны и перестать зацикливаться на привычном, но не сработавшем подходе. И, благодаря этому, можно тут же найти решение, над которым бились долгое время.

Примеры решений

Разберем на нескольких примерах последовательность шагов для решения задач. С разрешения Федора Самородова разберем первые две задачи, придуманные им.

Задача 1

`Какие продавцы продали в 1997 году более 30 штук товара №1?`

Логика задачи. Прежде всего, нужно понять какие именно данные получаются как результат решения, а затем уже проанализировать текст задачи и выявить остальные логические компоненты задачи, такие как критерии, показатели и пр. В формализованном виде можно представить задачу в следующем виде:

- результат: продавцы
- критерий:
    - продажи за 1997 год
    - товар №1
    - более 30 штук проданного товара
        - показатель: 30 штук проданного товара

Анализ данных. Далее мы смотрим схему базы данных и отображаем на нее логические компоненты задачи. По схеме мы видим, что напрямую между продавцами (Employees) и проданными товарами (Order Details) нет прямой связи, но обе таблицы связаны с таблицей (Orders), которая также нам требуется, чтобы выбрать данные на 1997 год. Это значит, что продавцы имеют косвенную связь с проданными товарами, которая, с точки зрения SQL, принципиально не отличается от прямой связи. В формализованном виде можно представить данные для задачи в следующем виде:

- продавцы: таблица Employees
- проданные товары: таблица [Order Details]
    - критерий:
        - товар №1: ProductId = 1
        - более 30 штук проданного товара: sum(Quantity) > 30
- связь продавцов с продажами товаров: через таблицу Orders
    - критерий:
        - продажи за 1997 год: year(OrderDate) = 1997

Для простоты выведем в качестве результата колонку с фамилией продавцов (LastName). В данном случае количество колонок не влияет на решение задачи.

Поскольку в задаче у нас есть показатель, который считается с помощью агрегатной функции, то в запросе нам потребуется сгруппировать данные с помощью GROUP BY. В качестве аналитики, соответственно мы укажем колонку LastName:

```
select LastName --, sum(od.Quantity)
from Employees as e
join Orders as o on e.EmployeeID = o.EmployeeID
join [Order Details] as od on o.OrderID = od.OrderID
where od.ProductID = 1
  and year(o.OrderDate) = 1997
group by LastName
having sum(od.Quantity) > 30
```

Для отладки запроса можно в SELECT вывести показатель количество продаж товара, но по условиям задачи это не требуется, поэтому в конечном варианте его можно закомментировать или удалить.

Для этой задачи использование JOIN и GROUP BY является простой и оптимальной техникой. Давайте в учебных целях рассмотрим вариант решения с подзапросами. В новом варианте у нас будут использоваться такие же таблицы, критерии и показатель. Но по синтаксису мы заменяем каждый JOIN на подзапрос.

При использовании подзапросов можно вначале написать общую структуру основного запроса с фейковыми подзапросами (SELECT NULL), чтобы мы поняли общую логику решения:

```
select LastName
from Employees as e
where (select null) > 30
```

Принципиальным преимуществом использования подзапросов является наглядное разделение задачи на подзадачи. Соответственно, если мы смогли выполнить декомпозицию сложной задачи, то она перестала для нас быть сложной. В этом секрет быстрого решения сложных задач.

Давайте вспомним как считается показатель продажи товара — важно преобразовать условия соединения из JOIN ON в критерий выбора данных WHERE. Поскольку Order Details связан с Orders как «многие к одному» (несколько товаров в одном заказе), то нам нужен предикат IN:

```
select sum(od.Quantity)
from [Order Details] as od
where od.ProductID = 1
  -- Соединение с таблицей Orders
  and od.OrderID IN (SELECT NULL)
```

Ключевым моментом для выбора синтаксической конструкции является понимание схемы базы данных. Если мы не понимаем, как соединять подзапросы между собой, значит нужно еще раз внимательно изучить схему, и

тогда мы вспомним, что продажи товаров у нас связаны с сотрудинками через таблицу Orders, тем более, что нам еще нужно реализовать критерий по году продаж, который вычисляется из поля OrderDate этой таблицы.

С помощью подзапросов у нас каждая подзадача решается с помощью простого запроса. Нам осталось добавить в подзапрос критерий для фильтрации заказов. Для этого нам нужен еще один подзапрос 2-го уровня. При тестировании подзапроса можно написать частный случай для сотрудника №1, а в конечном решении заменить код сотрудника на поле EmployeeID из таблицы Employees. Поскольку таблица Orders связана с таблицей Employees как «один к одному» (у каждого заказа один продавец), то нам нужен не IN а операция «=»:

```
select o.OrderID
from Orders as o
where year(o.OrderDate) = 1997
  -- Соединение с таблицей Employees
  and EmployeeID = 1
```

Собираем наши простые запросы в конечный рабочий запрос:

```
select LastName
from Employees as e
where (
  select sum(od.Quantity)
  from [Order Details] as od
  where od.ProductID = 1 and od.OrderID in (
    select o.OrderID
    from Orders as o
    where year(o.OrderDate) = 1997 and e.EmployeeID = o.EmployeeID)
) > 30
```

Задача 2

Еще одна задача от Федора Самородова, как всегда, очень изящная, не столько на технику, сколько на мышление.

Для каждого покупателя (имя, фамилия) найти два товара (название), на которые покупатель потратил больше всего денег в 1997-м году.

Логика задачи. В этой задаче важно не запутаться: очевидно, какой у нас должен получиться результат, мы сразу же видим, что нам нужен показатель по продажам и критерий по году продаж. Тонкость заключается в том, чтобы понять: «два товара» это ограничение результата на основе рейтинга по сумме продаж. Формализация задачи:

- результат: покупатель (имя, фамилия), товар (название)
  - ограничение: 2 товара
- критерий:
  - продажи за 1997 год
- рейтинг:
  - максимальная сумма продаж
    - показатель: сумма продаж
    - аналитика: покупатель, товар

Анализ данных. По схеме данных понимаем, что нам нужно связать справочник заказчиков (Customers), заказы (Orders) и проданные товары (Order Details). Формализация данных:

- заказчики: таблица Customers
  - результат: имя покупателя (ContactName)
- заказы: таблица Orders
  - критерий:
    - продажи за 1997 год: year(OrderDate) = 1997
- продажа товаров: таблица Order Details
  - рейтинг:
    - максимальная сумма продаж
      - показатель: сумма продаж: sum(Quantity * UnitPrice * (1 — Discount))
      - аналитика: имя покупателя (ContactName), название товара (ProductName)
- справочник товаров: таблица Products
  - результат: название товара (ProductName)

Рейтинг данных строится с помощью ORDER BY, а ограничение реализуется с помощью TOP в SELECT. Поскольку у нас есть показатель, который мы считаем с помощью агрегатной функции sum, то нам потребуется GROUP BY и задать с его помощью аналитику по ContactName и ProductName. Начальный вариант решения:

```
SELECT c.ContactName, p.ProductName
, SUM(od.Quantity * od.UnitPrice * (1 - od.Discount)) AS Amt
FROM Customers c
JOIN Orders o ON o.CustomerID = c.CustomerID
JOIN [Order Details] od ON od.OrderID = o.OrderID
JOIN Products p ON p.ProductID = od.ProductID
WHERE YEAR(o.OrderDate) = 1997
GROUP BY c.ContactName, p.ProductName
```

В этом варианте мы уже разобрались с логикой данных, но не реализовали ограничение: для каждого покупателя два товара с максимальной суммой продаж. Очевидно, что TOP и ORDER BY здесь не помогут, поскольку они действуют на весь запрос. И нужно вспомнить про итеративный способ соединения данных с помощью APPLY:

```
SELECT c.ContactName, p.ProductName
FROM Customers c
CROSS APPLY (
  SELECT TOP 2 p.ProductName
  FROM Orders o
  JOIN [Order Details] od ON od.OrderID = o.OrderID
  JOIN Products p ON p.ProductID = od.ProductID
  WHERE YEAR(o.OrderDate) = 1997
    — Соединение с внешним запросом
    AND o.CustomerID = c.CustomerID
    GROUP BY p.ProductName
    ORDER BY SUM(od.Quantity * od.UnitPrice * (1 — od.Discount)) DESC) p
```

Переписать JOIN на APPLY это уже техника, но нужно быть внимательным, чтобы правильно разбить исходный запрос на внешний и внутренний запросы и корректно их соединить в критерии внутреннего запроса.

Для данной задачи решение с APPLY является классическим, но можно решить эту задачу с помощью оконных функций, поскольку они также позволяют решать задачи с рейтингами. Давайте используем оконную функцию ROW_NUMBER, чтобы сделать рейтинг продаж товаров для каждого покупателя:

```
SELECT c.ContactName, p.ProductName
, ROW_NUMBER() OVER (
    PARTITION BY c.ContactName
    ORDER BY SUM(od.Quantity * od.UnitPrice * (1 - od.Discount)) DESC
) AS RatingByAmt
FROM Customers c
JOIN Orders o ON o.CustomerID = c.CustomerID
JOIN [Order Details] od ON od.OrderID = o.OrderID
JOIN Products p ON p.ProductID = od.ProductID
WHERE YEAR(o.OrderDate) = 1997
GROUP BY c.ContactName, p.ProductName
```

Поскольку оконные функции работают только в SELECT, для фильтрации данных с рейтингом 1 и 2 (по столбцу RatingByAmt) нужно использовать обертку как технику преодоления синтаксических ограничений в SQL:

```
SELECT ContactName, ProductName FROM (
SELECT c.ContactName, p.ProductName
, ROW_NUMBER() OVER (
    PARTITION BY c.ContactName
    ORDER BY SUM(od.Quantity * od.UnitPrice * (1 – od.Discount)) DESC
) AS RatingByAmt
FROM Customers c
JOIN Orders o ON o.CustomerID = c.CustomerID
JOIN [Order Details] od ON od.OrderID = o.OrderID
JOIN Products p ON p.ProductID = od.ProductID
WHERE YEAR(o.OrderDate) = 1997
GROUP BY c.ContactName, p.ProductName
) t
WHERE RatingByAmt < 3
```

Задача 3

Задача из моего авторского курса.

`Сколько товаров нужно заказать у поставщиков для выполнения текущих заказов.`

Важно при решении задач понимать бизнес-логику компании, в которой мы работаем и при неопределенной постановке задачи нужно задавать вопросы и уточнять требования. Если внимательно посмотреть на таблицу справочника товаров (Products), то мы увидим, что наличие товара это поле UnitsInStock. Но есть еще интересное поле ReorderLevel, в котором задается уровень запаса для обеспечения надежности поставок товара. И необходимо уточнить требование о необходимости его учитывать.

Логика задачи. Сама по себе задача несложная: нужно посчитать остатки товара (с учетом нормы запаса), вычесть количество проданных штук и выбрать товары с дефицитом. Формализация логики:

- результат: поставщик, товар, дефицит товара
  - показатель: [дефицит товара] = [к-во продаж] — [остаток на складе] — [норма запаса]
- критерий:
  - текущие (неотгруженные) товары

Анализ данных. Вначале разберемся, что такое неотгруженные товары. При анализе таблицы заказов мы видим поле ShippedDate, и, если внимательно изучить сами данные, то становится понятным, что если поле пустое, это значит, что товары по этому заказы еще не отгружены. Ну а дальше все просто: показатель по количеству проданного товара считаем по полю Quantity в таблице номенклатуры заказа (Order Details). А показатели остатков товара и нормы запаса хранятся в справочнике товаров (Products). Формализация данных:

- поставщики: таблица Suppliers
- справочник товаров: Products
    - показатель:
        - наличие товара на складе: UnitsInStock
        - норма запаса: ReorderLevel
- заказы: таблица Orders
    - критерий: заказ не отгружен (ShippedDate IS NULL)
- проданные товары: таблица Order Details
    - показатель: количество штук товара: sum(Quantity)

Вначале давайте посчитаем количество единиц товара в актуальных заказах — двигаемся от известного к неизвестному:

```
select od.ProductID, sum(od.Quantity)
from Orders o
join [Order Details] od on o.OrderID = od.OrderID
where o.ShippedDate is null
group by od.ProductID
```

Далее мы уже можем посчитать дефицит товара — важный нюанс, что показатели по наличию товара и нормы запаса уже имеются в готовом виде в таблице Products, поэтому нужно сделать соединение с таблицей Products и добавить их как аналитику в GROUP BY. В результате запроса мы можем вывести название продукта и, соответственно, поменять аналитику. Также нам нужно вывести название поставщика товара, для чего мы делаем соединение с таблицей Suppliers и добавляем название поставщика как аналитику в GROUP BY.

Поскольку в критерии у нас имеется показатель, который мы вычисляем с помощью агрегатной функции, то он работает через HAVING, а не через WHERE. Если мы хотим вывести количество единиц товара, которые нужно заказать, то с помощью функции abs убираем минус из результирующего значения (либо меняем порядок вычисления — делаем как нам понятнее):

```
select s.CompanyName, p.ProductName, abs(p.UnitsInStock – sum(od.Quantity) – p.Re
orderLevel) asToOrder
from Orders o
join [Order Details] od on o.OrderID = od.OrderID
join Products p on od.ProductID = p.ProductID
join Suppliers s on p.SupplierID = s.SupplierID
where o.ShippedDate is null
group by s.CompanyName, p.ProductName, p.UnitsInStock, p.ReorderLevel
having p.UnitsInStock – sum(od.Quantity) – p.ReorderLevel < 0
```