### Лекция 17:

1. Информационная энтропия. Использование в практических бизнес-задачах и задачах анализа данных.
2. Написание запроса на подсчет эффективного рабочего времени сотрудника: подсчет времени на работы по работнику, если интервалы начало-конец работы пересекаются. Повторение ranking function.
3. Курсоры

### Применение энтропии:

Определение энтропии:

, где p\_i – вероятность наступления события i.

Свойства энтропии:

1. Неотрицательная величина. Нулевое значение принимается только в случае, если какая-то из вероятностей равна 1, а остальные равны 0. (если считать, что 0\*ln(0)=0)
2. При заданном n энтропия принимает максимальное значение при p1=…= pn = 1/n. Принимаемое значение не превышает ln(n)

Так же удобное для анализа понятие – ширина спектра. Ширина спектра = exp(S), то есть ширина спектра показывает эффективное колво событий.

**Пример использования:**

Есть событие Xn (например, покупка товара Xn), а также есть набор событий Yk, состоящих в том, что товар Xn продавался вместе с товаром Yk. (например, встречались в одном документе).

Тогда для каждой пары XnYk можно посчитать величину – ni – количество документов (количество покупок), когда товары продавались вместе.

Как при помощи энтропии попытаться найти такие товары, которые продавались в паре значительное количество раз, используя свойства энтропии?

Рассмотрим несколько типичных случаев:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1й набор (ni)** | **pi** | **2й набор (ni)** | **pi** | **3й набор (ni)** | **pi** | **4й набор (ni)** | **pi** | **5й набор (ni)** | **pi** | **6й набор (ni)** | **pi** |
| 51 | 0.1 | 1 | 0 | 1 | 0 | 2 | 0 | 510 | 1 | 1 | 0.03 |
| 51 | 0.1 | 1 | 0 | 1 | 0 | 100 | 0.2 |  |  | 1 | 0.03 |
| 51 | 0.1 | 1 | 0 | 1 | 0 | 2 | 0 |  |  | 1 | 0.03 |
| 51 | 0.1 | 1 | 0 | 251 | 0.49 | 100 | 0.2 |  |  | 15 | 0.5 |
| 51 | 0.1 | 501 | 0.98 | 1 | 0 | 2 | 0 |  |  | 1 | 0.03 |
| 51 | 0.1 | 1 | 0 | 251 | 0.49 | 100 | 0.2 |  |  | 10 | 0.3 |
| 51 | 0.1 | 1 | 0 | 1 | 0 | 2 | 0 |  |  | 1 | 0.03 |
| 51 | 0.1 | 1 | 0 | 1 | 0 | 100 | 0.2 |  |  | 1 | 0.03 |
| 51 | 0.1 | 1 | 0 | 1 | 0 | 2 | 0 |  |  | 1 | 0.03 |
| 51 | 0.1 | 1 | 0 | 1 | 0 | 100 | 0.2 |  |  | 1 | 0.03 |
| Энтропия | 2.3 |  | 0.1 |  | 0.8 |  | 1.71 |  | 0 |  | 1.6 |
| Ширина спектра | 10 |  | 1.1 |  | 2.22 |  | 5.51 |  | 1 |  | 4.8 |
| ln(n)= | 2.3 |  |  |  |  |  |  |  |  |  |  |

Случай 1:

Товар 0 встречался с товарами 1-10 одинаковое количество раз во всех 510 документах, то есть 51 раз с каждым товаром. То есть, у товара 0 есть свойство «равномерно продастся с любым товаром», видим, что в этом случае энтропия действительно принимает максимально возможное значение, а значит, для нашего товара 0 не существует единственного явно выделенного товара, с которым он всегда продаётся вместе.

Случай 2:

Товар 0 встретился с товаром 5 501 раз, с остальными по 1 разу.

Ширина спектра близка к 1. То есть действительно для нашего товара есть товар, с которым он всегда продается в паре.

Случай 3:

Товар встретился с товарами 4 и 6 по 251 разу, с остальными по 1 разу. Ширина спектра близка к 2-м, что опять же отражает эффективное колво сопутствующих товаров для нашего товара 1.

Так же обратите внимание на случаи 4, 5, 6. Подумайте, почему в случае 6 ширина спектра не так хорошо отражает эффективное количество парных товаров для нашего заданного.

**Вопрос:** В чём преимущества использования энтропии по сравнению, например, с очевидным способом: для каждой пары товаров посчитать, сколько раз, они продавались в одном документе, и отсортировать по каждой паре товаров по убыванию этого показателя?

**Ответ:** Ширина спектра позволяет сказать, на каком товаре нужно провести отсечение для отображения эффективного колва товаров (то есть показывает TOP N), что в случае простой сортировки сделать достаточно просто в случае, если мы ищем пары для одного товара и почти невозможно, если делаем для 1000 товаров. Пример 6 показывает, что в некоторых случаях просто TOP N брать не совсем верно, но в целом, если есть значительно выделяющиеся пары, то простейший алгоритм вывода может руководствоваться значением ширины спектра.

Задание: напишите запрос, который по таблице Продажи вычисляет такую энтропию по товару, заданному параметром.

|  |
| --- |
| **Продажи** |
| **ндок** |
| **нпоз** |
| Товар\_ID |
| Колво |
| Цена |

Пример запроса для подсчета ширины спектра (написан в таком виде, что можно использовать с небольшими синтаксическими изменениями в Access, если для временных таблиц использовать запросы):

DECLARE @Продажи as TABLE (ндок int, нпоз int, Товар\_ID int, Колво int, Цена float, primary key(ндок, нпоз))

DECLARE @Документы as TABLE (ндок int, Товар\_ID int, primary key(ндок, Товар\_ID))

INSERT INTO @Документы

SELECT distinct ндок, Товар\_ID

FROM @Продажи

DECLARE @Пары TABLE (Товар\_ID\_1 int, Товар\_ID\_2 int, Раз int)

INSERT INTO @Пары

SELECT Д1.Товар\_ID, Д2.Товар\_ID, COUNT(\*) Раз

FROM @Документы Д1 INNER JOIN @Документы Д2 ON

Д1.ндок = Д2.ндок AND

Д1.Товар\_ID <> Д2.Товар\_ID

GROUP BY Д1.Товар\_ID, Д2.Товар\_ID

DECLARE @ПоТовару TABLE (Товар\_ID int, ВПаре int)

INSERT INTO @ПоТовару

SELECT Д1.Товар\_ID, COUNT(\*)

FROM @Документы Д1 INNER JOIN @Документы Д2 ON

Д1.ндок = Д2.ндок AND

Д1.Товар\_ID <> Д2.Товар\_ID

GROUP BY Д1.Товар\_ID

SELECT Товар\_ID\_1, EXP(-SUM(p\*LOG(p))) ШиринаСпектра

FROM

(

SELECT Пары.Товар\_ID\_1, Пары.Товар\_ID\_2, CAST(Пары.Раз as float)/ПоТовару.ВПаре p

FROM @Пары Пары LEFT JOIN @ПоТовару ПоТовару ON

Пары.Товар\_ID\_1 = ПоТовару.Товар\_ID

) Итог

GROUP BY Товар\_ID\_1

Но Вы уже знаете и много иных синтаксисов. Поэтому приведем примеры написания такого запроса с использованием подзапросов в SELECT или оконных функций:

SELECT Товар\_ID\_1, EXP(-SUM(p\*LOG(p))) ШиринаСпектра

FROM

(

SELECT Товар\_ID\_1, Товар\_ID\_2, CAST(Раз as float)/

(SELECT SUM(Раз) FROM @Пары Пары WHERE Пары.Товар\_ID\_1 = **П.Товар\_ID\_1**) p

FROM @Пары П

) T

GROUP BY Товар\_ID\_1

Или:

SELECT Товар\_ID\_1, EXP(-SUM(p\*LOG(p))) ШиринаСпектра

FROM

(

SELECT Товар\_ID\_1, Товар\_ID\_2, CAST(Раз as float)/

( SUM(Раз) OVER(PARTITION BY Товар\_ID\_1) ) p

FROM @Пары

) T

GROUP BY Товар\_ID\_1

**Другие примеры применения:**

На Вашем сайте для каждой товарной группы отображены параметры, пользуясь которыми покупатель может отфильтровать для себя список товаров. Обычно свойств у товаров очень много и не всегда можно предугадать, что конкретно будет наиболее важно для покупателя. Поэтому Вы можете собрать статистику, какие свойства чаще всего используются покупателями для фильтрации, и на основе этой информации изменить интерфейс отображения параметров на сайте.

**Прочие применения энтропии:** например для **Classification Tree** энтропия может применяться для выбора параметров в критерии разветвления.

Давайте приведем пример для лучшего понимания.

**Задача:** Вам нужно по свойствам объекта научиться понимать, к какому классу этот объект принадлежит.

Пусть у нас есть обучающая выборка следующего вида:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Номер** | **Класс** | **Длинный** | **Сладкий** | **Желтый** |
| 1 | банан | 1 | 1 | 1 |
| 2 | банан | 1 | 1 | 1 |
| 3 | банан | 1 | 1 | 0 |
| 4 | банан | 1 | 0 | 1 |
| 5 | банан | 1 | 1 | 1 |
| 6 | апельсин | 0 | 0 | 0 |
| 7 | апельсин | 0 | 0 | 1 |
| 8 | апельсин | 0 | 0 | 1 |
| 9 | арбуз | 1 | 1 | 0 |
| 10 | арбуз | 0 | 1 | 0 |

Нам дали неизвестный объект, для которого проставили значения свойств. Мы хотим проставить значение класса по этим признакам объекта.

Посчитаем сначала энтропию исходного множества: p\_банан = 0.5, p\_апельсина = 0.3, p\_арбуза = 0.2. Для таких значений вероятностей энтропия исходного множества равна ~ 1.49 (логарифм по основанию 2).

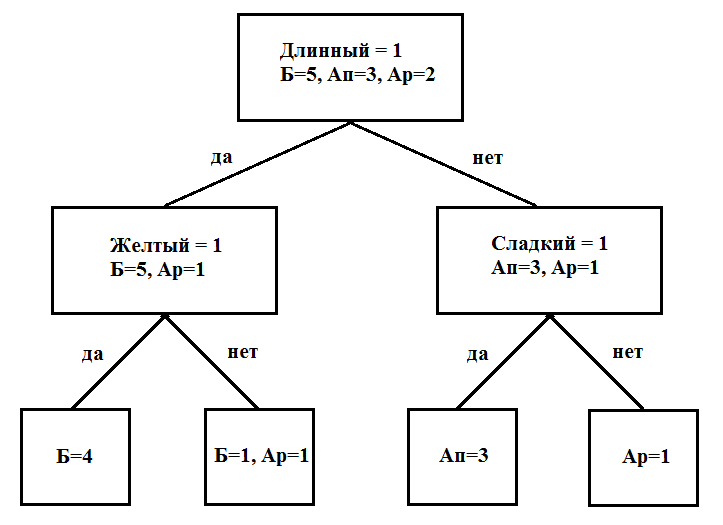
Теперь давайте разобьем объекты на классы в зависимости от значения каждого признака. Начнем с признака «длинный»:

Если разбить объекты на классы с «длинный» = 1, и с «длинный» = 0, то в первом классе p\_банан = 5/6, p\_апельсин = 0, p\_арбуз = 1/6, во втором классе p\_банан = 0, p\_апельсин = 3/4, p\_арбуз = 1/4. Энтропии в каждом классе соответственно равны: 0.65 и 0.81. В итоге изменение энтропии после деления на классы по признаку «Длинный» равно: 1.49 – 0.6\*0.65 – 0.4\*0.81 (0.6 и 0.4 – доля объектов в каждом классе) = 0.77.

Давайте аналогичным образом посчитаем изменение энтропии, если нужно поделить на группы по признаку «сладкий» и «желтый»: для признака «сладкий» изменение равно 0.61, для признака «желтый»: 0.33. То есть наибольшее изменение энтропии (внесение порядка) дает деление по признаку «длинный».

Далее внутри групп поделенных по признаку «длинный» нужно повторить операцию деления на группы (в каждой ветке по отдельности).

В результаты мы получим следующее дерево принятия решений:



Видим, что для данного примера не удалось полностью разделить все группы по признакам, но в данном случае – это проблема того, что трех признаков было мало, они не отделяли все бананы от всех арбузов.

Что дальше делать с таким деревом: во-первых, на тестовой выборке проверить, что Ваше дерево решений не избыточно. А дальше уже можно классифицировать произвольные выборки фруктов. В нашем курсе мы подробно не рассматриваем методы обучения. Об этом Вы можете почитать в любом курсе по Machine Learning. Например, мы вообще никак не рассматриваем случай, когда свойства имеют не бинарное значение (а конечный набор дискретных или непрерывные).

**Задача о балансе скобочек в реальной жизни:**

Есть список работ, выполняемых сотрудником, для каждой из них он отмечает глобальное время начала и глобальное время завершения. По сути деятельности работник может выполнять несколько работ одновременно. Необходимо вычислить время, которое было занято работами.

Например: сотрудник наклеивает стикеры на товар. Каждое задание для наклейки содержит несколько разных товаров. Товар приносят на стикерение постепенно. И может оказаться так, что часть товара из задания уже принесли, а часть еще нет. Поэтому сотрудник сам переключается между заданиями, при этом на каждом задании фиксируется, когда его выдали и когда последний товар с наклейками из этого задания забрали у работника. Идеально было бы, если бы в моменты переключения между заданиями работник это тоже фиксировал, но, к сожалению, не для всех типов деятельности это возможно сделать без уменьшения производительности труда.

Зачем нужно знать время, посвященное непосредственно работе? Вы можете предложить свои варианты. Мои варианты такие: для оценки производительности труда работника, для оценки эффективности бизнес-процесса (если у сотрудника большое время простоя и высокая производительность – это значит, что, возможно, нужно изменять процесс доставки работы сотруднику), для установления сдельной ставки работы, для оценки необходимого количества работников в пиковые дни.

Начнем с попытки решения задачи уже известными нам способами.

Пример данных (для примера без даты, только со временем, но считаем, что данные за один и тот же день):

|  |  |  |
| --- | --- | --- |
| **ндок** | **Начало** | **Конец** |
| 1 | 12:00 | 13:30 |
| 2 | 13:00 | 17:00 |
| 3 | 14:00 | 16:00 |
| 4 | 17:30 | 19:00 |

Для конкретного примера документы 1, 2, 3 пересекаются. Документ 4 отдельно от них.

Понятно, что суммарное время, которое сотрудник уделил заданиям равно 6.5 часов, что не равно сумме(конец - начало), так как пачки пересекаются.

Как поступать?

Давайте переведем таблицу в следующий вид, где флаг = 1 соответствует началу действия, а флаг -1 соответствует концу действия.

|  |  |  |
| --- | --- | --- |
| **ндок** | **Время** | **Флаг** |
| 1 | 12:00 | 1 |
| 2 | 13:00 | 1 |
| 1 | 13:30 | -1 |
| 3 | 14:00 | 1 |
| 3 | 16:00 | -1 |
| 2 | 17:00 | -1 |
| 4 | 17:30 | 1 |
| 4 | 19:00 | -1 |

Видим, что в таком формате задача уже становится похожа на задачу про «баланс скобок». Если считать, что входные данные у нас корректные, то нам всего лишь нужно сгруппировать «скобки» нужным образом. Здесь 1 – открывающаяся строка, а -1 – закрывающаяся.

Если посчитать для каждой строки сумму всех флагов до этой строки включительно (в отсортированном по времени списке), то получим следующее:

|  |  |  |  |
| --- | --- | --- | --- |
| ндок | Время | Флаг | Сумма |
| 1 | 12:00 | 1 | 1 |
| 2 | 13:00 | 1 | 2 |
| 1 | 13:30 | -1 | 1 |
| 3 | 14:00 | 1 | 2 |
| 3 | 16:00 | -1 | 1 |
| 2 | 17:00 | -1 | 0 |
| 4 | 17:30 | 1 | 1 |
| 4 | 19:00 | -1 | 0 |

То есть, там, где сумма = 1 и флаг = 1, начало группы действий, а там, где флаг = -1 и сумма = 0 – там завершение группы действий. Умея вычислять такую сумму, мы легко сможем вычислить эффективное время сотрудника.

Заметим, что задача про подсчет суммы похожа на задачу про подсчет рейтинга (где необходимо было проставить рейтинг покупателю в зависимости от оборота его покупок, то есть в зависимости от того, сколько покупателей закупились на большую чем он сумму).

SET DATEFORMAT DMY

DECLARE @date1 datetime, @date2 datetime

SET @date1 = '20210301 5:00:00'

SET @date2 = '20210401 5:00:00'

-- создаем список заданий в удобном виде. Таблица задания – исходная таблица в формате ндок, Начало, Конец

DECLARE @Список TABLE (ндок int, День datetime, Время datetime, Флаг int)

INSERT INTO @Список(ндок, День, Время, Флаг)

SELECT ндок, CAST(Начало as date) День, Начало Время, 1 Нач

FROM Задания

WHERE Начало >= @date1 AND Начало < @date2

UNION

SELECT ндок, CAST(Начало as date) День, Конец, -1

FROM Задания

WHERE Начало >= @date1 AND Начало < @date2

-- итоговый подсчет эффективного времени по дням

SELECT День, SUM(

CASE

WHEN Флаг = 1 AND Сумм = 1 THEN -1\*CAST(Время as float)

WHEN Флаг = -1 AND Сумм = 0 THEN 1\*CAST(Время as float)

END

)\*24 Длтн

FROM

(

SELECT Список.ндок, Список.День, Список.Время, Список.Флаг, SUM(Список\_1.Флаг) Сумм

FROM @Список Список INNER JOIN @Список Список\_1 ON

Список.Время >= Список\_1.Время

GROUP BY Список.ндок, Список.День, Список.Время, Список.Флаг

) T

GROUP BY День

Как Вы считаете, в чём проблема у такого способа решении задачи? В каком случае запрос будет выдавать неверные данные?

Подумайте, что получится, если, например, 3 и 2 задание вместе закончились только в 17:00? Реальное эффективное время от этого не изменится, но расчетное «испортится», потому что результат подзапроса с подсчетом суммы будет следующим (подумайте почему?):

|  |  |  |  |
| --- | --- | --- | --- |
| ндок | Время | Флаг | Сумма |
| 1 | 12:00 | 1 | 1 |
| 2 | 13:00 | 1 | 2 |
| 1 | 13:30 | -1 | 1 |
| 3 | 14:00 | 1 | 2 |
| 3 | 17:00 | -1 | 0 |
| 2 | 17:00 | -1 | 0 |
| 4 | 17:30 | 1 | 1 |
| 4 | 19:00 | -1 | 0 |

Почему? Потому что при суммировании Вы суммируете флаги по всем строкам, в которых время меньше или равно текущему. То есть нужно дополнительно заставить запрос различать 17:00 для 3-ого задания и для 2-ого, что реализуется небольшой доработкой запроса.

Для этого, конечно же, можно использовать и ranking function. Тогда реализация будет выглядеть следующим образом:

-- привлечение ranking function

DECLARE @СписокНомера TABLE (ндок int, День datetime, Время datetime, Флаг int, НомерСтроки int)

INSERT INTO @СписокНомера(ндок, День, Время, Флаг, НомерСтроки)

SELECT ндок, День, Время, Флаг, ROW\_NUMBER() OVER(ORDER BY Время, ндок) НомерСтроки

FROM @Список T

SELECT День, SUM(

CASE

WHEN Флаг = 1 AND Сумм = 1 THEN -1\*CAST(Время as float)

WHEN Флаг = -1 AND Сумм = 0 THEN 1\*CAST(Время as float)

END

)\*24 Длтн

FROM (

SELECT Список.ндок, Список.День, Список.Время, Список.Флаг, SUM(Список\_1.Флаг) Сумм

FROM @СписокНомера Список INNER JOIN @СписокНомера Список\_1 ON

Список.НомерСтроки > = Список\_1.НомерСтроки

GROUP BY Список.ндок, Список.День, Список.Время, Список.Флаг

) T

GROUP BY День

Смысл: мы нумеруем строки по возрастанию времени и номера задания, таким образом, мы сможем отличить одну строку с 17:00 от другой с 17:00.

Обратите внимание, что мы на самом деле считаем для каждого момента времени накопительную сумму по флагам. С 2012 версии MS SQL накопительную сумму можете сразу посчитать при помощи оконных функций.

Теперь рассмотрим еще один полезный способ решения такой задачи. Логически способ решения во всех случаях один и тот же. Различаются только конкретные реализации.

**Курсоры:**

**Что такое курсор?**

По сути, курсор – это нереляционный результат с гарантированным порядком строк. Вспомните, что в БД в определении отношения важно то, что строки не упорядочены, то есть Вы не знаете, какая первая – вторая – третья и тд. При этом Вы уже на первых занятиях писали запросы, в которых строки выводились в определенном порядке (ORDER BY).

При работе с базами данных иногда возникают случаи, когда необходимо работать не со всем массивом данных, а с отдельными строками, причем в определенном порядке. Например, какие-то данные хранятся в БД, но определенное пользовательское приложение может одновременно работать только с одной строкой (или несколькими, но не всей таблицей, выдаваемой запросом) (например, openrecordset в VB). Для таких целей в БД есть специальный объект – курсор.

Курсоры бывают:

1. последовательные и прокручиваемые.

Последовательный курсор - данные из курсора могут быть прочитаны только в определенном порядке, прокручиваемый курсор – данные читаются в определяемом пользователем порядке с возможностью перехода к определенной строке данных.

1. статические, динамические и управляемые набором ключей.

В статический курсор информация читается из БД один раз и хранится в виде моментального снимка, поэтому изменения, внесенные в базу другим пользователем, не видны. Статический курсор не изменяется после создания и всегда отображает тот набор данных, который существовал на момент открытия курсора. Изменения других пользователей данных в исходной таблице никак не влияют на статический курсор. Такой курсор всегда открывается в режиме «только для чтения».

Динамический курсор поддерживает данные в «живом» состоянии, но это требует затрат сетевых и программных ресурсов. При использовании динамических курсоров не создается копия исходных данных, а выполняется динамическая выборка из исходных таблиц только при обращении пользователя к тем или иным данным. На время выборки сервер блокирует строки, а все изменения, вносимые пользователем в полный результирующий набор курсора, будут видны в курсоре.

Курсор, управляемый набором ключей, находится посередине между двумя крайностями. Изменения отслеживаются. Пользователь видит изменения по строкам, ключевые поля которых изначально были подгружены в курсор, новые строки не видны.

Статические курсоры хороши для статистических и аналитических целей и лучше справляются с выборкой большого количества данных. В системах же электронных покупок или резервирования билетов необходимо динамическое восприятие обновляемой информации по мере внесения изменений. Здесь уже требуется динамический курсор, в этом случае обычно объем передаваемых данных невелик и доступ к ним осуществляется на уровне отдельных рядов.

Основные действия с курсорами:

* Создание или объявление курсора; (DECLARE)
* Открытие курсора (наполнение курсора данными); (OPEN)
* Выборка из курсора и изменение с его помощью строк данных; (FETCH)
* Закрытие курсора (после чего он становится недоступным для пользовательских программ); (CLOSE)
* «Освобождение» курсора (удаление курсора как объекта). (DEALLOCATE).

*Самую подробную информацию про курсоры с примерами применения читайте в хелпе.*

**Открытие курсора:**

Для открытия курсора и наполнения его данными из указанного при создании курсора запроса SELECT используется следующая команда:

OPEN {{[GLOBAL]имя\_курсора }

|@имя\_переменной\_курсора}

После открытия курсора происходит выполнение связанного с ним оператора SELECT, выходные данные которого сохраняются в многоуровневой памяти.

**Выборка данных:**

Сразу после открытия курсора можно выбрать его содержимое (результат выполнения соответствующего запроса) посредством следующей команды:

FETCH [[NEXT | PRIOR | FIRST | LAST

| ABSOLUTE {номер\_строки

| @переменная\_номера\_строки}

| RELATIVE {номер\_строки |

@переменная\_номера\_строки}]

FROM ]{{[GLOBAL ]имя\_курсора }|

@имя\_переменной\_курсора }

[INTO @имя\_переменной [,...n]]

В конструкции INTO @имя\_переменной [,...n] задается список переменных, в которых будут сохранены соответствующие значения столбцов возвращаемой строки. Порядок указания переменных должен соответствовать порядку столбцов в курсоре, а тип данных переменной – типу данных в столбце курсора. Если конструкция INTO не указана, то поведение команды FETCH будет напоминать поведение команды SELECT – данные выводятся на экран.

(Пример из MSDN)

DECLARE authors\_cursor SCROLL CURSOR FOR

SELECT au\_lname, au\_fname FROM authors

ORDER BY au\_lname, au\_fname

OPEN authors\_cursor

-- Fetch the last row in the cursor.

FETCH LAST FROM authors\_cursor

-- Fetch the row immediately prior to the current row in the cursor.

FETCH PRIOR FROM authors\_cursor

-- Fetch the second row in the cursor.

FETCH ABSOLUTE 2 FROM authors\_cursor

-- Fetch the row that is three rows after the current row.

FETCH RELATIVE 3 FROM authors\_cursor

-- Fetch the row that is two rows prior to the current row.

FETCH RELATIVE -2 FROM authors\_cursor

CLOSE authors\_cursor

DEALLOCATE authors\_cursor

GO

**Изменение и удаление данных**

Для выполнения изменений с помощью курсора необходимо выполнить команду UPDATE в следующем формате:

UPDATE имя\_таблицы

SET {имя\_столбца={DEFAULT | NULL | выражение}}[,...n]

WHERE CURRENT OF {{[GLOBAL] имя\_курсора} |@имя\_переменной курсора}

За одну операцию могут быть изменены несколько столбцов текущей строки курсора, но все они должны принадлежать одной таблице.

Для удаления данных посредством курсора используется команда DELETE в следующем формате:

DELETE имя\_таблицы

WHERE CURRENT OF {{[GLOBAL] имя\_курсора} |@имя\_переменной курсора}

В результате будет удалена строка, установленная текущей в курсоре.

**Закрытие курсора**

CLOSE {имя\_курсора | @имя\_переменной\_курсора}

После закрытия курсор становится недоступным для пользователей программы. При закрытии снимаются все блокировки, установленные в процессе его работы. Закрытие может применяться только к открытым курсорам. Закрытый, но не освобожденный курсор может быть повторно открыт. Не допускается закрывать неоткрытый курсор.

**Освобождение курсора**

Закрытие курсора необязательно освобождает ассоциированную с ним память. В некоторых реализациях нужно явным образом освободить ее с помощью оператора DEALLOCATE. После освобождения курсора освобождается и память, при этом становится возможным повторное использование имени курсора.

DEALLOCATE { имя\_курсора | @имя\_переменной\_курсора }

Для контроля достижения конца курсора рекомендуется применять функцию: @@FETCH\_STATUS

Функция @@FETCH\_STATUS возвращает:

0, если выборка завершилась успешно;

-1, если выборка завершилась неудачно вследствие попытки выборки строки, находящейся за пределами курсора;

-2, если выборка завершилась неудачно вследствие попытки обращения к удаленной или измененной строке.

**Применение курсора к нашей задаче:**

CREATE TABLE #СписокКурсор (ндок int, День datetime, Время datetime, Флаг int, Сумм int, primary key(ндок, День, Время, Флаг))

INSERT INTO #СписокКурсор (ндок, День, Время, Флаг, Сумм)

SELECT ндок, День, Время, Флаг, 0

FROM @Список

--Курсором

DECLARE @day datetime, @date datetime, @dok int, @flag int, @sum int

SET @sum = 0

DECLARE data\_cursor CURSOR FOR

SELECT ндок, День, Время, Флаг

FROM #СписокКурсор

ORDER BY День, Время, ндок, Флаг

OPEN data\_cursor

-- «помещаем» первую строку из таблицы в переменные

FETCH NEXT FROM data\_cursor

INTO @dok, @day, @date, @flag

WHILE @@FETCH\_STATUS = 0

BEGIN

-- переменная @sum будет хранить в себе сумму флагов по всем строкам, которые выше текущей по дате/времени, номеру документа и флагу

SET @sum = @sum + @flag

-- проставим значение суммы в таблицу

-- такая версия сработает всегда

--UPDATE #СписокКурсор SET Сумм = @sum WHERE ндок = @dok AND День = @day AND Время = @date AND Флаг = @flag

-- а эта только в случае, если Вы заведете в таблице #СписокКурсор ключи

UPDATE Список SET Сумм = @sum FROM #СписокКурсор Список WHERE CURRENT OF data\_cursor

-- помещаем следующую строку в переменные

FETCH NEXT FROM data\_cursor

INTO @dok, @day, @date, @flag

-- продолжаем действия в цикле, пока строки курсора не закончатся

END

CLOSE data\_cursor

DEALLOCATE data\_cursor

SELECT День, SUM(

CASE

WHEN Флаг = 1 AND Сумм = 1 THEN -1\*CAST(Время as float)

WHEN Флаг = -1 AND Сумм = 0 THEN 1\*CAST(Время as float)

END

)\*24 Длтн

FROM #СписокКурсор T

GROUP BY День

DROP TABLE #СписокКурсор

**Комментарий: не используйте курсор, если можно написать запрос без него, так как скорость выполнения операций обработки данных посредством курсора заметно ниже, чем у стандартных средств SQL. Не повторяйте анекдот про индусского программиста, который SUM значений по столбцу считал курсором.**

Поэтому старайтесь избегать курсоров, если это возможно в Вашей конкретной задаче. (Сравните, что такое ловить рыбу на удочку и ловить сетью? Как Вы поймаете больше за одно и то же время?) И еще хороший пример из той же книги T-SQL Fundamentals I Ben-Gan. Представьте, что у Вас есть две апельсиновые фабрики. Ваша задача разделить апельсины по размеру на 3 вида: маленькие, средние и крупные. Ваша старая фабрика устроена так, что в конце конвейера стоят рабочие и вручную раскладывают апельсины по коробкам, параллельно сортируя (моделирование курсора). Если у Вас новая фабрика, где есть механизм с отверстиями соответствующего размера, то всё разделение произойдет автоматически. Рабочим останется только запаковать коробки в конце конвейера (если это тоже не делает механизм). Какая фабрика будет работать быстрее?

В целом, при использовании курсоров стоит помнить:

- курсоры противоречат идее теории множеств, лежащей в основе БД (именно из-за упорядоченности);

- курсоры – это затратная операция, так как операции производятся с каждой записью по отдельности, в то время как SQL – это язык, где оперируют множествами. Поэтому обычный запрос и запрос с курсором с одинаковым результатом будут работать существенно различное время (курсор дольше);

- в целом реализации с курсором занимают больше строк и вся реализация становится менее читаемой (так как нужно отдельно объявить курсор, выделить память, поработать, закрыть).

**Резюмируя:**

Когда нужно применять курсоры: например, если Вам нужно выполнить определенную манипуляцию с каждой строчкой таблицы (например, в каждой строке таблицы указана отдельная задача/команда, которую нужно выполнить, возможно, выполнение требуется строго в определенном порядке).

Или если Ваше обычное решение работает долго, и Вы перепробовали всевозможные действия для ускорения, но ничего не помогает. Вы знаете, что данные обработать Вам нужно только однократно (может быть раз в месяц или неделю) и можно это сделать в фоновом режиме. В этом случае Вы можете запустить построчный курсор.

Пример задачи, которую в старых версиях MS SQL Server решали при помощи курсора: подсчет накопительной суммы (см. наш пример про подсчет накопительной суммы по флагам), пока не появились более совершенные оконные функции в более поздних версиях.