Содержание

Об авторе 11

Введение 12

Глава 1 Работа с окнами в SQL 17

Эволюция оконных функций 18

Основы оконных функций 19

Описание оконных функций 19

Программирование на основе наборов данных и курсоров/итераций ... 23

Недостатки альтернативных вариантов оконных функций 29

Ваши первые решения с применением оконных функций 34

Элементы оконных функций 40

Секционирование окна 40

Упорядочивание окна 41

Определение границ окна 44

Элементы запросов с поддержкой оконных функций 45

Логическая обработка запроса 45

Инструкции с поддержкой оконных функций 48

В обход ограничений 52

Возможности для использования дополнительных фильтров 54

Повторное использование определений окна 54

Заключение 56

Глава 2 Детальное изучение оконных функций 57

Агрегатные функции 57

Описание агрегатных оконных функций 57

Поддерживаемые элементы 58

Другие идеи по работе с окнами 81

Агрегаты и DISTINCT 86

Вложение группирующих функций в оконные 89

Ранжирующие функции 93

Поддерживаемые элементы 93
ROW_NUMBER....94
NTILE 100
RANK и DENSE_RANK....105
Статистические функции 106
Поддерживаемые элементы 107
Функции распределения рангов 107
Функции обратного распределения 110
Функции смещения 113
Поддерживаемые элементы 113
LAG и LEAD 114
FIRST_VALUE, LAST_VALUE и NTH_VALUE 117
RESPECT NULLS | IGNORE NULLS 121
Заключение 124

Глава 3 Функции упорядоченного набора 125
Функции гипотетического набора....126
RANK....126
DENSE_RANK 128
PERCENT_RANK 129
CUME_DIST....131
Обобщенное решение 131
Функции обратного распределения....134
Функции смещения 138
Конкатенация строк 143
Заключение 145

Глава 4 Распознавание шаблонов в строках....146
Предпосылки распознавания шаблонов в строках 146
R010: «Распознавание шаблонов в строках: инструкция FROM» 148
Описание задачи....149
ONE ROW PER MATCH....154
ALL ROWS PER MATCH....159
RUNNING и FINAL....169
Вложение функций FIRST | LAST в PREV | NEXT 171
R020: «Распознавание шаблонов в строках: инструкция WINDOW»....173
Решения на основе распознавания шаблонов в строках 176
Возвращение верхних N строк по группам 177
Объединение интервалов....178
Поиск пропусков и островов 182
Вычисление нестандартных накопительных итогов 187

Заключение ... 192

Глава 5 Оптимизация оконных функций ... 193
Исходные данные для примеров ... 194
Рекомендации по индексированию ... 196
POC-индекс ... 196
Merge Join (Concatenation) ... 199
Обратное сканирование ... 201
Эффективное имитирование опции NULLS LAST ... 205
Улучшение параллелизма с использованием инструкции APPLY ... 209
Пакетный режим обработки ... 212
Пакетный режим с индексами columnstore ... 214
Пакетный режим с индексами rowstore ... 220
Ранжирующие функции ... 223
ROW_NUMBER ... 224
NTILE ... 226
RANK и DENSE_RANK ... 227
Пакетная обработка ... 228
Агрегатные функции и функции смещения ... 230
Без упорядочивания и указания границ окна ... 231
С упорядочиванием и указанием границ окна ... 237
Функции распределения ... 249
Функции распределения рангов ... 250
Функции обратного распределения ... 252
Пакетный режим обработки ... 255
Заключение ... 256

Глава 6 Практическое применение оконных функций в T-SQL ... 258
Вспомогательные виртуальные таблицы с числами ... 258
Последовательности значений даты и времени ... 262
Последовательности ключей ... 264
Обновление столбца с заполнением уникальными значениями ... 264
Получение диапазона значений последовательности ... 266
Разбиение на страницы ... 269
Удаление дубликатов ... 272
Сведение ... 274
Первые ***N*** элементов в группе ... 278
Имитация IGNORE NULLS ... 281
Моды ... 287
Усеченное среднее ... 291

Нарастающие итоги ... 293
Решение на основе наборов данных с оконными функциями ... 295
Решение на основе наборов данных с подзапросами и объединениями ... 296
Решение на основе курсора ... 298
Решение на основе общеязыковой среды выполнения (CLR) ... 300
Вложенные итерации ... 302
Многострочный UPDATE с переменными ... 303
Сравнительный анализ ... 306
Максимальное количество пересекающихся интервалов ... 307
Традиционный подход на основе набора данных ... 309
Решения, основанные на оконных функциях ... 312
Объединение интервалов ... 317
Традиционный подход на основе набора данных ... 320
Решения, основанные на оконных функциях ... 321
Пропуски и острова ... 326
Пропуски ... 328
Острова ... 329
Медианы ... 334
Условные агрегаты ... 337
Сортировка иерархий ... 339
Заключение ... 344

Предметный указатель ... 345

Об авторе

Ицик Бен-Ган (Itzik Ben-Gan) – сооснователь и преподаватель образовательной компании SolidQ, с 1999 года является обладателем статуса Microsoft MVP в области платформ обработки данных. Ицик провел бесчисленное количество обучающих мероприятий по всему миру, делясь опытом написания и оптимизации запросов на языке T-SQL. Автор нескольких книг, включая *T-SQL Fundamentals* и *T-SQL Querying*. Также Ицик пишет статьи для платформ *sqlperformance.com*, *ITProToday* и *SolidQ* и участвует в различных конференциях, в числе которых *PASS Summit* и *SQLBits*. Является ведущим специалистом в области T-SQL и автором курсов *Advanced T-SQL Querying, Programming and Tuning* и *T-SQL Fundamentals* в компании SolidQ.

Введение

Для меня лично оконные функции представляются наиболее важным элементом, поддерживаемым как стандартом SQL, так и его диалектом для Microsoft SQL Server, называемым T-SQL. Они позволяют выполнять вычисления применительно к целым наборам строк в очень гибкой, понятной и эффективной манере. Искусность принципов реализации оконных функций трудно переоценить, и у традиционных альтернатив со всеми присущими им недостатками нет против них ни шансов, ни аргументов. Спектр задач, которые легко решаются при помощи оконных функций, столь широк, что стоит задуматься о том, чтобы изучить эту концепцию. С момента своего появления оконные функции получили большое развитие как в SQL Server, так и в стандарте SQL. В этой книге мы будем говорить как о специфических для SQL Server свойствах применения оконных функций, так и об особенностях их использования в стандарте SQL, включая те, которые еще не реализованы в SQL Server.

Для кого эта книга

Эта книга предназначена для разработчиков SQL Server, администраторов баз данных (DBA), специалистов в области обработки данных и бизнес-аналитики, а также для тех, кто пишет запросы и разрабатывает код на языке T-SQL. В книге мы будем предполагать, что у вас за плечами есть как минимум полугодовой опыт написания и отладки запросов на языке T-SQL.

Структура книги

В книге освещаются как логические аспекты оконных функций, так и вопросы их оптимизации и практического применения.

Глава 1. Работа с окнами в SQL. В данной главе мы опишем концепцию оконных функций в языке SQL. Здесь вы познакомитесь с назначением оконных функций, их типами и элементами, участвующими в их создании и применении, включая секционирование, упорядочивание и определение границ окна.

Глава 2. Детальное изучение оконных функций. В этой главе мы погрузимся в детали и специфику оконных функций, подробно разберем агрегат-

ные и ранжирующие оконные функции, функции смещения и статистические функции (функции распределения).

Глава 3. Функции упорядоченного набора. В третьей главе мы поговорим о поддержке в языке T-SQL и стандарте SQL функций для работы с упорядоченными наборами, включая конкатенацию строк, а также рассмотрим функции для работы с гипотетическими наборами, функции обратного распределения и другие. Кроме того, для стандартных функций, пока не реализованных в языке T-SQL, мы приведем работающие решения.

Глава 4. Распознавание шаблонов в строках. Здесь мы коснемся продвинутой концепции анализа данных, именуемой *распознаванием шаблонов в строках* (row-pattern recognition – RPR), которую можно рассматривать как следующий шаг развития оконных функций. В языке T-SQL данный функционал пока не реализован, но, как уже было упомянуто ранее, в этой книге мы представим вам все важнейшие аналитические концепции, даже не воплощенные на данный момент в T-SQL.

Глава 5. Оптимизация оконных функций. В этой главе мы рассмотрим способы оптимизации оконных функций применительно к SQL Server и SQL Azure Database. Мы поговорим о применении индексации с целью ускорения выполнения запросов, затронем тему параллельных вычислений, сравним построчную и пакетную обработку запросов и узнаем другие способы оптимизации оконных функций.

Глава 6. Практическое применение оконных функций в T-SQL. В заключительной главе книги мы обратимся к практическому применению оконных функций для решения распространенных бизнес-задач.

Системные требования

Оконные функции являются составной частью ядра баз данных Microsoft SQL Server и SQL Azure Database, в связи с чем их поддержка реализована во всех версиях продуктов. Для запуска примеров из этой книги вам необходимо иметь доступ к экземпляру установки SQL Server 2019 и старше (любой версии) или SQL Azure Database с установленной базой данных с *уровнем совместимости* (compatibility level) 150 и выше. Также вам придется установить базу данных с названием TSQLV5, к которой мы будем обращаться в данной книге. Если у вас нет доступа к установленному экземпляру СУБД, на сайте Microsoft присутствуют бесплатные версии. Подробнее можно узнать по адресу *https://www.microsoft.com/en-us/sql-server/sql-server-downloads*.

В качестве клиентских инструментов для подключения к базам данных и выполнения представленных в книге примеров вы можете использовать SQL Server Management Studio (SSMS) или Azure Data Studio. Я использовал SSMS для создания графических представлений планов выполнения запросов, показанных в книге. Вы можете загрузить этот инструмент по ссылке *https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms*.

Azure Data Studio можно скачать по адресу *https://docs.microsoft.com/en-us/sql/azure-data-studio/download*.

Примеры из книги

Все примеры из этой книги, а также сопутствующие данные, ресурсы и многое другое вы можете загрузить по адресу *https://www.microsoftpressstore.com/store/t-sql-window-functions-9780135861448#downloads*.

На странице книги присутствует ссылка на архив, включающий в себя все примеры кода из книги, а также файл *TSQLV5.sql*, содержащий инструкции для создания и наполнения базы данных TSQLV5.

Благодарности

Многие люди прямо или косвенно повлияли на выход в свет этой книги, и они, конечно, достойны упоминания и благодарностей.

Спасибо Лилах за придание смысла всему, что я делаю, и за помощь с вычиткой текста.

Благодарю всех разработчиков Microsoft SQL Server (бывших и нынешних), а в особенности: Конора Каннингема (Conor Cunningham), Джо Сака (Joe Sack), Василиса Пападимоса (Vassilis Papadimos), Марка Фридмана (Marc Friedman), Крейга Фридмана (Craig Freedman), Милана Ружича (Milan Ruzic), Милана Стоича (Milan Stojic), Йована Поповича (Jovan Popovic), Борко Новаковича (Borko Novakovic), Тобиаса Тернстрёма (Tobias Ternström), Лубора Коллара (Lubor Kollar), Умачандара Джаячандрана (Umachandar Jayachandran), Педро Лопеса (Pedro Lopes), Архениса Фернандеса (Argenis Fernandez) и многих других. Я очень признателен вам за реализацию и поддержку оконных функций в SQL Server, а также за общение со мной и ответы на мои вопросы и просьбы что-то разъяснить.

Кроме того, я благодарен всей редакторской команде издательства Pearson. Лоретта Йейтс (Loretta Yates), спасибо за то, что поверила в этот проект и позволила ему реализоваться! Моя признательность Чарви Ароре (Charvi Arora) за приложенные огромные усилия. Также я благодарен Рику Кугену (Rick Kughen) и Трейси Крум (Tracey Croom). Спасибо Ашвини Кумар (Aswini Kumar) и ее команде за работу над PDF-версией книги.

Помимо прочих, хочу выразить отдельную благодарность Адаму Маханику (Adam Machanic) за согласие стать техническим редактором книги. Существует не так много людей, понимающих архитектуру SQL Server лучше тебя. Для меня вопрос выбора человека на эту роль вообще не стоял.

Спасибо Q2, Q3 и Q4. Для меня большое счастье иметь возможность обсуждать идеи, связанные с книгой, с такими профессионалами в области SQL, как вы. Кажется, я могу делиться с вами всем без опасений за возможные последствия.

Также я хочу выразить благодарность моей компании SolidQ за последние два десятилетия. Быть частью такого коллектива и наблюдать за ростом компании – настоящее счастье. Для меня сотрудники этой компании – это больше, чем просто коллеги. Это партнеры, друзья и семья. Спасибо Фернандо Герреро (Fernando G. Guerrero), Антонио Сото (Antonio Soto) и многим другим.

Отдельную признательность я выражаю Аарону Бертрану (Aaron Bertrand) и Грегу Гонзалесу (Greg Gonzales). Мне очень приятно вести колонку на сайте *https://sqlperformance.com*. SentryOne – прекрасная компания, создающая для сообщества отличные продукты и сервисы.

Также я благодарен MVP в области SQL Server Александро Месе (Alejandro Mesa), Эрланду Соммарскогу (Erland Sommarskog), Аарону Бертрану (Aaron Bertrand), Полу Уайту (Paul White) и многим другим. Я очень рад быть частью этой великолепной программы. Уровень знаний этих людей просто поражает, и мне всегда приятно встречаться с ними, разговаривать на профессиональные темы или просто попить пива. Я уверен, что в Microsoft решили уделить отдельное внимание развитию оконных функций в SQL Server именно под влиянием сообщества, и не последнюю роль в этом сыграли наши MVP. Очень приятно наблюдать за тем, как совместные усилия выливаются в нечто большее.

Наконец, я хотел бы сказать спасибо своим студентам. Я обожаю преподавать SQL, это моя страсть, и я благодарен вам за возможность ее удовлетворять. А ваши бесконечные вопросы позволяют мне двигаться дальше, обретая новые знания.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу *dmkpress@gmail.com*. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Pearson Education очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

Работа с окнами в SQL

Оконные функции (window function) способны помочь в решении огромного спектра задач посредством более простого, интуитивного и эффективного подхода к расчетам в рамках наборов данных. Оконные функции представляют собой вычисления, применяемые к набору строк, определенному при помощи инструкции *OVER*. Главным образом такие функции используются для проведения аналитики данных, включая вычисление *накопительных или нарастающих итогов* (running totals), *скользящих средних* (moving average), а также определение *пропусков* (gap), *островов* (island), *интервалов* (interval) и других показателей в ваших данных. Оконные функции базируются на продвинутой концепции, выраженной в стандарте SQL ISO/IEC и называемой *работой с окнами* (windowing). Идея, лежащая в основе этой концепции, позволяет выполнять вычисления применительно к наборам строк (окнам) и возвращать одиночные значения.

С момента появления оконных функций в SQL Server 2005 эта концепция получила существенное развитие в следующих версиях SQL Server и SQL Azure Database. Вскоре мы посмотрим внимательнее на достигнутый за это время прогресс. В актуальных на данный момент версиях продуктов все еще не реализована часть стандартного функционала этой технологии, но за последние годы были сделаны большие шаги в этом направлении. В данной книге мы будем рассматривать как функционал, реализованный в SQL Server, так и возможности, до сих пор не получившие развития в этой среде. Впервые упоминая о тех или иных средствах, я буду отдельно говорить об их поддержке в SQL Server.

С момента появления оконных функций в SQL Server я стал все чаще применять их при решении самых разнообразных задач. Я до сих пор некоторые свои давние решения, опирающиеся на традиционные конструкции языка SQL, переписываю под поддержку оконных функций. И результат в плане эффективности и простоты чаще всего оправдывает приложенные усилия. Сейчас дошло до того, что практически в каждом моем решении так или иначе присутствуют оконные функции. Кроме того, в наши дни стандарт SQL и системы управления базами данных (СУБД), а с ними и вся индустрия данных в целом двигаются в сторону аналитических решений, и оконные функции являются одним из важнейших строительных элементов на этом пути. Упор на

средства оперативной обработки транзакций (online transaction processing – OLTP), который делался в 90-е, там и остался. Лично мне видится развитие технологии SQL в ближайшем будущем именно в сторону более эффективного использования оконных функций, а значит, время, потраченное на их изучение, не пройдет для вас даром.

В книге, которую вы держите в руках, я постарался наиболее полно и подробно рассказать обо всех аспектах применения оконных функций, их оптимизации и использовании в своих решениях. В этой главе мы начнем знакомство с этими функциями и обсудим следующие моменты:

- истоки оконных функций;
- беглый взгляд на решения, использующие оконные функции;
- описание основных элементов, входящих в состав оконных функций;
- обзор инструкций, поддерживающих использование оконных функций;
- разбор стандартного решения с повторным использованием определения окна.

Эволюция оконных функций

Как я уже говорил, с момента своего появления в SQL Server оконные функции проделали уже огромный путь, а в будущем их ждет еще большее развитие. На рис. 1.1 показаны все основные вехи эволюции оконных функций в SQL Server, включая возможности, которые, как мы все надеемся, будут реализованы в ближайшем будущем.

Рис. 1.1 Процесс эволюции оконных функций в SQL Server

На данном этапе многие из перечисленных возможностей могут вам ровным счетом ни о чем не говорить. По большей части этот графический экс-

курс в прошлое был сделан в качестве исторической справки. Гарантирую, что после прочтения книги все перечисленные здесь термины будут вам хорошо знакомы. А сейчас давайте кратко пройдемся по перечисленным на рис. 1.1 этапам:

- в версии SQL Server 2005 появилась поддержка агрегатных оконных функций, но без возможности задавать границы окна, а также были реализованы ранжирующие функции (*ROW_NUMBER*, *RANK*, *DENSE_RANK* и *NTILE*);
- в версиях SQL Server 2008 и 2008 R2 оконные функции затронуты не были;
- в SQL Server 2012 появилась поддержка агрегатных оконных функций с возможностью задавать границы окна, функций смещения (*LAG*, *LEAD*, *FIRST_VALUE* и *LAST_VALUE*), а также статистических или аналитических функций (*PERCENT_RANK*, *CUME_DIST*, *PERCENTILE_CONT* и *PERCENTILE_DISC*);
- в версии SQL Server 2014 оконные функции остались без изменений;
- в SQL Server 2016 оконные функции были оптимизированы за счет появления нового оператора пакетной обработки *Window Aggregate*. Однако, чтобы воспользоваться всеми его преимуществами, хотя бы для одной из участвующих в запросе таблиц должен был быть создан индекс *columnstore*;
- в версии в SQL Server 2017 появилась поддержка первой функции для работы с упорядоченными наборами – *STRING_AGG*, позволяющей конкатенировать строки;
- в SQL Server 2019 был реализован пакетный режим для данных *rowstore*, что позволило оставить в прошлом требование наличия индекса *columnstore* для участвующих в запросе таблиц;
- несколько важных возможностей, связанных с оконными функциями, на данный момент еще не реализованы в SQL Server. Сюда можно отнести создание границ окна с помощью инструкции RANGE с указанием параметра INTERVAL, технологию распознавания шаблонов в строках, вложенные оконные функции, опции *RESPECT | IGNORE NULLS*, инструкцию *WINDOW*, служащую для повторного использования определения окна, прочие функции для работы с упорядоченными наборами и многое другое.

Основы оконных функций

Перед изучением оконных функций вам будет полезно узнать, что послужило поводом для их появления. Именно об этом мы будем говорить в этом разделе. Мы опишем разницу между подходами, основанными на работе с наборами данных и курсорами/итерациями, которые применяются в запросах, и покажем, как оконные функции позволяют перебросить мостик между ними. Наконец, мы обсудим недостатки альтернативных методов и расскажем, почему применение оконных функций в подавляющем большинстве случаев будет лучшим выбором.

Описание оконных функций

Оконная функция (window function) – это функция, применяемая к набору строк. Термином *окно* (window) в стандарте SQL описывается контекст, в котором выполняется функция. Спецификация создаваемого окна задается в SQL при помощи инструкции *OVER*. Рассмотрим в качестве примера следующий простейший запрос:

```
USE TSQLV5;
SELECT orderid, orderdate, val,
RANK() OVER(ORDER BY val DESC) AS rnk
FROM Sales.OrderValues
ORDER BY rnk;
```

О том, где скачать базу данных TSQLV5 и сопроводительные файлы, читайте в вводной главе.

Ниже представлен сокращенный вывод этого запроса:

```
orderid  orderdate  val       rnk
-------- ---------- --------- ----
10865    2019-02-02 16387.50  1
10981    2019-03-27 15810.00  2
11030    2019-04-17 12615.05  3
10889    2019-02-16 11380.00  4
10417    2018-01-16 11188.40  5
10817    2019-01-06 10952.85  6
10897    2019-02-19 10835.24  7
10479    2018-03-19 10495.60  8
10540    2018-05-19 10191.70  9
10691    2018-10-03 10164.80  10
...
```

Оконной функцией в приведенном выше примере является функция *RANK*. Для выполнения ранжирования нам необходимо выполнить упорядочивание. В данном случае мы применили сортировку по полю *val* в *убывающем* (descending) порядке. Эта функция вычисляет порядковый номер для строки в рамках заданного набора данных с учетом указанного порядка сортировки. При использовании сортировки по убыванию, как в нашем примере, ранг вычисляется путем добавления единицы к количеству строк в нашем наборе данных, в которых сортируемое значение больше, чем в текущей строке. Возьмите, к примеру, строку с рангом 5 из нашего запроса. Значение 5 получилось путем прибавления единицы к количеству строк, в которых в столбце *val* находится сумма, превышающая сумму в нашей текущей строке (11 188,40).

Характеристики набора данных, к которому относится наша строка, включая порядок упорядочивания и другие параметры, если они присутствуют, описываются при помощи ключевого слова *OVER*. Если в этой инструкции не указать границы окна, как в нашем примере, набором данных окна будет считаться весь результирующий набор запроса.

Если быть точными, окно определяется как набор строк – или *отношение* (relation), – переданное на вход фазы *логической обработки запроса* (logical query processing), в которой представлена оконная функция. Однако такое определение на данном этапе мало что вам скажет. Так что с целью упрощения можно считать, что речь идет о результирующем наборе запроса. Более подробное описание этого процесса будет дано далее.

Гораздо важнее отметить, что в концептуальном смысле инструкция *OVER* определяет окно для функции относительно текущей строки. И это справедливо для всех строк в итоговом наборе данных. Иными словами, для каждой строки инструкция *OVER* определяет окно, независимое от окон, определенных для других строк. На самом деле это очень глубокая идея, понимание которой может прийти не сразу. Но когда вы осознаете всю полноту этого утверждения, вы сможете сказать, что постигли концепцию оконных функций, всю ее мощь и глубину. Если пока для вас это пустой звук, не отчаивайтесь. Я просто бросил семя в почву, ростка придется подождать.

Первое упоминание об оконных функциях в стандарте SQL появилось в расширенном документе к SQL:1999 в разделе «OLAP функции». С тех пор каждая версия стандарта – SQL:2003, SQL:2008, SQL:2011 и SQL:2016 – расширяла возможности оконных функций, а в последней версии поддержка этой технологии достигла небывалого уровня. Кроме того, оконные функции приобрели такой аналитический механизм, как распознавание шаблонов в строках, что говорит о намерениях рабочей группы, ответственной за стандарт SQL, продолжать развивать это направление.

Вы можете купить официальные документы стандарта SQL от ISO или ANSI. К примеру, базовый документ от ISO по стандарту SQL:2016, описывающий все ключевые конструкции языка, можно приобрести по адресу *https://www.iso.org/standard/63556.html*.

Стандарт SQL поддерживает несколько типов оконных функций: агрегатные, ранжирующие, аналитические или статистические (функции распределения) и функции смещения. Но помните, что работа с окнами в SQL представляет собой целую концепцию, так что в будущих версиях стандарта мы можем увидеть и новые типы оконных функций.

Агрегатные оконные функции (aggregate window functions) – это все те же функции агрегирования, с которыми вы давно и хорошо знакомы (*SUM*, *COUNT*, *MIN*, *MAX* и другие), но которые привыкли применять в контексте запросов с группировками. Агрегатные функции по определению работают с наборами данных, будь то наборы, определенные запросами с группировками или спецификацией окна.

К *ранжирующим функциям* (ranking functions) относятся следующие: *RANK*, *DENSE_RANK*, *ROW_NUMBER* и *NTILE*. При этом первые и последние две функции относятся в стандарте к разным категориям, и позже мы выясним причину этого. Я предпочитаю для простоты относить все четыре функции к одной категории по примеру официальной документации к SQL Server.

Статистические функции (statistical functions), или функции распределения, включают в себя функции *PERCENTILE_CONT*, *PERCENTILE_DISC*, *PERCENT_RANK* и *CUME_DIST*. Эти функции предназначены для расчета статистических показателей, таких как процентили, процентные ранги и накопительные распределения.

К *функциям смещения* (offset functions) относятся функции *LAG*, *LEAD*, *FIRST_VALUE*, *LAST_VALUE* и *NTH_VALUE*. SQL Server поддерживает первые четыре из этого списка. Функция *NTH_VALUE* в версии SQL Server 2019 не реализована.

В главе 2 мы подробно поговорим обо всех перечисленных здесь функциях.

Новые идеи, устройства или инструменты, даже если они очевидно превосходят своих предшественников в простоте использования, всегда являются определенным барьером. Все новое на первый взгляд кажется сложным. Так что если оконные функции для вас в новинку и вы ищете мотивацию в их освоении, я поделюсь несколькими наблюдениями из собственного опыта:

- оконные функции помогают в решении огромного спектра задач. Я даже не знаю, как еще лучше это акцентировать. Как я уже упоминал ранее, в настоящее время я применяю оконные функции практически во всех своих решениях. Заключительная шестая глава этой книги будет посвящена практическим примерам использования этих функций. А пока я лишь перечислю типы задач, с которыми оконные функции справляются легко и непринужденно:
  - постраничный вывод;
  - удаление дубликатов;
  - возвращение верхних *n* строк по группам;
  - вычисление накопительных итогов;
  - выполнение операций над интервалами, таких как объединение интервалов и вычисление максимального количества пересекающихся интервалов;
  - определение пропусков и островов;
  - вычисление процентилей;
  - вычисление моды распределения;
  - сортировка иерархий;
  - сведение данных;
  - определение новизны данных;
- я пишу запросы на языке SQL вот уже почти три десятилетия и последние несколько лет очень активно использую оконные функции. Могу сказать, что, несмотря на время, которое потребовалось на доскональное освоение этой концепции, сейчас применение оконных функций при решении самых разнообразных задач кажется мне гораздо более эффективным и интуитивным по сравнению с традиционными методами;
- оконные функции поддаются оптимизации, и далее в этой книге мы поговорим об этом более подробно.

Декларативный язык и оптимизация

Вас может удивить то, что в случае использования декларативного языка SQL, в котором вы логически объявляете свой запрос, а не описываете, как именно должен быть получен результат, две разновидности одного и того же запроса (к примеру, с использованием оконных функций и без) могут давать разную производительность. Почему движок SQL Server, использующий диалект языка T-SQL, не всегда способен понять, что две записи означают одно и то же, а значит, должны приводить к созданию одинаковых планов выполнения?

На то есть свои причины. Во-первых, оптимизатор SQL Server не идеален. Я не хочу показаться неблагодарным, ведь оптимизатор запросов, использующийся в SQL Server, представляет собой настоящий шедевр искусства создания программного обеспечения. Но факт в том, что даже в нем не реализованы все без исключения правила оптимизации запросов. Во-вторых, движок ограничен во времени, которое отводится на оптимизацию. Иначе могло бы произойти так, что на саму оптимизацию тратилось бы больше времени, чем можно было бы сэкономить за счет нее. Для любого запроса можно придумать бессчетное количество планов выполнения, и на проверку каждого просто не хватит времени. На основании факторов, таких как размер таблиц, участвующих в запросе, SQL Server вычисляет два значения: *стоимость* (cost), которая может считаться приемлемой для данного запроса, и максимальное количество времени, которое может быть затрачено на выполнение оптимизации. По достижении любого из этих значений оптимизация прекращается, и SQL Server использует лучший из найденных на этот момент времени планов выполнения запроса.

При этом оконные функции, о которых мы будем говорить в этой книге, зачастую могут быть оптимизированы гораздо более эффективно по сравнению с традиционными конструкциями при достижении одной и той же цели.

Здесь важно понимать, что вы должны сделать абсолютно осознанный выбор, переключаясь на использование оконных функций, поскольку это совершенно иная концепция, которую нужно использовать во благо. К работе с окнами в SQL привыкнуть бывает непросто. Но по прохождении периода адаптации вы осознаете всю легкость и интуитивность этой технологии. Вспомните, как долго вы привыкали к современным гаджетам, без которых сегодня просто не мыслите жизни.

Программирование на основе наборов данных и курсоров/итераций

Подходы, применяемые в диалекте T-SQL к решению задач, можно условно разделить на *декларативный* (declarative), использующий в своей основе наборы данных, и *итеративный* (iterative), базирующийся на курсорах. При этом в среде разработчиков T-SQL есть полное единодушие в пользу первого под-

хода. Но в то же время существует масса решений, в которых по-прежнему активно применяются итерации. Здесь возникает ряд интересных вопросов. Почему вариант с использованием наборов данных является более предпочтительным? И если это так, почему многие разработчики применяют итеративный подход? Что мешает им переключиться на рекомендованный декларативный стиль?

Чтобы докопаться до сути поставленных вопросов, сначала необходимо разобраться в основах T-SQL и понять, что из себя представляет подход, базирующийся на наборах данных. В процессе вы осознаете, что декларативные принципы для многих являются далеко не самыми интуитивными, тогда как итеративные кажутся простыми и понятными. Просто так устроен наш мозг, и мы позже поговорим об этом подробнее. Пропасть между программированием на основе наборов данных и курсоров просто огромна. Сократить ее можно, но сделать это очень непросто. И здесь на помощь приходят оконные функции. Лично я рассматриваю их как прочный мост между двумя концепциями, позволяющий обеспечить более плавный и легкий переход к мышлению на основе наборов данных.

Но для начала нужно ясно проговорить, что из себя представляет декларативный подход, базирующийся на выполнении операций с наборами данных, применительно к запросам на языке T-SQL. *T-SQL* – это диалект стандарта SQL (обоих стандартов – ISO/IEC и ANSI). SQL основывается (вернее, пытается это делать) на *реляционной модели* (relational model). Реляционная модель представляет собой математическую модель для управления и манипулирования данными, которая была сформулирована и предложена Эдгаром Франком Коддом (E. F. Codd) в конце 1960-х. Реляционная модель базируется на двух математических принципах: *теории множеств* (set theory) и *логике предикатов* (predicate logic). Многие аспекты вычислений были разработаны на основе чистой интуиции, и они до сих продолжают меняться достаточно динамично – до такой степени, что порой кажется, что ты носишься за собственным хвостом. Реляционная модель – это остров в этом мире вычислений, поскольку она основывается на гораздо более строгих математических постулатах. Многие считают, что с математикой не поспоришь. И, будучи основанной исключительно на математических принципах, реляционная модель получила славу очень логичной и надежной субстанции. Эта модель продолжает развиваться, но не так стремительно, как другие аспекты вычислений. На протяжении последних нескольких десятилетий реляционная модель прочно удерживает свои позиции и по-прежнему является основой всех современных лидирующих *систем управления реляционными базами данных* (relational database management system – RDBMS).

SQL стал своеобразной попыткой создать язык на основе реляционной модели. Но SQL не идеален и в некоторых аспектах отклоняется от принципов реляционной модели. В то же время он предоставляет достаточный выбор инструментов, чтобы при должном понимании концепции реляционной модели можно было использовать язык SQL на основе реляционных принципов. По большому счету SQL удерживает безоговорочное первенство и фактически является единственным языком данных.

Однако, как я уже упоминал, мыслить категориями реляционной модели для многих очень непривычно и даже противоестественно. Основной блок со-

держится в различиях между подходом, основывающимся на наборах данных, и итеративным подходом. И особенно трудно приходится тем, у кого есть за плечами опыт программирования на процедурных языках, в которых привычным способом чтения данных из файлов являются итерации, как показано на примере псевдокода ниже:

```
открываем файл
извлекаем первую запись
пока не достигнут конец файла
начало
    обрабатываем запись
    извлекаем следующую запись
конец
```

Данные в файлах или, если быть более точными, в *индексно-последовательном методе доступа* (indexed sequential access method – ISAM) хранятся в заданном порядке, и в процессе чтения информация извлекается в точности в том же порядке, в котором была записана. Кроме того, данные извлекаются порциями – по одной за раз. Таким образом, наш мозг запрограммирован воспринимать работу с данными именно так: порядок строго задан, получаем по одной записи. Это аналогично тому, как в T-SQL работают *курсоры* (cursor), и программисты с опытом работы с процедурными языками склонны использовать именно их или другие техники выполнения итераций в SQL, воспринимая это как естественное продолжение того, что они уже знают.

Реляционный подход, основанный на наборах или множествах данных, проповедует совершенно иные принципы работы с информацией. Чтобы проникнуть в самую их суть, давайте для начала посмотрим на определение *множества* (set), данное основателем теории множеств Георгом Кантором (Georg Cantor):

> Под «множеством» мы подразумеваем любое объединение M в одно целое, состоящее из определенных различающихся объектов m (называемых «элементами» M) в области наших мыслей и восприятий.
>
> –*Джозеф Даубен* (Joseph W. Dauben), *Георг Кантор* (Georg Cantor) (Princeton University Press, 1990)

В этом определении множества столько всего, что можно было бы не одну страницу посвятить его объяснению. Но я предпочел сфокусировать внимание на двух ключевых аспектах, один из которых присутствует в определении явно, а второй подразумевается:

- **целое.** Обратите внимание на использование в определении множества термина *целое*. Множество должно восприниматься и управляться как единое *целое*. И вы должны работать с ним как с общностью данных, а не разрозненным набором элементов. В процессе выполнения итеративного перебора это правило нарушается, поскольку записи из файла или курсора обрабатываются по одной. Таблица в SQL представляет (хотя и не совсем успешно) отношение в реляционной модели. *Отношением* (relation) называется общность похожих элементов, т. е. обладающих одинаковым

набором атрибутов. Взаимодействуя с таблицами при помощи запросов, базирующихся на наборах данных, вы работаете с ними как с *целым*, а не как с набором отдельных строк (*кортежей* (tuple) в отношении), как в отношении формулирования декларативных запросов SQL, так и с точки зрения мышления и общего подхода. Приобрести подобный взгляд на вещи для многих является огромной проблемой;

- **отсутствие строгого порядка.** Обратите внимание, что в определении множества ничего не сказано про порядок расположения элементов в нем. И это не просто так – в множестве элементы действительно не упорядочены. Это еще одна вещь, к которой людям бывает трудно привыкнуть при смене концепции. Файлы и курсоры *обладают* строгим порядком следования записей, и когда вы запускаете итерации в запросе, то записи извлекаются строго в порядке их следования. Таблица, будучи множеством, не обладает строгим порядком следования строк. Те, кто этого не понимает, часто путают логический слой модели данных и язык с физическим слоем реализации. Они напрасно предполагают, что если в таблице есть индекс, извлечение данных из нее будет производиться строго в порядке следования этого индекса. Иногда от этого неверного предположения может зависеть работоспособность решения. Разумеется, SQL Server не дает никаких подобных гарантий. К примеру, единственный способ гарантировать следование записей в результирующем наборе определенному порядку – снабдить запрос инструкцией *ORDER BY*. А сделав это, вы должны понимать, что возвращенный результат нельзя будет считать реляционным именно из-за гарантированного порядка следования строк в нем.

Если вы пишете запросы на SQL и хотите досконально знать этот язык, вам просто необходимо начать мыслить терминами наборов данных. И именно в этом вам помогут оконные функции, которые могут служить мостиком между итеративным мышлением (с гарантированным порядком и последовательной обработкой) и концепцией множеств (восприятие набора данных как единое целое, без строгого порядка). А удобная и логичная реализация оконных функций поможет вам осуществить этот переход легко и непринужденно.

В действительности оконные функции также располагают инструкцией упорядочивания данных в окне (*ORDER BY*), которой можно пользоваться при необходимости. Но наличие этой возможности еще не говорит о нарушении реляционных правил. На вход запросу поступает отношение, в котором данные не упорядочены, и на выходе мы также получаем отношение. Упорядочивание данных в этом случае используется при выполнении расчетов внутри окна с целью размещения атрибута в результирующем наборе, представляющем собой полноценное отношение. Нет никакой гарантии, что в результирующем наборе строки будут расположены в том же порядке, который был определен при расчетах внутри окна, – фактически разные оконные функции в одном и том же запросе могут располагать итоговые данные в наборе по-разному. Получается, что порядок сортировки окна никак не влияет – по крайней мере концептуально – на следование строк на выходе запроса. На рис. 1.2 продемонстрировано, что входные и выходные наборы данных запроса, использую-

щего оконную функцию, являются реляционными, несмотря на использование инструкции сортировки внутри запроса. Используя овалы, объединяющие входные и выходные данные, и расположив сами элементы хаотично, я хотел показать, что порядок следования строк здесь не имеет значения.

Есть еще один аспект оконных функций, который может помочь вам постепенно переключиться с итеративного восприятия данных на оперирование целыми множествами. При подаче нового материала преподаватель иногда вынужден что-то недоговаривать. Представьте, что вы как учитель знаете, что студент пока не готов к полноценному восприятию новой темы. В таких случаях бывает полезно предложить материал в упрощенном виде – не всегда полностью корректном, – чтобы дать студенту возможность проникнуться новой для него идеей. Позже, когда ученик будет готов услышать всю правду, можно подать концепцию в развернутом виде со всеми нюансами.

Такой подход применим и к подаче материала об оконных функциях. Можно в целом описать идею, но при этом она будет концептуально не до конца корректной, хоть и приводить к правильным результатам. В этом базовом описании можно использовать допуск о том, что строки во входных данных обрабатываются одна за другой по порядку. А уже в последующем полном описании технологии можно упомянуть, что набор данных на самом деле обрабатывается как единое целое, без итераций.

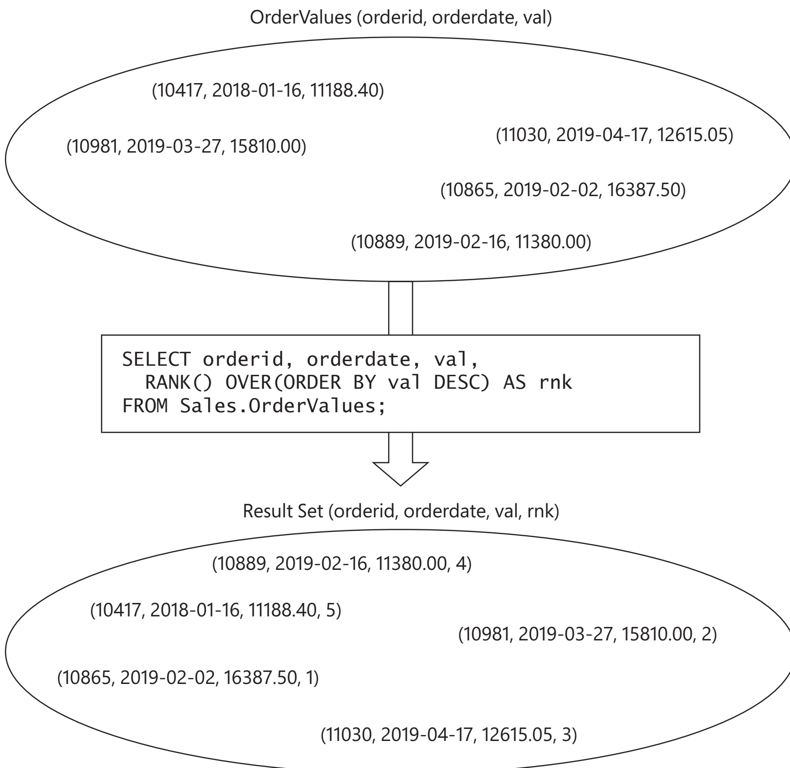

Рис. 1.2 Вход и выход запроса, использующего оконную функцию

Чтобы понять, что я имею в виду, рассмотрим следующий запрос:

```
SELECT orderid, orderdate, val,
    RANK() OVER(ORDER BY val DESC) AS rnk
FROM Sales.OrderValues;
```

Ниже приведен фрагмент вывода запроса. Обратите внимание, что порядок следования строк здесь не предопределен:

```
orderid  orderdate  val       rnk
-------- ---------- --------- ----
10865    2019-02-02 16387.50  1
10981    2019-03-27 15810.00  2
11030    2019-04-17 12615.05  3
10889    2019-02-16 11380.00  4
10417    2018-01-16 11188.40  5
...
```

Давайте при помощи псевдокода попробуем определить, как происходит расчет значений рангов:

```
сортируем данные по полю val в порядке убывания
проходим по строкам
для каждой строки
    если текущая строка является первой строкой в секции, вернуть 1 (в отсутствие явного
секционирования воспринимать весь набор данных как единственную секцию)
    иначе если значение в поле val равно предыдущему значению, вернуть предыдущий ранг
    иначе вернуть количество обработанных строк
```

На рис. 1.3 показано графически, как может выполняться описанная выше построчная обработка данных.

```
orderid   orderdate   val        rnk
--------- ----------- ---------  ------
10865     2019-02-02  16387.50   1
10981     2019-03-27  15810.00   2
11030     2019-04-17  12615.05   3
10889     2019-02-16  11380.00   4
10417     2018-01-16  11188.40   5
…
```

Рис. 1.3 Базовое восприятие алгоритма расчета рангов

Опять же, несмотря на получение правильных результатов, сама концепция описана здесь не совсем корректно. Кроме того, донести свою мысль мне мешает еще и то, что движок SQL на физическом уровне вычисляет ранги примерно так, как я только что описал. Но наша цель – не физический уровень, а концептуальный – на уровне языка и логической модели. Когда я говорю о некорректном мышлении, я имею в виду прежде всего то, что с точки зрения языка расчеты выполняются иначе, а именно на основе наборов данных, а не итераций. Помните, что язык никак не связан с реализацией движка на физическом уровне. Задача физического уровня состоит в обработке логического

запроса, максимально быстром его выполнении и возвращении правильных результатов.

Позвольте мне пояснить, что я имею в виду под более глубоким и правильным пониманием того, как язык воспринимает оконные функции. Функция логически определяет для каждой строки в результирующем наборе данных запроса отдельное, независимое окно. В отсутствие каких-либо ограничений спецификации окна каждое из окон будет состоять из набора всех строк результата запроса. При этом вы можете добавить элементы в спецификацию окна (например, секционирование, определение границ окна и т. д.), которые будут дополнительно ограничивать набор строк для каждого из них. Об этих и других элементах мы будем подробно говорить далее. На рис. 1.4 показано графическое представление этой идеи применительно к функции *RANK*.

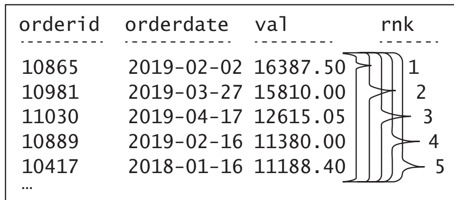

Рис. 1.4 Глубокое понимание вычисления значений рангов

Инструкция *OVER* концептуально создает отдельные окна для каждой оконной функции и строки в результирующем наборе данных запроса. В нашем примере мы никак не ограничили спецификацию окна, а лишь задали способ упорядочивания для вычисления. Таким образом, все наши окна состоят из всех строк в итоговом наборе данных запроса. При этом они все сосуществуют одновременно. В каждом окне ранг рассчитывается как число, на единицу превосходящее количество строк с большим значением атрибута *val* по сравнению с его текущим значением.

Как видите, гораздо более интуитивным является представление о том, что записи в наборе изначально упорядочены и перебираются по одной. Это нормально для этапа знакомства с оконными функциями, поскольку помогает правильно писать запросы, хотя бы самые простые. Со временем вы станете понимать оконные функции лучше и начнете мыслить не итерациями, а наборами данных.

Недостатки альтернативных вариантов оконных функций

Оконные функции обладают некоторыми явными преимуществами по сравнению с традиционными методами для достижения тех же результатов, также основанными на работе с наборами данных, в число которых входит использование запросов с группировками, вложенных запросов и др. В данном разделе мы рассмотрим конкретные примеры. При этом я сразу хочу заметить, что у оконных функций есть и другие существенные преимущества перед своими аналогами, помимо тех, что мы разберем здесь, но вам о них знать пока рановато.

Начнем с традиционных запросов с группировками. Эти запросы позволяют вам проанализировать ранее недоступную информацию, но только для агрегированных значений, без возможности посмотреть детализацию.

Применяя группировку, вы ограничиваете свои вычисления исключительно контекстом групп. А что, если вам необходимо применить калькуляции, затрагивающие как агрегаты, так и исходные данные? К примеру, вы хотите обратиться к представлению *Sales.OrderValues* и вычислить для каждого заказа его долю от общей суммы заказов по данному клиенту, а также разницу по сравнению со средней суммой заказов. Сумма текущего заказа представляет собой детализированные данные, а сумма и среднее по всем заказам для клиента – агрегированные значения. Если выполнить группировку по клиентам, вы утратите доступ к суммам конкретных заказов. Один из способов решения поставленной задачи состоит в создании *табличного выражения* (table expression) на основании запроса с группировкой и *объединении* (join) его с исходной таблицей на предмет выявления соответствий. Ниже показан запрос, реализующий описанный выше подход:

```
WITH Aggregates AS
(
    SELECT custid, SUM(val) AS sumval, AVG(val) AS avgval
    FROM Sales.OrderValues
    GROUP BY custid
)
SELECT O.orderid, O.custid, O.val,
    CAST(100. * O.val / A.sumval AS NUMERIC(5, 2)) AS pctcust,
    O.val - A.avgval AS diffcust
FROM Sales.OrderValues AS O
    INNER JOIN Aggregates AS A
        ON O.custid = A.custid;
```

Результирующий набор данных этого запроса (в сокращенном виде) будет таким:

```
orderid  custid  val     pctcust  diffcust
-------- ------- ------- -------- ------------
10835    1       845.80  19.79    133.633334
10952    1       471.20  11.03    -240.966666
10643    1       814.50  19.06    102.333334
10692    1       878.00  20.55    165.833334
11011    1       933.50  21.85    221.333334
10702    1       330.00  7.72     -382.166666
10625    2       479.75  34.20    129.012500
10759    2       320.00  22.81    -30.737500
10308    2       88.80   6.33     -261.937500
10926    2       514.40  36.67    163.662500
...
```

А теперь представьте, что вам также понадобилось извлечь долю заказа от общего итога и разницу с общим средним. Для этого вам придется создавать еще одно табличное выражение, как показано ниже:

```
WITH CustAggregates AS
(
    SELECT custid, SUM(val) AS sumval, AVG(val) AS avgval
    FROM Sales.OrderValues
    GROUP BY custid
),
GrandAggregates AS
(
    SELECT SUM(val) AS sumval, AVG(val) AS avgval
    FROM Sales.OrderValues
)
SELECT O.orderid, O.custid, O.val,
    CAST(100. * O.val / CA.sumval AS NUMERIC(5, 2)) AS pctcust,
    O.val - CA.avgval AS diffcust,
    CAST(100. * O.val / GA.sumval AS NUMERIC(5, 2)) AS pctall,
    O.val - GA.avgval AS diffall
FROM Sales.OrderValues AS O
    INNER JOIN CustAggregates AS CA
        ON O.custid = CA.custid
    CROSS JOIN GrandAggregates AS GA;
```

Сокращенный вывод этого запроса будет следующим:

```
orderid  custid  val     pctcust  diffcust    pctall  diffall
-------- ------- ------- -------- ----------- ------- ------------
10835    1       845.80  19.79    133.633334  0.07    -679.252072
10952    1       471.20  11.03    -240.966666 0.04    -1053.852072
10643    1       814.50  19.06    102.333334  0.06    -710.552072
10692    1       878.00  20.55    165.833334  0.07    -647.052072
11011    1       933.50  21.85    221.333334  0.07    -591.552072
10702    1       330.00  7.72     -382.166666 0.03    -1195.052072
10625    2       479.75  34.20    129.012500  0.04    -1045.302072
10759    2       320.00  22.81    -30.737500  0.03    -1205.052072
10308    2       88.80   6.33     -261.937500 0.01    -1436.252072
10926    2       514.40  36.67    163.662500  0.04    -1010.652072
...
```

Как видите, с каждым новым требованием запрос становится все более сложным, с использованием дополнительных табличных выражений и объединений.

Еще один способ выполнить требуемый расчет состоит в использовании отдельных *подзапросов* (subquery) для каждого вычисления. Ниже показаны аналогичные запросы с применением подзапросов вместо группировок:

```
-- подзапросы, детализация и агрегаты по клиентам
SELECT orderid, custid, val,
    CAST(100. * val /
        (SELECT SUM(O2.val)
        FROM Sales.OrderValues AS O2
        WHERE O2.custid = O1.custid) AS NUMERIC(5, 2)) AS pctcust,
    val - (SELECT AVG(O2.val)
        FROM Sales.OrderValues AS O2
        WHERE O2.custid = O1.custid) AS diffcust
```

```
FROM Sales.OrderValues AS O1;

-- подзапросы, детализация, агрегаты по клиентам и общие итоги
SELECT orderid, custid, val,
    CAST(100. * val /
        (SELECT SUM(O2.val)
        FROM Sales.OrderValues AS O2
        WHERE O2.custid = O1.custid) AS NUMERIC(5, 2)) AS pctcust,
    val - (SELECT AVG(O2.val)
        FROM Sales.OrderValues AS O2
        WHERE O2.custid = O1.custid) AS diffcust,
    CAST(100. * val /
        (SELECT SUM(O2.val)
        FROM Sales.OrderValues AS O2) AS NUMERIC(5, 2)) AS pctall,
    val - (SELECT AVG(O2.val)
        FROM Sales.OrderValues AS O2) AS diffall
FROM Sales.OrderValues AS O1;
```

У подхода с использованием подзапросов есть два существенных недостатка. Первый из них – это длинный и плохо читаемый код. Второй состоит в том, что на данный момент оптимизатор SQL Server не умеет корректно обрабатывать ситуации, когда разные подзапросы обращаются к одному и тому же набору данных. Таким образом, здесь мы дважды будем извлекать одни и те же данные. И чем больше подзапросов у нас будет, тем больше будет обращений. В отличие от первого недостатка, этот не связан напрямую с языком, а относится к нюансам оптимизации запросов в SQL Server.

Помните, что идея оконных функций состоит в определении набора строк или окна, в рамках которого выполняется функция. Агрегатные функции по своей природе выполняются над наборами данных, так что применительно к ним прекрасно работает концепция создания окон в качестве альтернативы запросам с группировками и подзапросами. Но при этом выполнение агрегатной оконной функции не приводит к потере детальных исходных данных. С целью определения окна для такой функции необходимо использовать инструкцию *OVER*. Например, для вычисления суммарных значений из результирующего набора данных используйте следующую простую конструкцию:

```
SUM(val) OVER()
```

Не ограничивая окно (оставляя скобки пустыми), вы тем самым указываете на то, что окно будет включать в себя все записи результирующего набора данных.

Если вам необходимо рассчитать агрегаты по строкам, в которых идентификатор клиента равен идентификатору в текущей строке, используйте *секционирование* (partitioning), о котором подробно мы будем говорить позже, как показано ниже:

```
SUM(val) OVER(PARTITION BY custid)
```

Обратите внимание, что секционирование в оконных функциях работает как фильтрация, поскольку в результате его применения исходный набор данных ограничивается.

С использованием оконных функций наша задача решается легко и элегантно, при этом мы одновременно получаем доступ и к исходным детализированным данным, и к агрегатам. В примере ниже оконные функции выделены жирным:

```
SELECT orderid, custid, val,
    CAST(100. * val / SUM(val) OVER(PARTITION BY custid) AS NUMERIC(5, 2)) AS pctcust,
    val - AVG(val) OVER(PARTITION BY custid) AS diffcust
FROM Sales.OrderValues;
```

А в следующем запросе мы добавили поддержку вычисления общих итогов:

```
SELECT orderid, custid, val,
    CAST(100. * val / SUM(val) OVER(PARTITION BY custid) AS NUMERIC(5, 2)) AS pctcust,
    val - AVG(val) OVER(PARTITION BY custid) AS diffcust,
    CAST(100. * val / SUM(val) OVER() AS NUMERIC(5, 2)) AS pctall,
    val - AVG(val) OVER() AS diffall
FROM Sales.OrderValues;
```

Только посмотрите, насколько запросы с применением оконных функций стали более простыми и понятными. Также очень важно отметить, что оптимизатор SQL Server умеет учитывать ситуации, когда разные функции обращаются к одной и той же спецификации окна. В этом случае он осуществляет повторное чтение из уже вычисленного набора данных, не рассчитывая его повторно. Таким образом, в последнем нашем примере движок SQL Server будет дважды обращаться к одному предрассчитанному окну для первых двух функций (с секционированием по полю *custid*) и дважды – ко второму (без секционирования). Подробно о концепциях оптимизации оконных функций мы поговорим в главе 5.

Еще одним явным преимуществом оконных функций над подзапросами является то, что исходным набором данных до наложения всех ограничений для них является результирующий набор данных запроса. Это означает, что секционирование и прочие операции будут применяться к результирующему набору уже после выполнения табличных операторов (например, объединений), фильтрации (с помощью инструкции *WHERE*), группировок и групповой фильтрации (*HAVING*). Это происходит из-за того, что оконные функции выполняются в фазе логической обработки запроса. Подробнее об этом мы поговорим далее в этой главе. Что касается подзапросов, то они начинают свое выполнение с нуля, а не на базе родительского запроса. Таким образом, если вы хотите, чтобы подзапрос выполнялся в том же наборе данных, что и базовый запрос, вам придется повторить для него все конструкции, присутствующие в общем запросе. Допустим, вам нужно, чтобы наши вычисления производились только для заказов, размещенных в 2018 году. В случае с использованием оконных функций все, что вам необходимо сделать, это добавить один фильтр в запрос, как показано ниже:

```
SELECT orderid, custid, val,
    CAST(100. * val / SUM(val) OVER(PARTITION BY custid) AS NUMERIC(5, 2)) AS pctcust,
```

```
    val - AVG(val) OVER(PARTITION BY custid) AS diffcust,
    CAST(100. * val / SUM(val) OVER() AS NUMERIC(5, 2)) AS pctall,
    val - AVG(val) OVER() AS diffall
FROM Sales.OrderValues
WHERE orderdate >= '20180101'
    AND orderdate < '20190101';
```

В этом случае точкой входа в оконные функции будет наш результирующий набор данных после применения фильтра. Что касается варианта с подзапросами, показанного ниже, вам придется добавлять один и тот же фильтр для всех вложенных запросов:

```
SELECT orderid, custid, val,
    CAST(100. * val /
        (SELECT SUM(O2.val)
        FROM Sales.OrderValues AS O2
        WHERE O2.custid = O1.custid
            AND orderdate >= '20180101'
            AND orderdate < '20190101') AS NUMERIC(5, 2)) AS pctcust,
    val - (SELECT AVG(O2.val)
        FROM Sales.OrderValues AS O2
        WHERE O2.custid = O1.custid
            AND orderdate >= '20180101'
            AND orderdate < '20190101') AS diffcust,
    CAST(100. * val /
        (SELECT SUM(O2.val)
        FROM Sales.OrderValues AS O2
            WHERE orderdate >= '20180101'
            AND orderdate < '20190101') AS NUMERIC(5, 2)) AS pctall,
    val - (SELECT AVG(O2.val)
        FROM Sales.OrderValues AS O2
        WHERE orderdate >= '20180101'
            AND orderdate < '20190101') AS diffall
FROM Sales.OrderValues AS O1
WHERE orderdate >= '20180101'
    AND orderdate < '20190101';
```

Конечно, вы могли бы использовать и обходные пути, например сначала можно было бы объявить *обобщенное табличное выражение* (common table expression – CTE) на базе запроса, выполняющего фильтрацию, а затем из внешнего и внутренних запросов обращаться к этому табличному выражению. В случае с использованием оконных функций вам не придется искать обходные пути, поскольку они выполняются на основании результирующего набора данных запроса. Позже в этой главе мы подробно обсудим этот аспект разработки оконных функций.

Кроме того, как уже упоминалось ранее, оконные функции поддаются эффективной оптимизации, тогда как об альтернативных вариантах такого, увы, не скажешь. Разумеется, встречаются и исключения из этого правила. В главе 5 мы будем подробно говорить об оптимизации оконных функций, а в главе 6 приведем множество примеров их эффективного использования.

Ваши первые решения с применением оконных функций

В первых пяти главах книги мы будем в основном говорить об оконных функциях, их оптимизации и сопутствующих аналитических возможностях. Материал будет содержать довольно много технических терминов, и, хотя лично мне он кажется весьма захватывающим, кто-то может посчитать его чересчур сложным и сухим. Разумеется, многим больше нравится читать о практическом применении функций и решении с их помощью конкретных задач, чему будет посвящена заключительная глава книги. Всю безграничную мощь оконных функций можно понять только на практике – глядя, как элегантно и эффективно они помогают решать сложнейшие задачи. Как же мне удержать ваше внимание до последней главы и сделать так, чтобы вы не потеряли интерес раньше времени? А что, если мы прямо сейчас посмотрим, как можно применять оконные функции при решении насущных задач?

В нашем примере мы рассмотрим обращение к таблице, содержащей последовательность значений в столбце, на предмет поиска последовательных диапазонов. Эта задача также известна как *задача на поиск островов* (islands problem). Последовательность при этом может быть числовой, временной (даже чаще) или любого другого типа, поддерживающего сортировку. Кроме того, она может содержать как уникальные значения, так и дублирующиеся. Интервалы для поиска в последовательности могут быть любой фиксированной длины, соотносящейся с выбранным типом данных. Например, для числового столбца это могут быть значения 1 или 7, для столбца с временными интервалами – один день или две недели. В главе 6 мы будем разбирать другие варианты этой задачи. Сейчас же я покажу вам простой пример, чтобы вы поняли, как это работает. Будем использовать для хранения последовательности числовой столбец и искать интервалы, равные единице. Используйте приведенный ниже код для создания исходной таблицы и заполнения ее значениями:

```
SET NOCOUNT ON;
USE TSQLV5;

DROP TABLE IF EXISTS dbo.T1;
GO

CREATE TABLE dbo.T1
(
    col1 INT NOT NULL
        CONSTRAINT PK_T1 PRIMARY KEY
);

INSERT INTO dbo.T1(col1)
    VALUES(2),(3),(11),(12),(13),(27),(33),(34),(35),(42);
```

Как видите, в значениях столбца *col1* присутствуют разрывы. Наша задача – выделить последовательные диапазоны существующих значений (также

известные как острова) и вернуть начальное и конечное значения для каждого острова. Ниже показано, как выглядит ожидаемый результат:

```
startrange  endrange
----------- -----------
2           3
11          13
27          27
33          35
42          42
```

Если вам интересно практическое применение этой задачи, она используется в огромном множестве областей. С помощью подобных расчетов формируются отчеты о наличии товаров или периодах активности (например, продаж), происходит идентификация последовательных периодов по заданному критерию (допустим, периодов, когда стоимость запасов была выше или ниже определенного порогового значения), определение диапазонов автомобильных номеров в наличии и т. д. Наш пример будет достаточно простым, чтобы вы могли сосредоточиться на используемой технике. В реальных, более сложных задачах вы будете использовать показанные здесь приемы с незначительными доработками. Попробуйте сами выработать эффективное решение для этой задачи, основанное на наборах данных. Для начала примените его к небольшому набору из этого примера. После этого заполните таблицу, внеся в нее порядка 10 млн значений, и опробуйте свою технику снова. Сравните результаты и только после этого приступайте к исследованию предложенного мной решения.

Перед тем как приступать к использованию оконных функций, я предложу одно из многочисленных решений с применением традиционных конструкций. В частности, давайте используем для решения этой задачи подзапросы. Для понимания техники первого решения давайте сначала присмотримся к значениям в столбце *T1.col1* с добавленным концептуальным атрибутом, не существующим в данный момент, который мы будем рассматривать в качестве идентификатора группы:

```
col1 grp
----- ---
2     a
3     a
11    b
12    b
13    b
27    c
33    d
34    d
35    d
42    e
```

Атрибут *grp* пока не создан. Концептуально значение в этом столбце уникально идентифицирует остров. Таким образом, буквы атрибута должны совпадать для значений, входящих в один остров, и различаться для значений,

принадлежащих разным островам. Если вы сможете вычислить этот идентификатор, далее вам останется лишь сгруппировать по нему и вернуть минимальное и максимальное значения из столбца *col1* для каждой группы (острова). Одним из способов определения группового идентификатора с применением традиционных методов SQL является вычисление для каждого текущего значения столбца *col1* минимального значения из этого же столбца, большего или равного текущему и не имеющего следующего значения.

Давайте, следуя этой логике, попробуем для числа 2 определить минимальное значение в столбце *col1*, которое больше или равно 2 и находится перед разрывом. Это число 3. Теперь давайте найдем такое значение для числа 3. Это также 3. Таким образом, групповой идентификатор 3 определяет остров, начинающийся числом 2 и заканчивающийся числом 3. Для острова, начинающегося числом 11 и заканчивающегося числом 13, идентификатором группы будет 13. Как видите, общим идентификатором для острова становится последнее входящее в него значение.

Ниже приведен запрос на языке T-SQL для реализации этой техники:

```
SELECT col1,
    (SELECT MIN(B.col1)
        FROM dbo.T1 AS B
        WHERE B.col1 >= A.col1
            -- Это последняя строка в своей группе?
            AND NOT EXISTS
                (SELECT *
                    FROM dbo.T1 AS C
                    WHERE C.col1 = B.col1 + 1)) AS grp
FROM dbo.T1 AS A;
```

Этот запрос сгенерирует следующий вывод:

```
col1        grp
----------- -----------
2           3
3           3
11          13
12          13
13          13
27          27
33          35
34          35
35          35
42          42
```

Дальше все просто: можно определить табличное выражение на основе предыдущего запроса, во внешнем запросе выполнить группировку по идентификатору и вернуть минимальное и максимальное значения из столбца *col1* для каждой группы следующим образом:

```
SELECT MIN(col1) AS startrange, MAX(col1) AS endrange
FROM (SELECT col1,
    (SELECT MIN(B.col1)
```

```
        FROM dbo.T1 AS B
        WHERE B.col1 >= A.col1
            AND NOT EXISTS
                (SELECT *
                    FROM dbo.T1 AS C
                    WHERE C.col1 = B.col1 + 1)) AS grp
    FROM dbo.T1 AS A) AS D
GROUP BY grp;
```

С этим решением есть две проблемы. Во-первых оно как-то чересчур сложновато. Во-вторых, катастрофически медленно работает. Сейчас мы не будем обращаться к планам выполнения запросов – позже в этой книге вы еще успеете устать от этого добра, – но поверьте мне на слово, что в данном случае для каждой строки в таблице SQL Server будет выполнять почти два полных сканирования данных. А теперь представьте, что у вас в таблице не десять записей, а 10 млн. Вывод очевиден. Общее число строк, которые должны будут подвергнуться обработке, будет просто огромным, а масштабируемость решения окажется квадратичной.

Теперь попробуем решить ту же задачу, но с использованием оконных функций. На первом шаге воспользуемся функцией *ROW_NUMBER* для расчета порядковых номеров строк на основе столбца *col1*. Подробнее об этой функции мы будем говорить далее в этой книге. Сейчас же вам достаточно знать, что эта функция устанавливает для строк в секции уникальные числовые значения с единичным приращением с учетом заданной сортировки, начиная с единицы.

Таким образом, представленный ниже запрос добавит к столбцу *col1* столбец *rownum*, в котором будут представлены числа по возрастанию, расположенные в порядке увеличения значений в столбце *col1*:

```
SELECT col1, ROW_NUMBER() OVER(ORDER BY col1) AS rownum
FROM dbo.T1;

col1        rownum
----------- --------------------
2           1
3           2
11          3
12          4
13          5
27          6
33          7
34          8
35          9
42          10
```

Присмотритесь к двум представленным последовательностям чисел. Одна из них (в столбце *col1*) – с разрывами, вторая (в столбце *rownum*) – без. Теперь попробуйте догадаться, как взаимосвязаны между собой числа в этих последовательностях применительно к островам. Правильно, в рамках одного острова числа в обеих последовательностях получают одинаковые приращения. А значит, разница между значениями последовательностей будет постоянной. Рас-

смотрите для примера остров, начинающийся числом 11 и заканчивающийся числом 13. В столбце *col1* для этого острова содержатся числа 11, 12 и 13, а соответствующие им порядковые номера – 3, 4 и 5. Таким образом, разница между значениями в двух столбцах для каждой строки в рамках этого острова будет постоянной и равняться 8. При открытии следующего острова число в колонке *col1* увеличится больше чем на 1, тогда как порядковый номер в столбце *rownum* увеличится ровно на единицу. Следовательно, разница между значениями двух столбцов увеличится. Иными словами, как мы уже говорили, разница между значениями столбцов будет уникальна и постоянна в рамках каждого острова. Выполните следующий запрос и взгляните на результаты:

```
SELECT col1, col1 - ROW_NUMBER() OVER(ORDER BY col1) AS diff
FROM dbo.T1;

col1        diff
----------- --------------------
2           1
3           1
11          8
12          8
13          8
27          21
33          26
34          26
35          26
42          32
```

Как видите, вычисленная разница удовлетворяет обоим требованиям для нашего группового идентификатора – быть постоянным и уникальным в рамках каждого острова. А значит, мы можем остановиться на этом. Оставшиеся действия будут похожи на то, что мы делали в предыдущем примере, – мы снова сгруппируем наши данные по идентификатору и вернем минимальное и максимальное значения из столбца *col1* для каждой группы, как показано ниже:

```
WITH C AS
(
    SELECT col1,
        -- разница постоянна и уникальна для каждого острова
        col1 - ROW_NUMBER() OVER(ORDER BY col1) AS grp
    FROM dbo.T1
)
SELECT MIN(col1) AS startrange, MAX(col1) AS endrange
FROM C
GROUP BY grp;
```

Обратите внимание, насколько простым и элегантным стало решение. Но, конечно, это не значит, что не нужно снабжать его комментариями для тех, кто сталкивается с такой техникой впервые.

Кроме того, это решение будет гораздо более эффективным по сравнению с предыдущим, а при правильном индексировании его масштабируемость будет линейной. Операция присвоения порядковых значений строкам выполня-

ется практически молниеносно в сравнении с ранее описанным сценарием. Для этого однократно выполняется упорядоченное сканирование индекса по столбцу *col1*, а оператор увеличивает счетчик значений. При запуске теста с объемом таблицы в 10 млн записей запрос отработал за три секунды. Все остальные решения выполнялись намного дольше.

Мне кажется, рассмотренного примера использования оконных функций должно быть достаточно, чтобы заинтриговать вас и мотивировать на продолжение чтения книги. Теперь же вернемся к теоретическим нюансам реализации оконных функций. Но скучать вам не придется, ведь на протяжении этой книги мы рассмотрим еще немало примеров применения этих мощнейших функций на практике.

Элементы оконных функций

Спецификация оконной функции содержится внутри инструкции *OVER* и может включать в себя несколько элементов. К трем базовым элементам относятся *секционирование окна* (window partitioning), *упорядочивание* (ordering) и *определение границ окна* (framing). Не все оконные функции поддерживают все три элемента. При описании элементов я буду упоминать, какие именно функции их поддерживают.

Секционирование окна

Необязательный элемент *секционирования окна* (window partitioning) реализован при помощи инструкции *PARTITION BY* и поддерживается всеми без исключения оконными функциями. Секционирование служит для ограничения окна, в рамках которого будет применено вычисление, только теми строками набора данных, в которых значения *столбцов секционирования* (partitioning column) совпадают со значениями тех же столбцов в текущей строке. Например, если в вашей оконной функции используется секционирование вида *PARTITION BY custid*, и значение поля *custid* в текущей строке равно 1, секция окна, связанная с текущей строкой, будет включать в себя все строки результирующего набора запроса, в которых значение поля *custid* также равно 1. Если в поле *custid* в текущей строке будет находиться значение 2, в секцию для этой строки будут включены все записи с таким же значением поля *custid*.

В случае, если инструкция *PARTITION BY* не указана, окно для текущей строки ограничено не будет. Можно сказать, что в отсутствие явного указания секционирования будет применено секционирование по умолчанию, при котором весь набор данных запроса будет являться одной секцией.

Если это для вас не очевидно, позвольте мне пояснить на примере, что для разных оконных функций в одном запросе могут быть указаны разные спецификации в отношении секционирования. Рассмотрим запрос, приведенный ниже:

```
SELECT custid, orderid, val,
    RANK() OVER(ORDER BY val DESC) AS rnkall,
```

```
    RANK() OVER(PARTITION BY custid
        ORDER BY val DESC) AS rnkcust
FROM Sales.OrderValues;
```

Обратите внимание, что первая функция *RANK*, генерирующая атрибут *rnkall*, использует неявное секционирование, а вторая (*rnkcust*) – явное по полю *custid*. На рис. 1.5 графически показаны секции, определенные для трех результатов вычисления в запросе, одна секция для атрибута *rnkall* и две – для *rnkcust*.

```
custid  orderid  val      rnkall  rnkcust
------- -------- -------- ------- --------
1       11011    933.50   419     1
1       10692    878.00   440     2
1       10835    845.80   457     3
1       10643    814.50   469     4
1       10952    471.20   615     5
1       10702    330.00   686     6
2       10926    514.40   592     1
2       10625    479.75   608     2
2       10759    320.00   691     3
2       10308    88.80    797     4
…
```

Рис. 1.5 Секционирование окна

Стрелками на рис. 1.5 показаны секции, в рамках которых происходит вычисление тех или иных значений.

Упорядочивание окна

Инструкция, отвечающая за *упорядочивание окна* (window ordering), определяет сортировку в рамках секции для выполнения вычисления. Интересно, что применение этой инструкции незначительно отличается для разных категорий функций. Для ранжирующих функций упорядочивание выполняется интуитивно. Например, при использовании сортировки по убыванию функция *RANK* вернет на единицу большее значение по сравнению с количеством строк в секции, в которых значение сортировки больше текущего. Если сортировка установлена по возрастанию, функция вернет аналогичное значение в сравнении со строками, в которых значение сортировки меньше текущего. На рис. 1.6 показана процедура вычисления рангов из предыдущего примера – на этот раз включая интерпретацию сортирующего элемента.

На рис. 1.6 показано вычисление ранга для трех строк. Разумеется, таких вычислений в запросе производится гораздо больше – 1660, если быть точным. Всего в результирующем наборе данных 830 строк, и для каждой из них вычисляется два ранга. Любопытно отметить, что концептуально все эти окна сосуществуют одновременно.

При использовании агрегатных оконных функций упорядочивание окна имеет несколько иной смысл по сравнению с ранжирующими функциями. Вы могли подумать, что упорядочивание в этом случае определяет порядок при-

менения агрегации, но это не так. Вместо этого сортирующий элемент оказывает влияние на определение границ окна, о чем мы будем говорить далее. Иными словами, этот элемент позволяет определить строки, которыми будет ограничено окно.

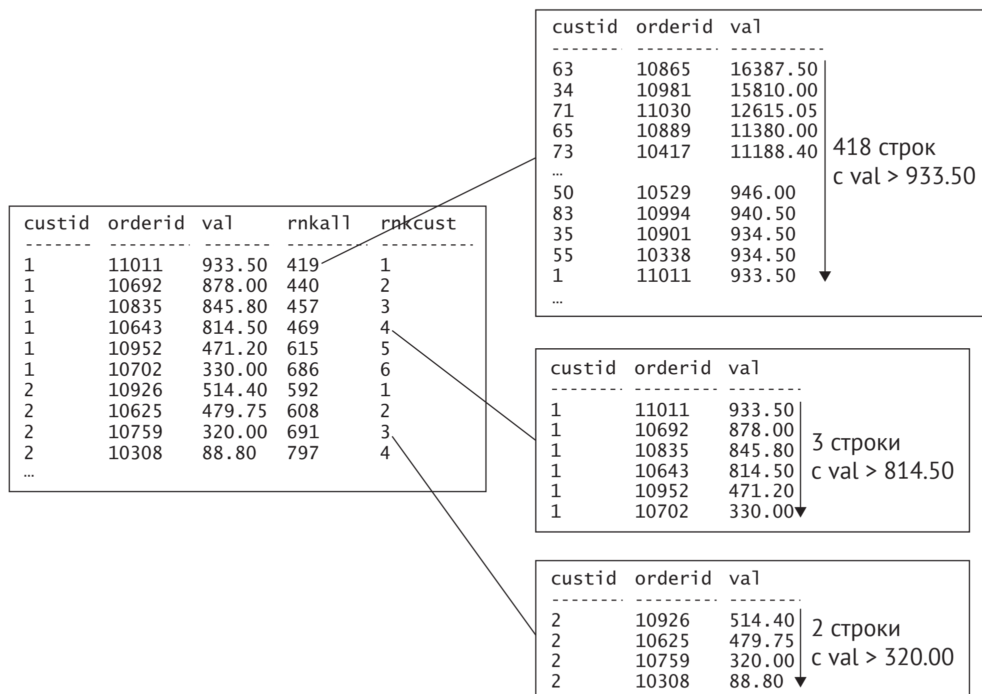

Рис. 1.6 Упорядочивание окна

Упорядочивание значений NULL

Если в сортируемых столбцах допустимо присутствие значений *NULL*, стандарт SQL полагается на конкретную реализацию при определении того, будут ли значения *NULL* располагаться перед или после значимых элементов. Microsoft при использовании сортировки по возрастанию предпочитает выводить сначала значения *NULL*. Стандарт SQL также позволяет указать для инструкции сортировки дополнительные опции *NULLS FIRST* или *NULLS LAST*. Это применимо как к упорядочиванию внутри окна, так и к сортировке при выводе результатов запроса. Предположим, вам необходимо обратиться к таблице *Sales.Orders* и извлечь идентификаторы и даты доставки заказов с сортировкой по дате доставки по возрастанию. При этом недоставленные заказы должны отображаться в конце списка. Признаком таких заказов является значение *NULL* в поле *shippeddate*. Согласно стандарту SQL, вы можете использовать инструкцию *NULLS LAST* следующим образом: ⇒

```
SELECT orderid, shippeddate
FROM Sales.Orders
ORDER BY shippeddate NULLS LAST;
```

К сожалению, SQL Server не поддерживает опцию *NULLS LAST*. Вместо этого он неявным образом использует опцию *NULLS FIRST*. Если вам нужно, чтобы значения *NULL* выводились в конце списка, вам придется реализовывать для этого собственное программное решение. Один из способов добиться этого – разместить еще один сортирующий элемент перед полем *shippeddate* на основе выражения *CASE*, возвращающий большее значение (например, 2) для значений *NULL* и меньшее (например, 1) – для не *NULL*. Пример такого запроса показан ниже:

```
SELECT orderid, shippeddate
FROM Sales.Orders
ORDER  BY
    CASE WHEN shippeddate IS NOT NULL THEN 1 ELSE 2 END,
    shippeddate;
```

Сокращенный вывод этого запроса будет следующим:

```
orderid     shippeddate
----------- -----------
10249       2017-07-10
10252       2017-07-11
10250       2017-07-12
10251       2017-07-15
10255       2017-07-15
...
11050       2019-05-05
11055       2019-05-05
11063       2019-05-06
11067       2019-05-06
11069       2019-05-06
11008       NULL
11019       NULL
11039       NULL
11040       NULL
...
```

К сожалению, этот способ ведет к снижению производительности запроса, поскольку не позволяет полагаться на сортировку по индексу, а вместо этого добавляет в план выполнения запроса явную сортировку. В главе 5 мы подробно поговорим об этом и о том, как избежать снижения производительности.

Определение границ окна

Ранее мы упоминали, что секционирование окна выполняет роль фильтра. *Определение границ окна* (window framing), по сути, также представляет собой фильтр, позволяющий еще больше ограничить используемые в рамках сек-

ции строки. Эта концепция применима к агрегатным оконным функциям, а также к трем следующим функциям смещения: *FIRST_VALUE*, *LAST_VALUE* и *NTH_VALUE*, последняя из которых не поддерживается SQL Server. Вы можете думать об этом элементе оконных функций как об определении двух граничных точек или разделителей в секции текущего окна на основании заданной сортировки, в рамках которых выполняется вычисление.

В стандарте спецификация определения границ окна включает в себя следующие опции: *ROWS*, *GROUPS* или *RANGE*. При помощи них происходит определение первой и последней строки рамок окна, а также задаются опции исключения. SQL Server в полном объеме поддерживает реализацию опции *ROWS*, частично поддерживает *RANGE* и не обладает собственной реализацией опции *GROUPS*.

Опция *ROWS* позволяет задать точки в границах окна в качестве смещения относительно текущей строки в соответствии с упорядочиванием окна. Опция *GROUPS* обладает тем же смыслом, что и *ROWS*, но смещение задается в количестве уникальных групп по отношению к текущей группе в соответствии с упорядочиванием окна. Опция *RANGE* является более динамической и позволяет задавать смещения с учетом разницы между значением сортировки в границах окна и значением в текущей строке. *Опции исключения* (frame-exclusion option) служат для указания действий с текущей строкой и ее соседями в случае совпадения значений. Это объяснение далеко от идеала, но в данный момент мне не хотелось бы вдаваться в подробности. Более детально об этом мы будем говорить далее. Сейчас же я хотел бы просто познакомить вас с общей концепцией и привести простой пример. Ниже показан запрос к представлению *EmpOrders*, в котором рассчитывается накопительный итог по количеству для каждого сотрудника по месяцам:

```
SELECT empid, ordermonth, qty,
    SUM(qty) OVER(PARTITION BY empid
        ORDER BY ordermonth
        ROWS BETWEEN UNBOUNDED PRECEDING
            AND CURRENT ROW) AS runqty
FROM Sales.EmpOrders;
```

Здесь мы применяем агрегатную функцию *SUM* к атрибуту *qty* с секционированием по полю *empid*, сортировкой по *ordermonth* и ограничением строк от первой в сегменте (UNBOUNDED PRECEDING) до текущей строки (CURRENT ROW). Проще говоря, в результате мы будем получать сумму значений по всем предшествующим строкам в обозначенных рамках, включая текущую строку. Этот запрос сгенерирует показанный ниже вывод:

```
empid  ordermonth qty  runqty
------ ---------- ---- -------
1      2017-07-01 121  121
1      2017-08-01 247  368
1      2017-09-01 255  623
1      2017-10-01 143  766
1      2017-11-01 318  1084
...
```

```
2      2017-07-01 50   50
2      2017-08-01 94   144
2      2017-09-01 137  281
2      2017-10-01 248  529
2      2017-11-01 237  766
...
```

Обратите внимание, что спецификацию оконных функций в запросе можно читать, словно обычный текст на английском. В главе 2 мы будем подробнее говорить об определении границ окна.

Элементы запросов с поддержкой оконных функций

Оконные функции поддерживаются не всеми элементами запросов. Фактически эти функции можно использовать только в инструкциях *SELECT* и *ORDER BY*. Чтобы понять причину этого ограничения, вам необходимо познакомиться с концепцией, получившей название логическая обработка запроса. После этого мы пройдемся по инструкциям, поддерживающим оконные функции, а в заключении обсудим некоторые альтернативные пути, позволяющие обойти указанные ограничения.

Логическая обработка запроса

Логическая обработка запроса (logical query processing) представляет собой последовательность выполнения запроса *SELECT* с учетом логической структуры языка. Описание процесса состоит из последовательности шагов или фаз, начиная от обработки исходных таблиц и заканчивая формированием результирующего набора данных запроса. Обратите внимание, что под логической обработкой запроса подразумевается именно концептуальная последовательность выполнения запроса, которая не обязательно должна совпадать с тем, как SQL Server обрабатывает запрос физически. В процессе оптимизации плана выполнения SQL Server может переставлять местами части запроса, выполняя их в разной последовательности, да и вообще может творить все что угодно. Разумеется, до тех пор, пока запрос будет выдавать такие же результаты, как в случае с логической обработкой декларативного запроса.

На каждом шаге логической обработки запроса производится операция с одной или несколькими виртуальными таблицами (наборами строк), поступающими на вход, и на выходе также возвращается виртуальная таблица. Таким образом, виртуальная таблица, получающаяся в результате выполнения одного шага, поступает на вход другого.

На рис. 1.7 представлена диаграмма, иллюстрирующая логическую обработку запроса в SQL Server.

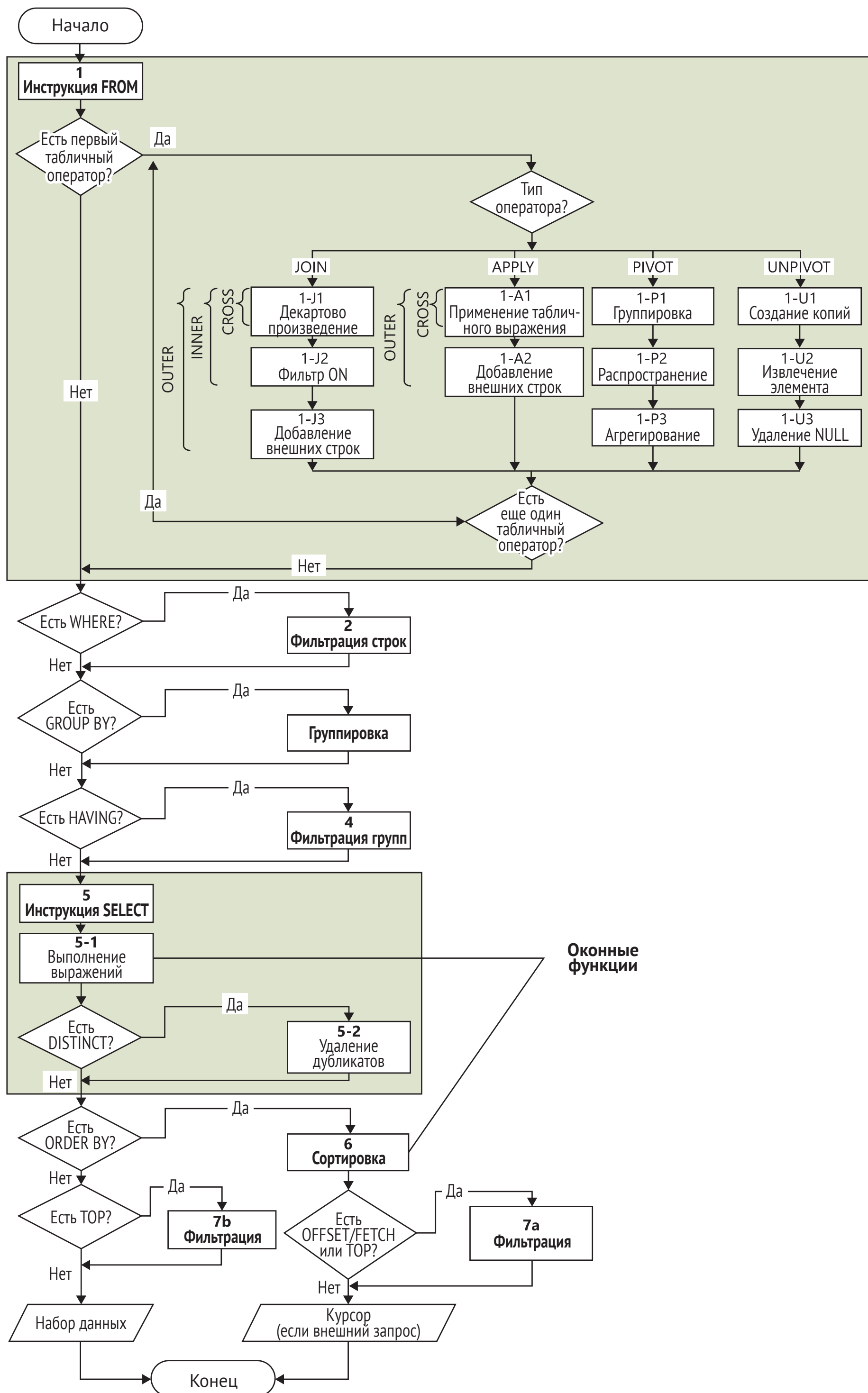

Рис. 1.7 Логическая обработка запроса

Обратите внимание, что инструкция *SELECT*, стоящая обычно в самом начале запроса, на диаграмме логической обработки запроса появляется едва ли не последней – непосредственно перед инструкцией *ORDER BY*.

Говорить о диаграмме логической обработки запроса можно бесконечно, но это тема для другой книги. Здесь нам важно понять, в каком порядке выполняются инструкции. Ниже в упрощенном виде приведена последовательность выполнения запроса (фазы, на которых применимы оконные функции, выделены жирным шрифтом):

1. *FROM*
2. *WHERE*
3. *GROUP BY*
4. *HAVING*
5. *SELECT*
   5-1. Выполнение выражений
   5-2. Удаление дубликатов
6. ***ORDER BY***
7. *OFFSET-FETCH/TOP*

Понимание процесса логической обработки запроса позволит вам разобраться в том, почему оконные функции могут применяться только на определенных этапах запроса.

Инструкции с поддержкой оконных функций

Как видно на рис. 1.7, напрямую оконные функции поддерживают только инструкции *SELECT* и *ORDER BY*. Причина этого ограничения состоит в том, чтобы избежать двусмысленности при работе с (почти) окончательным набором данных в качестве источника для произведения оконных вычислений. Если бы оконные функции могли появляться до инструкции *SELECT*, их входные окна могли бы отличаться от итоговых, что привело бы к большим затруднениям при получении правильных результатов. Я продемонстрирую эту двусмысленность на примере. Для начала запустите следующий код для создания таблицы *T1* и заполнения ее данными:

```
SET NOCOUNT ON;
USE TSQLV5;
DROP TABLE IF EXISTS dbo.T1;
GO
CREATE TABLE dbo.T1
(
col1 VARCHAR(10) NOT NULL
CONSTRAINT PK_T1 PRIMARY KEY
);

INSERT INTO dbo.T1(col1)
VALUES(‘A’),(‘B’),(‘C’),(‘D’),(‘E’),(‘F’);
```

Представьте, что оконные функции могут появляться в запросе раньше выполнения инструкции *SELECT* – допустим, на стадии *WHERE*. Посмотрите на

следующий запрос и попытайтесь угадать, какими значениями в результате будет заполнен столбец *col1*:

```
SELECT col1
FROM dbo.T1
WHERE col1 > 'B'
    AND ROW_NUMBER() OVER(ORDER BY col1) <= 3;
```

Прежде чем ответить, что в результирующем наборе должны остаться буквы C, D и E, примите во внимание принцип одновременного выполнения в SQL. Согласно нему, все выражения, появляющиеся на одной и той же логической фазе запроса, концептуально выполняются одновременно. Это означает, что строгий порядок выполнения выражений не должен иметь значения. Учитывая это, приведенный ниже запрос должен быть идентичен предыдущему:

```
SELECT col1
FROM dbo.T1
WHERE ROW_NUMBER() OVER(ORDER BY col1) <= 3
    AND col1 > 'B';
```

А какой ответ будет правильным теперь? C, D и E или только C?

Именно о такой двусмысленности я и говорил. Ограничение употребления оконных функций инструкциями *SELECT* и *ORDER BY* позволяет полностью устранить подобные неоднозначности.

Глядя на рис. 1.7, вы могли заметить, что фаза инструкции *SELECT* включает в себя шаг *5-1 (Выполнение выражений)*, на котором и происходит запуск оконных функций, и он выполняется раньше шага *5-2 (Удаление дубликатов)*. Если вам интересно, почему это так важно, я продемонстрирую вам на примере.

Ниже показан запрос, извлекающий все содержимое столбцов *empid* и *country* из таблицы *Employees*:

```
SELECT empid, country
FROM HR.Employees;

empid       country
----------- ---------------
1           USA
2           USA
3           USA
4           USA
5           UK
6           UK
7           UK
8           USA
9           UK
```

Теперь посмотрите на следующий запрос и попытайтесь угадать, каким будет его вывод:

```
SELECT DISTINCT country, ROW_NUMBER() OVER(ORDER BY country) AS rownum
FROM HR.Employees;
```

Кто-то может предположить, что результат должен быть следующим:

```
country          rownum
-------------- -------------------
UK             1
USA            2
```

Но на самом деле результат будет таким:

```
country  rownum
-------- -------
UK       1
UK       2
UK       3
UK       4
USA      5
USA      6
USA      7
USA      8
USA      9
```

Вы должны понимать, что функция *ROW_NUMBER* вызывается на шаге 5-1, где вычисляются все выражения *SELECT*, еще до удаления дубликатов на шаге 5-2. В результате функция *ROW_NUMBER* присваивает строкам девять уникальных числовых значений, чем полностью лишает инструкцию *DISTINCT* работы, поскольку дубликатов в таблице просто не остается.

Если принять это во внимание, а также учесть концепцию логической обработки запроса, можно выработать решение и для сценария, в котором должны остаться всего две строки. Например, мы могли бы создать табличное выражение на основе запроса, возвращающего список уникальных стран, а оконную функцию запустить уже после удаления дубликатов, как показано ниже:

```
WITH EmpCountries AS
(
    SELECT DISTINCT country FROM HR.Employees
)
SELECT country, ROW_NUMBER() OVER(ORDER BY country) AS rownum
FROM EmpCountries;

country  rownum
-------- -------
UK       1
USA      2
```

Быть может, вы знаете более простые способы решить эту задачу?

Тот факт, что оконные функции выполняются на фазах *SELECT* или *ORDER BY*, означает, что окно, определенное для вычисления – до дальнейших ограничений, – является промежуточной формой строк запроса, прошедших все предшествующие фазы. А значит, оконные функции применяются после выполнения инструкции *FROM* со всеми ее операторами вроде объединения таблиц и после наложения фильтров инструкцией *WHERE*, группировки данных и фильтрации групп. Рассмотрим следующий пример:

```
SELECT O.empid,
    SUM(OD.qty) AS qty,
    RANK() OVER(ORDER BY SUM(OD.qty) DESC) AS rnk
FROM Sales.Orders AS O
    INNER JOIN Sales.OrderDetails AS OD
        ON O.orderid = OD.orderid
WHERE O.orderdate >= '20180101'
    AND O.orderdate < '20190101'
GROUP BY O.empid;

empid  qty   rnk
------ ----- ----
4      5273  1
3      4436  2
1      3877  3
8      2843  4
2      2604  5
7      2292  6
6      1738  7
5      1471  8
9      955   9
```

Сначала выполняется инструкция *FROM* с объединением таблиц. Затем данные в результирующем наборе фильтруются по датам, чтобы остались только заказы за 2018 год. После этого выполняется группировка данных по идентификатору сотрудника (*empid*). И только в завершение вычисляются выражения из инструкции *SELECT*, включая функцию *RANK*, рассчитывающую ранги на основе сортировки по общему количеству заказанных товаров в порядке убывания. Если бы в инструкции *SELECT* присутствовали другие оконные функции, они бы все использовали в качестве отправной точки для своих вычислений тот же результирующий набор. Помните о том, что мы говорили применительно к альтернативам оконным функциям (например, подзапросам), – все они начинают вычисление с нуля, а значит, вам было бы необходимо повторять всю логику, реализованную во внешнем запросе, для каждого подзапроса, что неизбежно привело бы к увеличению сложности итогового запроса.

Теперь вернемся к моему вопросу о том, существует ли более простой способ присвоить присвоить номера строкам с уникальными странами по сравнению с использованием обобщенного табличного выражения. Вот более простое решение:

```
SELECT country, ROW_NUMBER() OVER(ORDER BY country) AS rownum
FROM HR.Employees
GROUP BY country;
```

У нас в базе есть девять сотрудников из двух стран, и на этапе группировки мы выделяем эти две группы. Выражения в инструкции *SELECT*, включая функцию *ROW_NUMBER*, вычисляются применительно к этим группам, что ожидаемо приводит к выводу двух следующих строк в результирующем наборе данных:

```
country  rownum
-------- -------
UK       1
USA      2
```

В обход ограничений

Я уже говорил о причинах того, почему в процессе логической обработки запроса оконные функции не могут выполняться на этапах, предшествующих инструкции *SELECT*. А что, если вам необходимо выполнить фильтрацию или группировку на основе вычислений оконных функций? В этом случае вы можете использовать табличное выражение или *производную таблицу* (derived table). Напишите запрос с табличной функцией с присвоенным ей *псевдонимом* (alias) в составе инструкции *SELECT*, объявите табличное выражение на основе этого запроса и во внешнем запросе обращайтесь к вычислению вашей оконной функции по псевдониму там, где вам необходимо.

Ниже приведен пример запроса с фильтрацией по результату вычисления оконной функции с использованием обобщенного табличного выражения:

```
WITH C AS
(
    SELECT orderid, orderdate, val,
        RANK() OVER(ORDER BY val DESC) AS rnk
    FROM Sales.OrderValues
)

SELECT *
FROM C
WHERE rnk <= 5;

orderid  orderdate  val       rnk
-------- ---------- --------- ----
10865    2019-02-02 16387.50  1
10981    2019-03-27 15810.00  2
11030    2019-04-17 12615.05  3
10889    2019-02-16 11380.00  4
10417    2018-01-16 11188.40  5
```

В запросах на изменение данных оконные функции неприменимы по причине отсутствия в них инструкций *SELECT* и *ORDER BY*. Но бывают случаи, когда в таких запросах необходимо обращаться к результатам вычисления оконных функций. С этой целью также могут использоваться табличные выражения, поскольку язык T-SQL позволяет применять их совместно с запросами на изменение данных. Ниже я покажу пример использования этой техники применительно к обновлению данных с помощью инструкции *UPDATE*. Для начала запустите следующий код для создания таблицы *T1* со столбцами *col1* и *col2* и заполнения ее исходными данными:

```
SET NOCOUNT ON;
USE TSQLV5;
```

```
DROP TABLE IF EXISTS dbo.T1;
GO

CREATE TABLE dbo.T1
(
    col1 INT NULL,
    col2 VARCHAR(10) NOT NULL
);

INSERT INTO dbo.T1(col2)
    VALUES('C'),('A'),('B'),('A'),('C'),('B');
```

Как видите, мы прописали явные значения в столбце *col2*, а в столбце *col1* будут установлены значения по умолчанию *NULL*.

Допустим, на примере этой таблицы мы хотим показать проблемы с качеством данных. В ней не присутствуют ключевые поля, а значит, невозможно уникально идентифицировать записи. Ваша цель – заполнить поле *col1* уникальными значениями. В идеале вам бы хотелось выполнить какой-то такой запрос:

```
UPDATE dbo.T1
    SET col1 = ROW_NUMBER() OVER(ORDER BY col2);
```

Но это невозможно. Обходной путь состоит в написании запроса к таблице *T1*, возвращающего поле *col1* и выражение, основанное на оконной функции *ROW_NUMBER* (назовите его *rownum*), определении табличного выражения на базе этого запроса и запуске внешнего запроса *UPDATE* по отношению к табличному выражению с присвоением значения вычисления *rownum* полю *col1*, как показано ниже:

```
WITH C AS
(
    SELECT col1, col2,
        ROW_NUMBER() OVER(ORDER BY col2) AS rownum
    FROM dbo.T1
)
UPDATE C
    SET col1 = rownum;
```

Теперь обратитесь к таблице *T1* – в поле *col1* будут находиться уникальные значения:

```
SELECT col1, col2
FROM dbo.T1;

col1  col2
----- -----
5     C
1     A
3     B
2     A
6     C
4     B
```

Возможности для использования дополнительных фильтров

Ранее я показал вам обходной путь, позволяющий неявно использовать в языке T-SQL оконные функции в инструкциях, напрямую их не поддерживающих. Этот путь был основан на использовании табличных выражений или производных таблиц. Все это хорошо, но использование табличных выражений добавляет дополнительный слой к запросу, что ведет к его усложнению. Рассмотренные нами примеры были довольно простыми, но в реальной жизни вы будете иметь дело с гораздо более сложными запросами. Есть ли более простое решение нашей задачи, не подразумевающее создания дополнительных слоев?

В SQL Server на данный момент такого решения не существует. Но мы можем посмотреть, как справляются с этим другие системы. К примеру, в СУБД Teradata есть возможность создания фильтрующей инструкции *QUALIFY*, которая концептуально выполняется позже *SELECT*. Таким образом, в этой инструкции вы можете напрямую обращаться к результатам вычислений оконных функций, как показано ниже:

```
SELECT orderid, orderdate, val
FROM Sales.OrderValues
QUALIFY RANK() OVER(ORDER BY val DESC) <= 5;
```

Более того, вы можете обращаться и к псевдонимам столбцов, определенных в инструкции *SELECT*:

```
SELECT orderid, orderdate, val,
RANK() OVER(ORDER BY val DESC) AS rnk
FROM Sales.OrderValues
QUALIFY rnk <= 5;
```

Инструкция *QUALIFY* не определена в стандарте SQL, а присутствует только в СУБД Teradata. В то же время такое решение является довольно логичным и обоснованным, и было бы здорово, если бы его поддержка появилась в стандарте SQL и СУБД SQL Server.

Повторное использование определений окна

Предположим, вам необходимо выполнить несколько оконных функций в рамках одного запроса, при этом для некоторых или всех из них требуется использовать одну и ту же спецификацию окна. Если повторить эту спецификацию для каждой функции, запрос неимоверно разрастется и станет абсолютно нечитаемым, как показано ниже:

```
SELECT empid, ordermonth, qty,
    SUM(qty) OVER (PARTITION BY empid
        ORDER BY ordermonth
        ROWS BETWEEN UNBOUNDED PRECEDING
            AND CURRENT ROW) AS runsumqty,
    AVG(qty) OVER (PARTITION BY empid
        ORDER BY ordermonth
        ROWS BETWEEN UNBOUNDED PRECEDING
            AND CURRENT ROW) AS runavgqty,
    MIN(qty) OVER (PARTITION BY empid
        ORDER BY ordermonth
        ROWS BETWEEN UNBOUNDED PRECEDING
            AND CURRENT ROW) AS runminqty,
    MAX(qty) OVER (PARTITION BY empid
        ORDER BY ordermonth
        ROWS BETWEEN UNBOUNDED PRECEDING
            AND CURRENT ROW) AS runmaxqty
FROM Sales.EmpOrders;
```

В стандарте SQL есть решение этой проблемы в виде особой инструкции *WINDOW*, позволяющей задать имя для спецификации окна, к которому в дальнейшем можно обращаться как в самих оконных функциях, так и при определении других окон. Концептуально эта инструкция выполняется после *HAVING* и до *SELECT*.

SQL Server пока не поддерживает инструкцию *WINDOW*. Но при использовании стандарта SQL вы можете переписать предыдущий запрос с применением инструкции *WINDOW* следующим образом:

```
SELECT empid, ordermonth, qty,
    SUM(qty) OVER W1 AS runsumqty,
    AVG(qty) OVER W1 AS runavgqty,
    MIN(qty) OVER W1 AS runminqty,
    MAX(qty) OVER W1 AS runmaxqty
FROM Sales.EmpOrders
WINDOW W1 AS ( PARTITION BY empid
    ORDER BY ordermonth
    ROWS BETWEEN UNBOUNDED PRECEDING
        AND CURRENT ROW );
```

Как видите, совсем другое дело. В данном случае мы при помощи инструкции *WINDOW* задали спецификации окна, включающей секционирование, упорядочивание и определение границ окна, имя *W1*. После этого для всех оконных функций в запросе мы использовали эту именованную спецификацию. Инструкция *WINDOW* может быть очень полезной. Как я упоминал ранее, при помощи нее вы можете объявлять не всю спецификацию окна, а лишь ее часть, а дополнения указывать явно при определении окна. В описании стандарта SQL инструкции *WINDOW* отводится целых десять страниц, и пересказывать все подробности ее использования – не самое большое удовольствие.

Сразу несколько платформ поддерживают инструкцию *WINDOW*. В их число входят PostgreSQL, MariaDB, MySQL и SQLite. Было бы здорово, если бы и в SQL

Server в будущем появилась поддержка этой полезной инструкции. Сейчас, когда оконные функции получают большое распространение, а разработчики пишут все более сложные спецификации окон, в этом действительно есть необходимость.

Заключение

В этой главе мы познакомились с концепцией работы с окнами в SQL и усвоили основные мотивы для их использования. Также мы решили несложную задачку на определение островов в данных при помощи оконных функций. Кроме того, мы поверхностно рассмотрели основные элементы определения спецификации окна, включая секционирование, упорядочивание и определение границ окна. Наконец, мы посмотрели, как в стандарте SQL можно повторно использовать именованные спецификации окон или их части. В следующей главе мы продолжим изучать оконные функции и углубимся в детали.