

































































































































































































































































































































































































































































































## Вставка в толстой транзакции

Если пакетная вставка не может быть использована, программисту остаётся построчная передача данных от источника к приёмнику. Часто встречающаяся ошибка в этом случае — отсутствие явных транзакций.

Как уже было сказано, вставка строки в таблицу является неявной транзакцией, поэтому, например, перекачивая миллион даже из текстового файла строк в таблицу реляционной СУБД вы неявно проведёте миллион транзакций, что резко снизит производительность вашей программы.

Выходом из ситуации является объединение вставок в пакет под одной транзакцией. Размер пакета следует подбирать экспериментально, он может достигать и сотен тысяч записей. Предварительно, данные кешируются в клиентском наборе данных (DataSet).

Пример загрузки в среде Delphi/Lazarus с использованием библиотеки компонентов доступа к различным СУБД UniDAC.

```
procedure BulkLoad(Source, Target: TUniTable);

const
  BatchSize = 10000; // размер пакета в одной транзакции

procedure CommitBatch;
var
  Tnx: TUniTransaction;
begin
  Tnx := TUniTransaction.Create(nil);
  try
    try
      Tnx.AddConnection(Target.Connection);
      Target.UpdateTransaction := Tnx;
      Tnx.StartTransaction;
      Target.ApplyUpdates; // вставка порции в таблицу СУБД
      Tnx.Commit;
      Target.CommitUpdates; // очистка кеша
    except
      Tnx.Rollback;
      raise;
    end;
  finally
    Target.UpdateTransaction := nil;
```

```

        FreeAndNil(Tnx);
    end;
end;

var
    i, CurrRecNo: integer;
begin
    Source.First;
    CurrRecNo := 0;
    while not Source.EOF do
    begin
        Inc(CurrRecNo);
        Target.Append;
        for i := 0 to Source.Fields.Count - 1 do
            Target.Fields[i].Value := Source.Fields[i].Value;
        if CurrRecNo mod BatchSize = 0 then
            CommitBatch;
        Source.Next;
        end;
        if Target.UpdatesPending then
            CommitBatch;
        end;
end;

```

По сравнению с одинаковыми значениями записей, толстая транзакция обеспечивает даже не в разы, а на порядки (100-1000 раз) большее быстродействие. Оборачивать стороны толстой транзакции является потенциальная блокировка работы других пользователей. Поэтому для интенсивно используемых транзакционных БД размер пакета не должен превышать 10-100 записей, обеспечивая быстрое выполнение пакета.

Все вышесказанное также касается операций обновления и удаления данных.

## ***РСУБД и неполно структурированные данные***

В главе «Неполно структурированные данные и высокая нагрузка» был описан метод проектирования, позволяющей средствами реляционной СУБД получить функционал, отсутствие которого может быть причинами выбора других решений из области NoSQL.

К концу 1990х годов XML стал фактическим стандартом для систем электронного обмена данными<sup>21</sup> (СОД), шлюзов между информационными системами разных уровней. Возможность работы с XML была введена основными производителями прежде всего для поддержки интеграции и обработки XML-документов средствами РСУБД. Например, SQL Server способен даже реализовать полноценную веб-службу исключительно средствами языка Transact SQL.

Современные РСУБД обладают широкими возможностями хранения и обработки неполно структурированных данных (НСД), прежде всего в виде XML. Программист получает возможность не только совмещать сильные стороны обоих подходов, но и обрабатывать НСД, используя множественный подход и соединения с физическими таблицами.

## Поддержка XML

Рассмотрим на примерах возможности обработки XML, предоставляемые СУБД PostgreSQL (таблица 9).

Создадим две таблицы (заказов (orders) и продуктов (products), но, в отличие от обычного реляционного подхода, будем хранить часть атрибутов в виде XML. Для продуктов в качестве НСД выступает спецификация, которая может содержать разнообразную информацию, а для заказов — его переменный состав.

```
CREATE TABLE products (  
  id_prod integer NOT NULL,  
  code    varchar(10) NOT NULL,  
  name    varchar(50) NOT NULL,  
  spec    xml NOT NULL,  
  CONSTRAINT pk_products PRIMARY KEY (id_prod)  
);  
  
CREATE TABLE orders  
(  
  id_order integer NOT NULL,  
  created  date NOT NULL,  
  doc_data xml NOT NULL,  
  CONSTRAINT pk_orders PRIMARY KEY (id_order)
```

<sup>21</sup> В англоязычной среде используется аббревиатура EDI — Electronic Data Interchange

```
);
```

Таблицы получились несвязными явным образом, поскольку требуемая колонка внешнего ключа отсутствует. Первая плата за гибкость — СУБД не гарантирует целостность, заказы могут содержать ссылки на несуществующие продукты. Но при необходимости можно реализовать соответствующую поддержку триггером.

Заполним таблицы тестовыми данными. Как можно видеть, сохраняемый XML не имеет строгой схемы и потому относится к НСД.

```
/* Инициализация таблицы продуктов */
INSERT INTO products (id_prod, code, name, spec)
SELECT 1, 'P01', 'Мука пшеничная в/с',
XMLPARSE(DOCUMENT convert_from(
  '<packs>
  <pack>
    <weight>1</weight>
    <unit>кг</unit>
  </pack>
  <pack>
    <weight></weight>
    <unit>кг</unit>
  </pack>
</packs>', 'utf-8')::xml)
UNION ALL
SELECT 2, 'P02', 'Дрожжи',
XMLPARSE(DOCUMENT convert_from(
  '<packs>
  <pack>
    <weight>100</weight>
    <unit>г</unit>
  </pack>
  <expired>10</expired>
</packs>', 'utf-8')::xml)
UNION ALL
SELECT 3, 'P03', 'Сахар',
XMLPARSE(DOCUMENT convert_from(
  '<packs>
  <pack>
    <weight>10</weight>
    <unit>кг</unit>
  </pack>
  <color>белый</color>
</packs>', 'utf-8')::xml);
```

```

/* Инициализация таблицы заказов */
INSERT INTO orders (id_order, created, doc_data)
SELECT 1, date '2014-02-20',
XMLPARSE(DOCUMENT convert_from(
  '<items>
    <product>
      <id>1</id>
      <quantity>50</quantity>
      <units>кг</units>
      <price>45</price>
    </product>
    <product>
      <id>2</id>
      <quantity>300</quantity>
      <units>г</units>
      <price>80</price>
    </product>
  </items>', 'utf-8')::xml)
UNION ALL
SELECT 2, date '2014-02-21',
XMLPARSE(DOCUMENT convert_from(
  '<items>
    <product>
      <id>3</id>
      <quantity>100</quantity>
      <units>кг</units>
      <price>70</price>
    </product>
  </items>', 'utf-8')::xml)
UNION ALL
SELECT 3, date '2014-02-22',
XMLPARSE(DOCUMENT convert_from(
  '<items>
    <product>
      <id>1</id>
      <quantity>80</quantity>
      <units>кг</units>
      <price>50</price>
    </product>
  </items>', 'utf-8')::xml);

```

Рассмотрим подробнее, какие возможности даёт использование XML в запросах. PostgreSQL использует стандартизованный язык XPath, уже упоминавшийся в главе «Иерархическая модель». Выражение XPath в

PostgreSQL возвращает значение в виде одномерного массива из XML-элементов.

```
SELECT name,
       xpath('//pack/weight/text()', spec) AS pack_weight
FROM products
```

Результат:

| name                  | pack_weight |
|-----------------------|-------------|
| character varying(50) | xml[]       |
| -----                 | -----       |
| Мука пшеничная в/с    | {1,50}      |
| Дрожжи                | {100}       |
| Сахар                 | {10}        |

Для преобразования одномерного массива в реляционный вид используется встроенная функция `unnest`. Например следующий оператор вернёт таблицу из одной колонки со значениями массива. В терминах линейной алгебры, проводится операция транспонирования вектора-строки в столбец.

```
SELECT unnest('СИНЙ|КРАУ|ЗЕЛЁНЙ' AS ARRAY<TEXT>) AS color
```

Результат:

| color   |
|---------|
| text    |
| -----   |
| Синий   |
| Красный |
| Зелёный |

Другая полезная функция-предикат — `xpath_exists`, возвращающая истину в случае нахождения заданного пути в XML-документе или ложь в противном случае. Поскольку документы не ограничены схемой, использование проверки наличия соответствующих элементов будет частой необходимостью.

```
SELECT name, xpath_exists('//color', spec)
FROM products
```

Результат:

| name                  | xpath_exists |
|-----------------------|--------------|
| character varying(50) | boolean      |
| -----                 | -----        |

|                    |   |
|--------------------|---|
| Мука пшеничная в/с | f |
| Дрожжи             | f |
| Сахар              | t |

Теперь попробуем применить на практике полученное представление о работе XML-функций.

Запрос №1. Вывести все продукты, у которых специфицирован максимальный срок реализации.

```
SELECT *
FROM products
WHERE xpath_exists('//expired', spec)
```

Запрос №2. То же, но с выводом числового значения срока (в сутках). Обратите внимание, что требуются несколько операций: конвертация значений XML-массива в текстовый и целочисленный типы, преобразование в скаляр.

```
SELECT id_prod,
       unnest(xpath('//expired[1]/text()', spec)::text[]::int[])
FROM products
WHERE xpath_exists('//expired', spec)
```

Ещё одна плата за гибкость: из-за отсутствия сортировки данных, мы не можем гарантировать что значение срока истечения годности в спецификации будет единственным. Поэтому средствами XPath явно выбираем только первый подходящий элемент.

Другой вариант — выбрать первый элемент массива уже от полученного результата. При большом размере массива он может быть менее производительным, чем отсечение элементов на уровне XPath-запроса.

```
SELECT id_prod,
       (xpath('//expired/text()', spec))[1]::text::int
FROM products
WHERE xpath_exists('//expired', spec)
```

Запрос №3. Выбрать все заказы, включающие продукты, у которых в спецификации не указаны сведения о сроке истечения годности. Здесь запрос немного посложнее и состоит из двух выборок, соединяемых по ключу.

```
SELECT o.id_order, p.id_prod, p.name
FROM
```

```

(SELECT id_prod, name
 FROM products
 WHERE NOT xpath_exists('//expired', spec)
 ) p
INNER JOIN
(SELECT id_order,
      unnest(xpath('//items/product/id/text()',
                  doc_data)::text[]::int[]) AS id_prod
 FROM orders
 ) o
ON p.id_prod = o.id_prod

```

Другой вариант использует вложенный подзапрос.

```

SELECT o.id_order, p.id_prod, p.name
FROM orders o
      INNER JOIN products p
      ON p.id_prod = ANY(
      xpath('//items/product/id/text()',
            o.doc_data)::text[]::int[])
WHERE NOT xpath_exists('//expired', p.spec)

```

Как мы помним, мы не знаем, какие значения ключа в XML-документе и ключ продукта не гарантировано, существующие значения будут просто пропущены. Для их отображения надо использовать внешнее соединение и произвести корректировку ассоциированных данных.

Запрос №4. Рассчитать общую сумму покупки по каждому заказу.

```

SELECT id_order, sum(price * qty)
FROM
  (SELECT id_order,
        unnest(xpath('//items/product/price/text()',
                    doc_data)::text[]::int[]) AS price,
        unnest(xpath('//items/product/quantity/text()',
                    doc_data)::text[]::int[]) AS qty
  FROM orders) o
GROUP BY id_order

```

Очередная плата за гибкость: в отсутствии схемы остаётся лишь надеяться, что значение элемента <price> может быть преобразовано в целочисленное. В противном случае выполнение запроса прервётся по ошибке.

## Индексация поиска

За рамками примеров остался вопрос индексации НСД. Пока таблица содержит небольшое количество документов, выполнение запросов, осуществляющих последовательное сканирование таблицы и проверку каждого документа.

Таблица заказов из примера содержит всего три строки. Если заполнить её данными, объёмом в несколько сотен тысяч строк, то время отклика при выполнении тех же запросов будет исчисляться уже не миллисекундами, а единицами и десятками секунд.

В рамках примера ограничимся программированием небольшого SQL-скрипта, заполняющего таблицы продуктов и заказов. В то же время пример заполнения таблицы тестовыми данными одним единственным оператором SELECT показывает, насколько язык развит и высок по уровню. Напишите тот же код на Яве или C# и сравните по числу операторов и строк.

```
TRUNCATE TABLE products;
INSERT INTO products (id, created, name, doc_data)
SELECT id,
       'P' || (id)::text AS code,
       md5(random()::text) AS name,
       xmlparse(document (
         '<packs><pack><weight>' ||
         trunc(random() * 100)::text ||
         '</weight></pack></packs>')::xml)
FROM (SELECT *
      FROM generate_series(1, 1000) AS id
      ) AS seq;

TRUNCATE TABLE orders;
INSERT INTO orders (id_order, created, doc_data)
SELECT id,
       date '2014-01-01' + trunc(random() * 100)::int,
       xmlparse(document (
         '<items>' ||
         '<product><id>' || trunc(random() * 500 + 1)::text ||
         '</id>' ||
         '<quantity>' || trunc(random() * 700)::text ||
         '</quantity>' ||
```

```

    '<price>' || trunc(random() * 100)::text || '</price>'
||
    '</product>' ||
    '<product><id>' || trunc(random() * 500 + 501)::text
||
    '</id>' ||
    '<quantity>' || trunc(random() * 300)::text ||
    '</quantity>' ||
    '<price>' || trunc(random() * 200)::text || '</price>'
||
    '</product>' ||
    '</items>')::xml)
FROM (SELECT *
      FROM generate_series(1, 100000) AS id
      ) AS seq;

```

Если выполнить на заполненной базе относительно простой запрос, то можно констатировать увеличение времени отклика до нескольких секунд.

```

SELECT *
FROM orders
WHERE (xpath('//*[items/product/id/text()]',
            doc_data)[1]::text)::integer = 123

```

Добавив перед запросом команду EXPLAIN, можно увидеть, что план запроса состоит из последовательного сканирования таблицы orders и фильтрации по XPath-выражению, выполняемому над документом.

```

Seq Scan on orders (cost=0.00..4500.00 rows=500 width=40)
  Filter: (((xpath('//*[items/product/id/text()]'::text,
doc_data, '{}')::text[1]))[1])::text)::integer = 123)

```

В общем случае, ситуацию можно поправить построением XPath-индексов по наиболее часто используемым выражениям. Для приведённого XPath-запроса индекс будет аналогом индекса по внешнему ключу.

```

CREATE INDEX ix1_orders_id_prod_xml
ON orders USING btree (
  ((xpath('//*[items/product/id/text()]',
          doc_data))[1]::text)::int)
)

```

Перезапустив запрос можно увидеть, что время выполнения сократилось на порядки. В моём частном случае величины равнялись 1868 и 72 миллисекунды, соответственно, что составляет оптимизацию по времени в 26 раз. Абсолютные значения будут зависеть от мощности

компьютера, но относительная разница в 20-30 раз воспроизводится независимо от аппаратной конфигурации.

План выполнения подтверждает использование созданного индекса.

```
Bitmap Heap Scan on orders (cost=12.14..1259.81 rows=500
width=40)
  Recheck Cond: (((xpath('//items/product/id/text()'::text,
doc_data, '{}')::text[1])[1])::text)::integer = 123)
  -> Bitmap Index Scan on ix1_orders_id_prod_xml
(cost=0.00..12.02 rows=500 width=0)
    Index Cond:
      (((xpath('//items/product/id/text()'::text, doc_data,
'{}')::text[1])[1])::text)::integer = 123)
```

Решена ли проблема?

Не решена, но локализована. Использовать данный индекс для других типов выражений невозможно, тогда как введение колонки, хранящей НДС в виде XML, было сделано в соответствии с пожеланием хранить любую информацию с заранее неизвестными атрибутами. Если добавится новая информация, пользователь столкнется с дилеммой: создавать новый индекс или оставить пользователя в ожидании перед монитором.

Следует понимать, что выбор между индексацией, вызывающей увеличение размера БД и замедление её модификаций, не зависит от модели данных, используемой в основе СУБД. Алгоритмы и способы быстрого поиска — это проблематика теоретического программирования, а не результат «магических действий» администратора БД. Более подробно о построении индексов см. главу «Производительность SQL-запросов».

## Поддержка JSON

Начиная с версии 9.3, СУБД PostgreSQL обладает встроенным типом `json`, позволяющем хранить НДС в ранее упомянутом формате JSON.

Как и в случае XML, можно создать колонку соответствующего типа НДС.

```
CREATE TABLE products (
  id_prod integer NOT NULL,
  code     varchar(10) NOT NULL,
  name     varchar(50) NOT NULL,
  spec     json NOT NULL,
```

```
CONSTRAINT pk_products PRIMARY KEY (id_prod)
);
```

Заполним таблицу данными, аналогичным взятым из примера для XML.

```
INSERT INTO products (id_prod, code, name, spec)
SELECT 1, 'P01', 'Мука пшеничная в/с',
'{"packs": [
  {"weight": 1, "unit": "кг"},
  {"weight": 50, "unit": "кг"}]
}':::json
UNION ALL
SELECT 2, 'P02', 'Дрожжи',
'{"packs": [{"weight": 100, "unit": "г"}],
"expired": 10
}':::json
UNION ALL
SELECT 3, 'P03', 'Сахар',
'{"packs": [{"weight": 10, "unit": "кг"}],
"color": "белый"
}':::json
```

В отличие от XML, при приёме массива данных в таблицу будут определены значения для JSON-значений операторов.



Табл. 19. Основные операторы для работы с JSON

| Оператор | Тип и назначение операнда правой части | Что делает?                           | Пример и результат   |
|----------|--|---------------------------------------|--|
| ->       | Целое (номер элемента)                 | Возвращает элемент массива по номеру  | SELECT<br>'[1,2,3]':::json->2<br>AS value<br><br>value<br>json<br>-----<br>3 |
| ->       | Текст (имя элемента)                   | Возвращает значение элемента по имени | SELECT<br>'{"a":1,"b":2}':::json->'b'<br>AS value                            |

| Оператор | Тип и назначение операнда правой части | Что делает?   | Пример и результат   |
|----------|--|---|--|
|          |  |   | value<br>json<br>-----<br>2  |
| ->>      | Целое (номер элемента)                 | Возвращает элемент массива по номеру в виде текста  | SELECT<br>'[1,2,3]':::json->>2<br>AS value                             |
|          |  |   | value<br>text<br>-----<br>3  |
| ->>      | Текст (имя элемента)                   | Возвращает значение элемента по имени в виде текста | SELECT<br>'{"a":1,"b":2}':::json->>'b'<br>AS value                     |
|          |  |   | value<br>text<br>-----<br>2  |
| #>       | Массив строк (путь)                    | Возвращает элемент по заданному пути                | SELECT<br>'{"a":[1,2,3],"b":[4,5,6]}':<br>::json#>'{a,2}'<br>AS value  |
|          |  |   | value<br>json<br>-----<br>3  |
| #>>      | Массив строк (путь)                    | Возвращает элемент по заданному пути в виде текста  | SELECT<br>'{"a":[1,2,3],"b":[4,5,6]}':<br>::json#>>'{a,2}'<br>AS value |
|          |  |   | value<br>text<br>-----<br>3  |

Часто используемая в запросах функция `json_array_elements` возвращает элементы массива, транспонированные в виде колонки.

```
SELECT json_array_elements(
  '[{"weight": 100}, {"weight": 50}]') AS value
```

Результат:

```
value
json
-----
{"weight": 100}
{"weight": 50}
```

Запрос №1. Вывести все продукты, у которых специфицирован максимальный срок реализации.

```
SELECT *
FROM products p
WHERE (spec#>'{expired}') IS NOT NULL
```

Запрос №2. То же, но с выводом числового значения срока.

```
SELECT *, (spec#>'{expired}')::int
FROM products
WHERE (spec#>'{expired}') IS NOT NULL
```

Запрос №3. Вывести все спецификации упаковок товаров в колонки «Вес» и «Единицы измерения».

```
SELECT *,
  (json_array_elements(spec#>'{packs}')#>'{weight}')
  ::int AS "Вес",
  json_array_elements(spec#>'{packs}')#>'{unit}'
  AS "Единица изм."
FROM products p
```

Для часто используемых путей извлечения из документа значений можно строить индексы.

```
CREATE INDEX ON products((spec#>'{expired}'));
```

## Выводы

Современные реляционные СУБД представляют достаточно широкие возможности по работе с неполно структурированными данными (НСД), прежде всего с XML. В меньшей степени поддерживается JSON.

Возможность индексации НСД позволяет оптимизировать время поиска в БД относительно большого объёма. При этом проблемы достижения оптимума между производительностью поиска, операциями модификации и размером БД являются общей проблемой независимо от используемой модели данных.

Следует понимать, что работа со встроенными типами XML или JSON даже средствами РСУБД не делает этот подход реляционным. По-прежнему, программист оперирует понятиями соответствующей модели НСД.

Использование НСД в рамках РСУБД позволяет комбинировать подходы, опираясь на сильные стороны каждого из них.

Для использования НСД в условиях, приближенных к высокой нагрузке, необходимо выносить в основную таблицу все значимые атрибуты, по которым ведётся поиск.

## **Постраничные выборки**

Тема постраничных выборок в последнее время, одной и пузырями всплыла в связи с ростом разработки веб-приложений. Интерфейс пользователя в этом смысле несколько отличается от традиционного оконного, прежде всего тем, что вместо традиционного вывода данных на экран вместо их прокрутки. Поисковики в Интернет выдают сотни тысяч результатов с темами, подобной «как вывести записи с 100 по 150-ю». При этом мало кто задумывается о практической целесообразности вывода на экран результата выборки с возможностью постраничного просмотра, если она содержит многие сотни и тысячи записей.

Добавлю несколько рекомендаций веб-программистам.

- Ограничивайте размер выборки, за исключением специально оговорённых случаев. Нет смысла возвращать все 10500 строк, если пользователь ищет в БД населения города по фамилии Смирнов. Не стоит огорчать пользователя и нагружать СУБД бесполезными запросами с пролистыванием. Покажите вопрошающему первые несколько десятков записей и попросите уточнить запрос ввиду большого количества результатов. Если сгруппировать результаты в

иерархию, например, по именам или годам рождения с соответствующим интерфейсом, то можно выводить и большее число записей. Вспоминайте, как часто вы сами ходите дальше первой страницы поиска в Яндексе или Google.

- Если запрос все-таки содержит достаточное количество строк, чтобы показывать результирующую таблицу с функцией пролистывания, то кэшируйте результат. Объект «Набор данных» уровня сессии для корпоративного приложения с десятками-сотнями соединений пользователей будет работать быстрее, чем повторные запросы к СУБД с целью выкачать очередную страницу из 20 записей. В веб-приложениях число соединений может быть велико, но тем ценнее производительность кэширования, только механизмы усложняются.

Как писал классик, «величайшей ошибкой было бы думать», что постраничные выборки появились именно в веб-эпоху. Наиболее востребованная для них задача — передача данных пачками по N записей, где N может быть весьма велико. В этом параграфе мы и рассмотрим соответствующие средства, предоставляемые СУБД. Но все сказанное для веб-приложений тоже пригодится.

Средства постраничной выборки можно разделить на следующие группы:

- ограничения, выполняемые на уровне ядра СУБД. Это конструкции, являющиеся расширением синтаксиса SQL-запросов, воспринимаются СУБД однозначно и соответствующим образом учитываются на стадии оптимизации. Это наиболее быстрый способ в большинстве случаев, но могут быть и исключения, о них будет сказано ниже;
- ограничения на уровне пользовательских запросов и функций. Сюда входят разнообразные способы использования порядка следования записей в запросе и отсекающие «лишних» записей результат запроса, включая предварительные выборки во временную таблицу;
- ограничения серверных курсоров. Итог запроса не возвращается клиенту, на сервере открывается курсор, который позволяет построчно пролистывать результирующее множество данных;

- ограничения клиентских курсоров. Итог запроса целиком закачивается в DataSet клиента, вся дальнейшая навигация происходит в памяти клиентского приложения (им может быть и веб-сервер, и сервер приложений и служба).

## Обзор способов постраничной выборки

### Способ 1. Ограничения на уровне синтаксиса запроса

К ограничениям уровня синтаксиса SQL запросов относятся, например, возможности MySQL.

```
SELECT *
FROM orders
ORDER BY created ASC
LIMIT 101, 120 /* вывод строк с 101 по 120-ю */
```

Аналогично работают ограничения в PostgreSQL с несколько изменённой логикой.

```
SELECT *
FROM orders
ORDER BY created ASC
LIMIT 20 OFFSET 100 /* вывод строк с 1 по 120-ю */
```

Может показаться странным, но до версии 2000 Microsoft SQL Server не имел в своём арсенале возможностей ограничивать выборки на уровне ядра СУБД, указывая соответствующие критерии в теле запроса. И вот, наконец, у разработчиков этой передовой во многих отношениях СУБД, что называется «дошли руки».

```
SELECT *
FROM orders
ORDER BY created ASC
OFFSET 100 ROW FETCH NEXT 20 ROWS ONLY /* вывод строк с 101 по 120-ю */
```

Как вы могли заметить, во всех примерах имеется конструкция ORDER BY, хотя в случае MySQL и PostgreSQL она не является обязательной. Причина тому родом из теории множеств. Результат SQL-запроса — это неупорядоченное множество записей. Даже если дважды выполненный запрос вернёт одно и то же множество, порядок следования записей в нём не гарантирован. На практике, он зависит от физического размещения

записей в БД, которое может меняться при модификациях данных. Поэтому, возьмите за правило.

Выборки с ограничениями размера должны сопровождаться упорядочиванием результатов при помощи ORDER BY.

Почему вышеприведённые конструкции являются в большинстве случаев наиболее эффективными с точки зрения производительности?

Явное задание критериев ограничения учитывается при составлении плана запроса. Если на уровне пользовательских функций отсеивается уже результат, то на уровне ядра СУБД оптимизатор может решить, например, прервать внутреннее соединение двух таблиц на 100-й записи.

### **Способ 2. Использование функций ранжирования**

Тем не менее, встречаются случаи, когда ограничить выборку на уровне ядра не является корректным с логической точки зрения. Это многочисленные случаи, когда запросы выполняются с группировками, когда целью стоит, например, «выбрать группы с 10-ю по 20-ю». Те же функции можно использовать и для ограничения на уровне строк.

```
WITH ordered_sales AS (  
  SELECT  
    *,  
    row_number() OVER (  
      ORDER BY id_product, id_customer, sale_date  
    ) AS row_num  
  FROM sales  
)  
SELECT *  
FROM ordered_sales  
WHERE row_num BETWEEN 101 AND 120;
```

Оборотная сторона такого способа — меньшее быстродействие по сравнению с основным.

### **Способ 3. Временная таблица**

Если к выбранной странице или к их совокупности с 1 по N предполагается обращаться многократно, например, используя в

последующих соединения с другими таблицами, то может пригодиться способ на основе временной таблицы.

```
/* создание структуры временной таблицы */
SELECT * INTO #s FROM sales WHERE 1 = 0;
/* добавляем колонку - номер строки, первичный ключ */
ALTER TABLE #s ADD row_num INT NOT NULL IDENTITY(1, 1)
PRIMARY KEY;
/* заполняем данные */
INSERT INTO #s
SELECT TOP 120 *
FROM sales
ORDER BY id_product, id_customer, sale_date;
/* выборка последней страницы */
SELECT * FROM #s
WHERE row_num BETWEEN 101 and 120;
/* выборка третьей страницы */
SELECT * FROM #s
WHERE row_num BETWEEN 41 and 60;
```

Недостаток способа — необходимость предварительно выбирать все записи до N-й.

#### **Способ 4. Использование функции SELECT TOP**

Может встретиться СУБД, у которой реализовано только ограничение общего числа выводимых строк результата. Тогда решением может служить пересечение двойное использование этой функции со взаимно обратной сортировкой.

```
SELECT *
FROM
(
    SELECT TOP 20 *
    FROM
        (SELECT TOP 120 *
        FROM sales
        ORDER BY id_product ASC, id_customer ASC, sale_date
ASC
        ) t1
    ORDER BY id_product DESC, id_customer DESC, sale_date
DESC
) t2
ORDER BY id_product, id_customer, sale_date
```

## Способ 5. Серверный курсор

Если необходимо выбирать большие пачки записей из длинной таблицы, то основной способ (1) может оказаться менее производительным, чем серверный курсор, несмотря на большее время первоначальной загрузки. Ниже — пример для SQL Server, подробную документацию можно найти в MSDN по заголовку «Cursor Stored Procedures (Transact-SQL)».

```
DECLARE @handle int, @rows int;
EXEC sp_cursoropen
    @handle OUT,
    'SELECT * FROM sales ORDER BY id_product, id_customer,
    sale_date',
    1, /* 0x0001 - тип курсора (keyset-driven cursor) */
    1, /* только чтение */
    @rows OUT; /* общее количество записей */

EXEC sp_cursorfetch
    @handle,
    16, /* отсчет по абсолютному номеру строки */
    100, /* пропускаем первые 100 строк */
    20 /* выбираем 20 последующих */

EXEC sp_cursorclose @handle /* закрываем курсор */
```

## Способ 6. Стандартный ANSI SQL и переносимость

Данный способ представляет собой скорее академический интерес, показывая, что даже на минимальном уровне совместимости со стандартом задачу можно единообразно решить на любой СУБД. На практике, его быстродействие с ростом  $N$  падает как  $N^2$  (зависимость типа  $O(n^2)$ ), что не позволяет рекомендовать его для использования без дополнительных обоснований.

```
SELECT o.*
FROM orders o
WHERE
    (SELECT count(1)
     FROM orders ol
     WHERE ol.product_code <= o.product_code
    ) BETWEEN 101 AND 20
ORDER BY o.product_code ASC
```

Однако, если рассматривать расширенный стандарт ANSI SQL, то в него входит функция `row_number()`, реализованная во многих СУБД. Соответственно, вышеприведённый способ №2 также может быть отнесён к категории переносимых между СУБД методов.

## Тестирование способов постраничной выборки

Приведём подробное описание испытаний, чтобы читатели смогли понять суть подходов к такого рода практическим проверкам. В качестве подопытной системы используется Microsoft SQL Server 2012, но аналогичные тесты могут быть проведены на любой другой СУБД с небольшими изменениями SQL-сценариев.

Пусть имеется две таблицы: клиенты и продажи. Клиенты расположены в 7 странах. Необходимо выбирать информацию о продажах пачками по N записей относящиеся ко всем клиентам заданной страны продажи. В сценарии задан фильтр по Италии (IT), но при относительно равномерном распределении информации относительно стран выбор не является существенным. Результаты прогонов относятся к специально созданную для этих целей таблицу.

Конфигурация оборудования на данном испытании также не важна, поскольку проводится лишь относительное сравнение разных способов на одной и той же БД. Разумеется, на более быстром сервере ждать итогов придётся меньше, а объёмы данных можно нарастить.

Вначале создадим БД и таблицы, это будет первым SQL-сценарием в серии.

```
SET NOCOUNT ON;
CREATE DATABASE test_paging
GO
ALTER DATABASE test_paging SET RECOVERY SIMPLE
GO
ALTER DATABASE test_paging MODIFY FILE
(NAME=N'test_paging', SIZE=1024MB, MAXSIZE=UNLIMITED,
FILEGROWTH=512MB )
GO
ALTER DATABASE test_paging MODIFY FILE
```

```
(NAME=N'test_paging_log', SIZE=512MB, MAXSIZE=UNLIMITED,  
FILEGROWTH=256MB)
```

```
GO
```

```
USE test_paging
```

```
GO
```

```
CREATE TABLE dbo.customers (  
    id_customer      int          NOT NULL,  
    country_code     nchar(2)     NOT NULL,  
    name             nvarchar(255) NULL,  
    street_address   nvarchar(100) NULL,  
    city            nvarchar(40)  NULL,  
    postal_code      nvarchar(15) NULL,  
    CONSTRAINT pk_customers  
        PRIMARY KEY CLUSTERED (id_customer)  
)
```

```
GO
```

```
CREATE INDEX IX1_COUNTRY_CODE ON dbo.customers (country_code  
ASC)
```

```
GO
```

```
CREATE TABLE dbo.sales (  
    id_product      int          NOT NULL,  
    id_customer     int          NOT NULL,  
    sale_date       datetime     NOT NULL,  
    qty            int          NOT NULL,  
    CONSTRAINT pk_sales  
        PRIMARY KEY CLUSTERED (id_product, id_customer,  
sale_date),  
    CONSTRAINT fkl_orders_customers FOREIGN KEY(id_customer)  
        REFERENCES dbo.customers (id_customer)  
)
```

```
GO
```

```
CREATE TABLE dbo.results (  
    method int NOT NULL,  
    offset int NOT NULL,  
    page_size int NOT NULL,  
    attempt int NOT NULL,  
    duration_msec int NOT NULL,  
    CONSTRAINT pk_results  
        PRIMARY KEY CLUSTERED (method, offset, page_size,  
attempt)  
)
```

```
GO
```

Второй сценарий заполняет таблицы псевдослучайным образом сгенерированными данными.

```
USE test_paging
GO
CREATE VIEW dbo.rand2 AS
    SELECT rand(abs(convert(int, convert(varbinary, newid()))))
AS rand_value;
GO
SET NOCOUNT ON;

DECLARE
    @max_products_count int = 1000,
    @max_customers_count int = 10000,
    @max_sales_count int = 10000000,
    @min_date datetime = '20100101',
    @max_qty int = 1000;
DECLARE
    @max_days int = datediff(day, @min_date, getdate()),
    @i int;

TRUNCATE TABLE dbo.sales;
DELETE FROM dbo.customers;

PRINT 'Insert customers...'
SET @i = 1;
DECLARE @id_country int;
BEGIN TRANSACTION
WHILE @i <= @max_customers_count BEGIN
    SET @id_country = floor((SELECT rand_value * 7 FROM
rand2));
    INSERT INTO dbo.customers (id_customer, country_code, name,
street_address, city, postal_code)
    SELECT
        @i,
        CASE @id_country
            WHEN 0 THEN 'ES'
            WHEN 1 THEN 'FR'
            WHEN 2 THEN 'GE'
            WHEN 3 THEN 'IT'
            WHEN 4 THEN 'NL'
            WHEN 5 THEN 'RU'
            WHEN 6 THEN 'UK'
        END AS country_code,
```

```

    'Name' + convert(nvarchar(16), floor((SELECT rand_value
FROM rand2) * @max_customers_count + 1)),
    'Street ' + convert(nvarchar(16), floor((SELECT
rand_value FROM rand2) * 100000 + 1)),
    'City' + convert(nvarchar(16), floor((SELECT rand_value
FROM rand2) * 1000 + 1)),
    convert(nvarchar(16), floor((SELECT rand_value FROM
rand2) * 100000 + 1));
    IF @i % 1000 = 0 BEGIN
        COMMIT TRANSACTION
        BEGIN TRANSACTION
    END
    SET @i = @i + 1;
END
COMMIT TRANSACTION
PRINT 'Finished'

PRINT 'Insert sales data...'
SET @i = 1;
BEGIN TRANSACTION
WHILE @i <= @max_sales_count BEGIN
    INSERT INTO Sales (ProductID, CustomerID, sale_date,
qty)
    SELECT
        floor((SELECT rand_value FROM rand2) *
@max_products_count + 1),
        floor((SELECT rand_value FROM rand2) *
@max_customers_count + 1),
        dateadd(second, @i + floor((SELECT rand_value * 10 FROM
rand2)), @min_date),
        floor((SELECT rand_value FROM rand2) * @max_qty + 1)
    ;
    IF @i % 1000 = 0 BEGIN
        COMMIT TRANSACTION
        BEGIN TRANSACTION
    END
    IF @i % 100000 = 0
        PRINT convert(nvarchar(16), @i) + ' processed!';
    SET @i = @i + 1;
END
COMMIT TRANSACTION
PRINT 'Finished'
GO

DROP VIEW dbo.rand2

```

```
GO
```

```
SELECT count(1) AS sales_count, c.country_code  
FROM dbo.sales s INNER JOIN dbo.customers c ON s.id_customer  
= c.id_customer  
GROUP BY c.country_code  
ORDER BY c.country_code
```

Третий сценарий создаёт хранимые процедуры, по одной на каждый из испытываемых способов.

```
USE test_paging
```

```
GO
```

```
IF EXISTS(SELECT 1 FROM sys.views WHERE object_id =  
OBJECT_ID(N'dbo.test_sales_data'))
```

```
    DROP VIEW dbo.test_sales_data
```

```
GO
```

```
CREATE VIEW dbo.test_sales_data
```

```
AS
```

```
SELECT
```

```
    s.id_customer,  
    s.id_product,  
    s.qty,  
    s.sale_date,  
    c.country_code,  
    c.name
```



```
FROM
```

```
    dbo.sales s  
    INNER JOIN dbo.customers c  
        ON s.id_customer = c.id_customer
```

```
WHERE
```

```
    c.country_code = 'IT'
```

```
GO
```

```
IF EXISTS(SELECT 1 FROM sys.procedures WHERE object_id =  
OBJECT_ID(N'dbo.test_paging_m1'))
```

```
    DROP PROCEDURE dbo.test_paging_m1
```

```
GO
```

```
CREATE PROCEDURE dbo.test_paging_m1
```

```
    @offset int,  
    @page_size int
```

```
AS
```

```
BEGIN
```

```
    SET NOCOUNT ON;
```

```

SELECT * FROM dbo.test_sales_data
ORDER BY id_product, id_customer, sale_date
OFFSET @offset - 1 ROW FETCH NEXT @page_size ROWS ONLY
END
GO

IF EXISTS(SELECT 1 FROM sys.procedures WHERE object_id =
OBJECT_ID(N'dbo.test_paging_m2'))
    DROP PROCEDURE dbo.test_paging_m2
GO
CREATE PROCEDURE dbo.test_paging_m2
    @offset int,
    @page_size int
AS
BEGIN
    SET NOCOUNT ON;
    WITH ordered_sales AS (
        SELECT
            *,
            row_number() OVER( ORDER BY id_product, id_customer,
sale_date) AS row_num
        FROM dbo.test_sales_data
    )
    SELECT *
    FROM ordered_sales
    WHERE row_num BETWEEN @offset AND @offset + @page_size -
1;
END
GO

IF EXISTS(SELECT 1 FROM sys.procedures WHERE object_id =
OBJECT_ID(N'dbo.test_paging_m3'))
    DROP PROCEDURE dbo.test_paging_m3
GO
CREATE PROCEDURE dbo.test_paging_m3
    @offset int,
    @page_size int
AS
BEGIN
    SET NOCOUNT ON;
    -- Temporary table
    SELECT * INTO #s FROM dbo.test_sales_data WHERE 1 = 0;
    ALTER TABLE #s ADD row_num INT NOT NULL IDENTITY(1, 1)
PRIMARY KEY;

```

```

INSERT INTO #s
SELECT TOP (@offset + @page_size - 1) * FROM
dbo.test_sales_data
ORDER BY id_product, id_customer, sale_date;

SELECT * FROM #s
WHERE row_num BETWEEN @offset and @offset + @page_size - 1;
END
GO

IF EXISTS(SELECT 1 FROM sys.procedures WHERE object_id =
OBJECT_ID(N'dbo.test_paging_m4'))
DROP PROCEDURE dbo.test_paging_m4
GO
CREATE PROCEDURE dbo.test_paging_m4
@offset int,
@page_size int
AS
BEGIN
SET NOCOUNT ON;

SELECT *
FROM
(
SELECT TOP (@page_size)
FROM
(SELECT TOP (@offset + @page_size - 1) *
FROM dbo.test_sales_data
ORDER BY id_product ASC, id_customer ASC, sale_date
ASC
) t1
ORDER BY id_product DESC, id_customer DESC, sale_date
DESC
) t2
ORDER BY id_product, id_customer, sale_date
END
GO

IF EXISTS(SELECT 1 FROM sys.procedures WHERE object_id =
OBJECT_ID(N'dbo.test_paging_m5'))
DROP PROCEDURE dbo.test_paging_m5
GO
CREATE PROCEDURE dbo.test_paging_m5
@offset int,
@page_size int

```

```

AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @handle int, @rows int;
    EXEC sp_cursoropen
        @handle OUT,
        'SELECT * FROM dbo.test_sales_data ORDER BY id_product,
id_customer, sale_date',
        1,
        -- 16 - 0x0010 - Fast forward-only cursor
        -- 1 - 0x0001 - Keyset-driven cursor
        -- 8 - 0x0008 - Static cursor
        --select cast(0x0008 as int)
        1, -- Read-only
        @rows OUT; -- Contains total rows count

    EXEC sp_cursorfetch
        @handle,
        16, -- Absolute row index
        @offset, -- Fetch from row
        @page_size, -- Row count to
    EXEC sp_cursorclose @handle;
END
GO

```

Четвёртый сценарий производит запуск тестов, ведя запись хронологии и результатов в соответствующей таблице. В сценарии заданы повторения по 4 попытки для каждого из 5 способов. Параметром @offset\_count можно регулировать

```

USE test_paging
GO
SET NOCOUNT ON;

DECLARE
    @attempt_count int = 4,
    @methods_count int = 5,
    @offset_count int = 9,
    @method int,
    @offset int,
    @page_size int,
    @test_proc_name sysname,
    @started_at datetime,

```

```

    @i int,
    @j int;

TRUNCATE TABLE dbo.results;

SET @method = 1;
WHILE @method <= @methods_count BEGIN
    SET @test_proc_name = N'dbo.test_paging_m' +
convert(nvarchar(2), @method);

    SET @offset = 1;
    SET @page_size = 100;
    SET @i = 1;
    WHILE @i <= @offset_count BEGIN
        SELECT @offset =
            CASE
                WHEN @i = 1 THEN 1
                WHEN @i = 2 THEN 100
                WHEN @i = 3 THEN 1000
                WHEN @i = 4 THEN 10000
                WHEN @i >= 5 THEN 100000 * (@i - 4)
            END;
        DBCC DROPCLEANBUFFERS;
        DBCC FREEPROCCACHE;
        SET @j = 1;
        WHILE @j <= @attempt_count BEGIN
            SET @started_at = getdate();
            EXEC @test_proc_name @offset = @offset, @page_size =
@page_size;
            INSERT INTO dbo.results (method, offset, page_size,
attempt, duration_msec)
                SELECT @method, @offset, @page_size, @j,
datediff(millisecond, @started_at, getdate());
            SET @j = @j + 1;
        END;
        SET @i = @i + 1;
    END;
    SET @method = @method + 1;
END;
GO

```

Запуск этого сценария следует выполнять из командной строки, чтобы не тратить время на заполнение графического представления таблиц, возвращаемых запросами. Например, так.

```
sqlcmd.exe -S localhost -d test_paging -E -i 04_Test.sql -o Test.log
```

Подробнее, смотрите опции запуска утилиты sqlcmd в документации,

Наконец, самый простой заключительный скрипт, выводящий результаты: отдельно для первой попытки и усреднённые по совокупности всех последующих. Первая таблица, соответственно, даёт значения времени выполнения «холодных» запросов (при очищенном кэше СУБД), вторая — среднее время из трёх (по умолчанию) попыток при наличии данных и скомпилированного запроса в кэше.

```
SELECT *
FROM
(
  SELECT method, offset, page_size, duration_msec
  FROM dbo.results
  WHERE attempt = 1
) p
PIVOT
(
  AVG(duration_msec)
  FOR method IN ([1], [2], [3], [4], [5])
) pvt

SELECT *
FROM
(
  SELECT method, offset, page_size, duration_msec
  FROM dbo.results
  WHERE attempt > 1
) p
PIVOT
(
  AVG(duration_msec)
  FOR method IN ([1], [2], [3], [4], [5])
) pvt
```

При воспроизведении тестов на относительно слабом компьютере уровня рабочей станции, получаются следующие результаты (размер страницы N = 100 записей).

Табл. 20. Результаты прогона «холодных» запросов, миллисекунды

| offset | 1     | 2     | 3     | 4     | 5     |
|--------|-------|-------|-------|-------|-------|
| 1      | 6690  | 6693  | 6763  | 6906  | 35033 |
| 100    | 7260  | 6790  | 7190  | 7083  | 34736 |
| 1000   | 9230  | 8996  | 8570  | 8663  | 34653 |
| 10000  | 9330  | 8916  | 8570  | 7200  | 35433 |
| 100000 | 14946 | 15126 | 16156 | 15180 | 35103 |
| 200000 | 21120 | 20216 | 22863 | 21126 | 35736 |
| 300000 | 25223 | 23210 | 26063 | 24970 | 35473 |
| 400000 | 29923 | 25690 | 31336 | 29846 | 34933 |
| 500000 | 29326 | 28240 | 31526 | 30300 | 34750 |

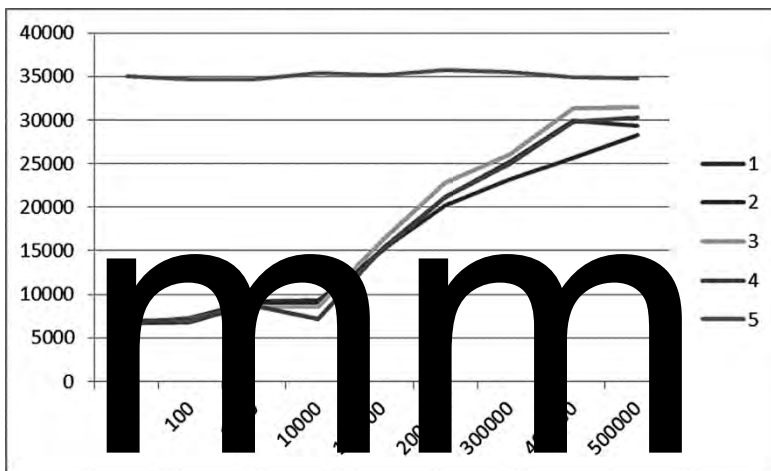


Рис.59. Графики результатов прогона «холодных» запросов

Табл. 21. Результаты прогона «горячих» запросов, среднее время, миллисекунды

| offset | 1    | 2     | 3    | 4    | 5    |
|--------|------|-------|------|------|------|
| 1      | 7    | 9     | 23   | 13   | 6669 |
| 100    | 16   | 12    | 24   | 16   | 6617 |
| 1000   | 33   | 34    | 48   | 37   | 7048 |
| 10000  | 140  | 210   | 243  | 131  | 6662 |
| 100000 | 1037 | 2084  | 1711 | 1182 | 6632 |
| 200000 | 3291 | 4206  | 4477 | 3218 | 6678 |
| 300000 | 4754 | 6340  | 5997 | 4923 | 6640 |
| 400000 | 1634 | 8407  | 2981 | 2006 | 6612 |
| 500000 | 1642 | 10641 | 3232 | 2196 | 6717 |

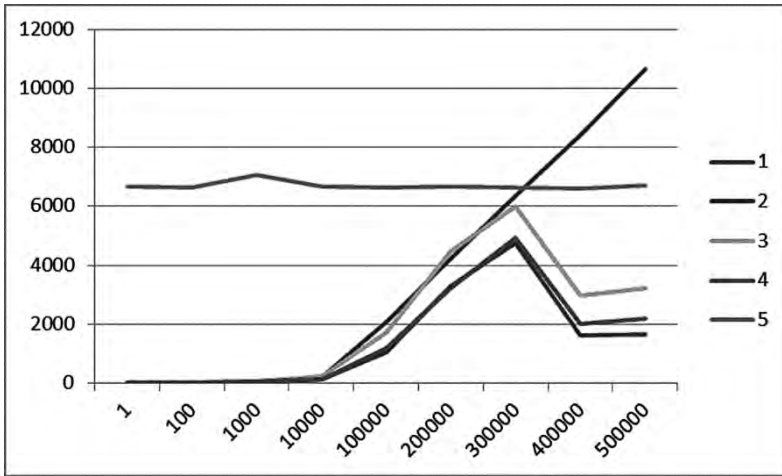


Рис.60. Графики результатов прогона «горячих» запросов

## Выводы

Испытания на примере MS SQL Server 2012 показывают, что штатный способ №1 ограничения размера выборки на уровне синтаксиса SQL-запроса является самым быстрым.

Между тем, единственным способом, показавшим стабильно одинаковое время отклика независимо от параметров выборки является метод серверного курсора №5.

## SQL и модульное тестирование

### Место модульного тестирования в системе испытаний

Важнейший этап разработки программной системы — её испытание на соответствие требованиям. Требования к системе распределяются по уровням детализации, и, значит, соответствующие им тесты имеют различную форму и ответственных за их создание. На самом нижнем уровне располагаются модульные тесты, для которых также используется жаргонное словечко «юнит-тест», калька с английского термина unit test.

Табл. 22. Уровни испытаний и требований

| Уровень                       | Тип требований          | Источник, документ  | Вид теста                              | Ответственный                |
|-------------------------------|-------------------------|---------------------|--|------------------------------|
| Система                       | Требования к системе    | Техническое задание | Системный тест (system test)           | Главный инженер/ конструктор |
| Подсистема (компонент, пакет) | Требования к подсистеме | Технический проект  | Интеграционный тест (integration test) | Ведущий инженер-программист  |
| Модуль                        | Требования к модулю     | Спецификация модуля | Модульный тест (unit test)             | Инженер-программист          |

Как следует из таблицы, ответственным за создание модульных тестов является инженер-программист, разрабатывающий данный модуль. Модульные тесты относятся к категории «белый ящик», потому что в отличие от ящика чёрного, внутреннее устройство испытываемого объекта известно.

Рекламируемые в последние годы методики разработки «от тестов» (TDD - Test Driven development) базируются на обязательном создании модульных тестов ещё до написания существенного кода. Преследуется цель 100 %го покрытия тестами кода приложения, где под покрытием имеется в виду отношение числа тестируемых функций/модулей к общему их числу. Разумеется, чем больше модульных тестов и больше покрытие, тем выше степень надёжности отдельных модулей. Однако подобная методика имеет и видимые недостатки:

- корректное выполнение модульных тестов не гарантирует соответствие системы требованиям на вышестоящих уровнях;
- сложность разработки модульных тестов сравнима со сложностью разработки тестируемого кода. Таким образом, общее время программирования увеличивается в 2-3 раза.

В самом деле, работоспособность отдельных компонентов компьютера вовсе не гарантирует, что, собранный вашими руками из купленной россыпи деталей, он заработает сразу и с нужной производительностью. Нельзя также быть уверенным, что все приложения будут корректно

функционировать после установки. Именно поэтому поставщики компьютеров и ПО предлагают клиентам совместимые конфигурации, прошедшие системные испытания.

## **Особенности разработки на процедурных расширениях SQL**

В традиционных средах программирования, таких как C++, Delphi, Java, C# и др., модульное тестирование — обычная практика: имеются соответствующая инфраструктура, интегрированная со средой разработки, библиотеки и стандартные методики создания и прогона тестов вручную или в автоматическом режиме.

Программисты приложений баз данных, разрабатывающие много серверного кода на SQL и его процедурных расширениях, оказываются в худшем положении: поставщики СУБД, как правило, не включают в комплект не только инструменты для модульного тестирования, но и порой даже простые средства отладки и трассировки. Следует сказать, что отсутствие пошаговой отладки является в данной области критичным фактором. Далеко не все СУБД поддерживают подобие модульности для разрабатываемых хранимых процедур и функций. Задача создания инфраструктуры инструментов и методики для модульного тестирования ложится на плечи программиста.

## **Пример задачи для модульного теста**

Рассмотрим пример разработки с использованием модульных тестов, реализованный для Microsoft SQL Server (версия 2008 и выше). Общие принципы подхода не привязаны к СУБД, поэтому вы сможете использовать его и в других случаях с минимальными изменениями.

Положим, на базе имеющейся ежедневной статистики продаж продукции в магазинах требуется составить недельные прогнозы продаж на заданный период методом простой экстраполяции. Упрощённая схема данных будет выглядеть следующим образом.

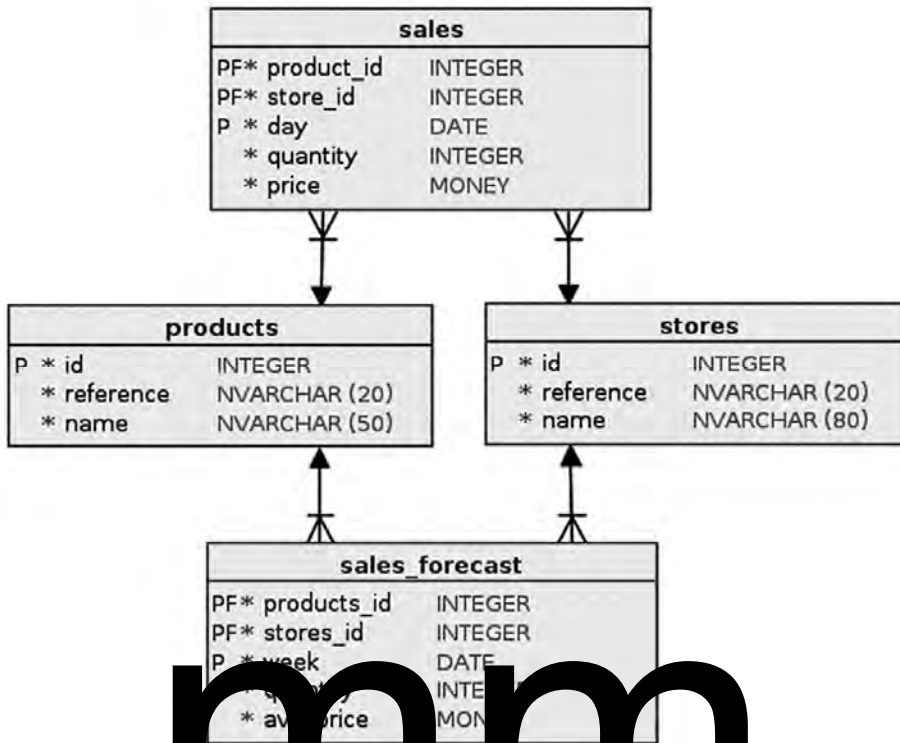


Рис.61. Схема данных для примера модульного тестирования

В таблицах `products` и `stores` хранятся списки товаров и данные о магазинах. В таблице `sales` ведётся статистика продаж: количество и цена — в разрезе «продукт—магазин—день». А в таблицу `sales_forecasts` нужно внести данные прогноза продаж: количество и среднюю цену.

Функция расчёта упрощена: на входе имеем даты начала и окончания будущего периода и начальную дату прошлого. На базе этих сведений и будет сделан линейный прогноз. Данные по продажам консолидируются с уровня дней до недель, причём их количество суммируется, а цена вычисляется средняя.

Исходные тексты программ примера находится в приложении к книге, здесь же мы приведём только основные элементы. Хранимая процедура `sales_forecast_init` (исходный файл `SalesForecast.sql`), производит вычисления и заполняет таблицу прогноза.

```
CREATE PROCEDURE dbo.sales_forecast_init
```

```

    @first_week      date,
    @last_week       date,
    @first_week_ref  date
AS
BEGIN
    SET NOCOUNT ON;

    DELETE FROM sales_forecast
    WHERE week BETWEEN @first_week AND @last_week;

    INSERT INTO sales_forecast
        (store_id,
         product_id,
         week,
         quantity,
         avg_price)
    SELECT s.store_id,
           s.product_id,
           w.start_date,
           sum(s.quantity),
           avg(s.price)
    FROM sales_forecast s
    LEFT OUTER JOIN
        dbo.get_weeks_list(@first_week, @last_week) AS w
    ON s.start_date BETWEEN
        dateadd(ww,
                datediff(ww, @first_week, w.start_date),
                @first_week_ref)
        AND
        dateadd(ww,
                datediff(ww, @first_week, w.start_date) +
1,
                @first_week_ref)
    GROUP BY s.store_id,
             s.product_id,
             w.start_date;
END;

```

Функция `utils_get_weeks_list()` возвращает таблицу-список недель между двумя заданными датами. Неделя задаётся датой первого дня (понедельника). Например, вызов функции

```
utils_get_weeks_list('20080204', '20080218')
```

возвратит таблицу из трёх строк:

| <code>week_num</code> | <code>start_date</code> |
|-----------------------|-------------------------|
| 1                     | 2008-02-04              |
| 2                     | 2008-02-11              |
| 3                     | 2008-02-18              |

Теперь требуется создать модульные тесты для проверки не только нашей основной процедуры, но и вспомогательных функций.

## Создаём специализированный макроязык

Хотя в столь простом случае можно обойтись исключительно средствами самого Transact SQL, решение получилось бы достаточно громоздким. Например, нам понадобятся стандартные процедуры проверок типа `assert`, генерирующие ошибку, если величина не равна/равна/больше/меньше заданной. Но в Transact SQL при вызове процедур нельзя напрямую передавать им значения, полученные из SQL-запросов или других функций. Следовательно, придётся всякий раз объявлять и инициализировать локальные переменные, а затем передавать их в процедуру. Проверить и избежать этих и других других неудобств поможет макропрограммирование.

Определим несколько макросов, которые будем использовать в тексте процедур на Transact SQL. Перед трансляцией процедуры в СУБД исходный текст проходит предварительную обработку макропроцессором, макросы раскрываются, получается чистый SQL. Подобный принцип активно применяется в языках семейства Си/C++. По своей сути и назначению же макропрограммирование позволяет обойти ограничения любого языка и создать поверх него собственный предметный язык более высокого уровня для эффективного решения частных задач.

В качестве макропроцессора используется GNU `m4`, входящий в стандартный инструментарий для любого Linux-окружения. Версия `m4` для Windows включена в пример.

Исходные файлы SQL (с макросами) имеют расширения `.scm`, но это ограничение не строгое, а введённое лишь с целью различать исходники на SQL с внедрёнными макросами от чистого Transact SQL. Обработанный

макропроцессором файл SQM превращается в SQL после запуска трансляции из командной строки.

```
D:\sqlunit\SalesTest>m4 -I .\Include
UtilsTest.sqm >UtilsTest.sql
```

Текст процедуры модульного теста функции `utils_get_weeks_list()` с использованием макросов будет выглядеть более наглядно, чем если бы мы остались в рамках Transact SQL.

```
include(Common.sqh)
include(MSSQL.sqh)
include(SqlUnit.sqh)

DeclareProcedure(dbo.test_utils_get_weeks_list)
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @start_date    date,
            @end_date      date,
            @weeks_to_add  int;

    SELECT @start_date = getdate(),
           @weeks_to_add = 7;
    SELECT @end_date = dateadd (ww, @week_to_add,
@start_date);

    SQLUnit_AreEquals(
        int,
        {SELECT count(*)
        FROM dbo.utils_get_weeks_list(@start_date,
@end_date)},
        @weeks_to_add + 1);

    SQLUnit_AreEquals(
        int,
        {SELECT max(week_num)
        FROM dbo.utils_get_weeks_list(@start_date,
@end_date)},
        @weeks_to_add + 1);
END;
```

Если посмотреть на результат раскрытия макроса `SQLUnit_AreEquals` в сгенерированном файле `UtilsTest.sql`, то можно увидеть кусок

рутинного кода, не подлежащего факторизации и оформлению в виде хранимой процедуры или функции.

```
DECLARE
    @Val18 int,
    @Val19 int,
    @ValStr18 nvarchar(255),
    @GivenValStr18 nvarchar(255);
SELECT
    @Val18 = (SELECT count(*) FROM
dbo.utils_get_weeks_list(@start_date, @end_date)),
    @Val19 = (@weeks_to_add + 1);
IF (@Val18 IS NULL OR NOT(@Val18 = @Val19))
BEGIN
    SELECT @ValStr18 = convert(nvarchar(255), @Val18),
           @GivenValStr18 = convert(nvarchar(255), @Val19);
    raiserror('Error. Value is "%s". Expected "%s".
Expression: SELECT count(*) FROM
dbo.utils_get_weeks_list(@start_date, @end_date)', 16, 1,
@ValStr18, @GivenValStr18);
END;
```

Как можно увидеть, четыре строки (красные) проверки целочисленных результатов двух выражений на равенство раскрываются 14 строк чистого Transact SQL! А ведь подобных проверок в теле рядовой процедуры теста используется в среднем около десятка.

Теперь взглянем на исходный текст модульных тестов в файле SalesForecastTest.sql. В процедуре test\_sales\_forecast\_setup производится заполнение таблиц временными данными для теста.

```
DeclareProcedure(dbo.test_sales_forecast_setup)
AS
BEGIN
    SET NOCOUNT ON;

    /* Fill test data */
    INSERT INTO products(id, reference, name)
    SELECT newid(), 'TEST#prod1', 'Product 1'
    UNION
    SELECT newid(), 'TEST#prod2', 'Product 2'
    UNION
    SELECT newid(), 'TEST#prod3', 'Product 3'
```

```

INSERT INTO stores(id, reference, name)
SELECT newid(), 'TEST#store1', 'Store1 Msk'
UNION
SELECT newid(), 'TEST#store2', 'Store2 SPb';

INSERT INTO sales (store_id, product_id, [day], quantity,
price)
SELECT s.id, p.id, d.start_date,
        convert(int, ((SELECT rand_value FROM rand2) *
1000)),
        (SELECT rand_value FROM rand2) * 100.00
FROM products p
CROSS JOIN stores s
CROSS JOIN
        dbo.utils_get_days_list(TestWeek11, TestWeek12) d
WHERE p.reference LIKE 'TEST#%' AND
        s.reference LIKE 'TEST#%'

SQLUnit_MoreThan(
    {int},
    {SELECT
        FROM sales
        INNER JOIN stores ON sales.store_id = stores.id
        INNER JOIN products
        ON sales.product_id = products.id
        WHERE stores.reference LIKE 'TEST#%' AND
        products.reference LIKE 'TEST#%' AND
        day BETWEEN TestWeek11 AND TestWeek12
    },
    {0},
    {Sales table has no data});
END;

```

В процедуре `test_sales_forecast_tear-down`, производящей очистку, мы удаляем эти данные.

```

DeclareProcedure(dbo.test_sales_forecast_tear-down)
AS
BEGIN
    SET NOCOUNT ON;

    DELETE FROM sales_forecast
    WHERE product_id IN
        (SELECT id FROM products WHERE reference LIKE 'TEST#%');

```

```

DELETE FROM sales
WHERE product_id IN
    (SELECT id FROM products WHERE reference LIKE 'TEST#%');

DELETE FROM products WHERE reference LIKE 'TEST#%';
DELETE FROM stores WHERE reference LIKE 'TEST#%';
END;

```

Термины *setup* (инициализация) и *teardown* (очистка) — стандарты де-факто в модульном тестировании, поэтому мы их не изменяем. Сам тест проводится в процедуре `test_sales_forecast_init`, также использующей макросы.

```

DeclareProcedure(dbo.test_sales_forecast_init)
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @first_week    date,
            @last_week     date,
            @first_week_ref date;

    SELECT
        @first_week = TestWeek,
        @last_week  = TestWeek,
        @first_week_ref = TestWeek;

    EXEC dbo.sales_forecast_init
        @first_week    = @first_week,
        @last_week     = @last_week,
        @first_week_ref = @first_week_ref;

    SQLUnit_MoreThan(
        int,
        {SELECT count(*)
         FROM sales_forecast
          INNER JOIN stores
                ON sales_forecast.store_id = stores.id
          INNER JOIN products
                ON sales_forecast.product_id = products.id
         WHERE stores.reference LIKE 'TEST#%' AND
              products.reference LIKE 'TEST#%' AND
              week BETWEEN @first_week AND @last_week
        },
        {0},
        {Sales forecasts has no data});

```

```

/* Провера расчета на случайно выбранную неделю */
DECLARE @store_id      uniqueidentifier,
        @product_id   uniqueidentifier,
        @week_to_check date,
        @ref_week     date;

SELECT @store_id = id FROM stores WHERE reference =
'TEST#store1';
SELECT @product_id = id FROM products WHERE reference =
'TEST#prod2';
SELECT @week_to_check =
    dateadd(ww,
            datediff(ww, @first_week, @last_week) *
            (SELECT rand_value FROM rand2),
            @first_week);
SELECT @ref_week =
    dateadd(ww,
            datediff(ww, @first_week,
@week_to_check),
            @first_week_ref);
SQLUnit_AreEqual(
    int,
    {SELECT quantity
    FROM sales_forecast
    WHERE store_id = @store_id AND
        product_id = @product_id AND
        week = @week_to_check
    },
    {SELECT SUM(quantity)
    FROM sales
    WHERE store_id = @store_id AND
        product_id = @product_id AND
        [day] BETWEEN @ref_week AND
        dateadd(ww, 1, @ref_week)
    },
    {Invalid quantity});

SQLUnit_AreEquals(
    int,
    {SELECT avg_price
    FROM sales_forecast
    WHERE store_id = @store_id AND
        product_id = @product_id AND
        week = @week_to_check

```

```

    },
    {SELECT AVG(price)
     FROM sales
     WHERE store_id = @store_id AND
           product_id = @product_id AND
           [day] BETWEEN @ref_week AND
           dateadd(ww, 1, @ref_week)
    },
    {Invalid average price});
END;

```

Запуск теста выполняет процедура `test_sales_forecast_all`.

```

DeclareProcedure(dbo.test_sales_forecast_all)
AS
BEGIN
    SET NOCOUNT ON;
    EXEC dbo.test_sales_forecast_setup;
    EXEC dbo.test_sales_forecast_init;
    EXEC dbo.test_sales_forecast_teardown;
END;

```

Теперь, имея файл `run_tests.cmd` и команду для запуска тестов, мы можем автоматизировать процесс с помощью обычного командного файла `run_tests.cmd`. Утилита выполняет SQL из командной строки, поставляемой с любой СУБД. Для MS SQL Server это уже упоминавшаяся `sqlcmd.exe`.

```

@echo off

rem установка переменных окружения
call "%~d0%~p0..\set_env.cmd"

echo.Testing utils module...
sqlcmd -b -r 0 -E -S %SERVER_NAME% -d %DATABASE_NAME% -Q
"EXEC dbo.test_utils_all" -o %OUTPUT_FILE%
if errorlevel 1 goto batch_failed

echo.Testing sales forecast module...
sqlcmd -b -r 0 -E -S %SERVER_NAME% -d %DATABASE_NAME% -Q
"EXEC dbo.test_sales_forecast_all" -o %OUTPUT_FILE%
if errorlevel 1 goto batch_failed

goto all_done

```

```
:batch_failed
echo Test FAILED
exit /b 1

:all_done
echo Test OK
exit /b 0
```

Запускаем командный файл в консоли Windows и видим непосредственный итог наших тестов. Возврат статуса выполнения (0 или 1 при ошибке) позволяет встроить запуск модульных тестов в систему их автоматического выполнения в рамках постоянной интеграции.

## Остановиться и оглянуться

Обратите внимание, процедура `test_sales_forecast_init` проводит весьма простую проверку: мы сверяем цифры по одному товару и одному магазину за единственную неделю, выбранную случайным образом из заданного диапазона. При этом её текст даже с применением лаконичных макросов растягивается на 110 строк. Если же добавить сюда ещё текст, сгенерированный SQL-файлом. Если ещё учесть примерно 80 строк предварительной инициализации и очистки, то получается очень много, ведь текст собственной тестируемой процедуры `sales_forecast_init` занимает всего 33 строки! Соотношение объёма основного кода к тестирующему получается около одного к четырём-пяти.

Существует рабочее эмпирическое правило: если тест получился короче тестируемой процедуры, значит он недостаточно её тестирует.

Данная картина типична для разработки с использованием модульных тестов, где отношение объёма тестируемого кода к тестам примерно один к двум-четырёх. Отсюда и неизбежные дополнительные затраты времени, превышающие создание собственно кода приложения. Однако труд не пропадёт даром: надёжность вашего модуля возрастет, а процесс поиска и предупреждения ошибок будет систематичен.

Окончательный выбор оптимального покрытия, соотношения между объёмом программного кода модулей и модульных тестов, будет зависеть от

многих факторов, в первую очередь, от степени критичности приложения и стоимости простоя в эксплуатации.

## **Производительность SQL-запросов**

### **Общие рекомендации**

Прежде чем приступать к следующим главам, посвящённым выявлению проблем, хотелось бы дать несколько общих рекомендаций по программированию на SQL, с целью минимизации самой возможности их появления.

Уже упоминалась особенность реляционных СУБД — множественная обработка. Забудьте про переменные, циклы, про указатель `this`. «`This`» теперь указывает не на объект, а на множество, и операции будут проводиться одновременно со всеми элементами множества. Если в традиционном программировании для изменения элементов списка по условию надо писать цикл, то в SQL все решается одним оператором.

Проведите несколько запросов с помощью «Введение в SQL» [3], чтобы на несложных примерах понять всю мощь множественного подхода к обработке данных.

Не используйте курсоры и навигационный подход. Практика показывает, что любую задачу, использующую курсоры, можно перепрограммировать без него. Производительность при этом возрастает на порядки.

Для функциональной декомпозиции используйте процедуры и функции, принимающие и/или возвращающие множества. В противном случае вам придётся программировать курсоры, вызывающие на каждую строку соответствующую функцию.

```
/* Такая реализация может привести
   к навигационным подходам и курсорам */
CREATE PROCEDURE check_qty(@qty integer, @id_sku integer)
AS
BEGIN
    SET NOCOUNT ON;
    IF @qty > (SELECT qty FROM stock WHERE id_sku = @id_sku)
        RAISERROR('Не хватает запаса!', 11, 1)
END
```

```

GO

/* Реализация на основе множественного подхода */
CREATE TYPE t_requirement_list AS TABLE (
    id integer,
    qty integer)
GO

CREATE PROCEDURE check_qty(
    @quantities t_requirement_list READONLY
)
AS
BEGIN
    SET NOCOUNT ON;
    IF EXISTS(SELECT 1
        FROM stock s
            INNER JOIN @quantities q ON s.id_sku = q.id
            WHERE q.qty > s.qty)
        RAISERROR('Не хватает запаса!', 11, 1)
END
GO

```

Используйте предикат EXISTS вместо SELECT COUNT(1), потому что для проверки существования извлекается только первый элемент, а подсчёт количества просматривает все выбранные строки.

```

/* Плохой код */
SELECT *
FROM sales s
WHERE (SELECT COUNT(1)
    FROM sales s2
    WHERE s.id_customer <> s2.id_customer AND
        s.id_product = s2.id_product AND
        s.qty = s2.qty) > 0
/* Так будет быстрее */
SELECT *
FROM sales s
WHERE EXISTS(SELECT 1
    FROM sales s2
    WHERE s.id_customer <> s2.id_customer AND
        s.id_product = s2.id_product AND
        s.qty = s2.qty)

```

Вместо связанных подзапросов старайтесь использовать обычные соединения. Не все оптимизаторы запросов могут отловить эту ситуацию.

```

/* Связанный подзапрос может привести
   к вложенным циклам в плане выполнения */
SELECT *
FROM sales s
WHERE
  qty < (SELECT AVG(qty)
        FROM sales s2
        WHERE s2.id_product = s.id_product)

/* Соединение может выполняться разными
   и более быстрыми способами */
SELECT s.*
FROM sales s
  INNER JOIN
    (
      SELECT AVG(qty) AS avg_qty, id_product
      FROM sales s2
      GROUP BY id_product
    ) s3
  ON s.id_product = s3.id_product
WHERE s.qty < s3.avg_qty

```

## Анализ плана выполнения запроса

Как в общем случае понять, эффективен ли запрос? Если в среде разработки, где данные далеки от реальных объёмов, запрос будет выполняться быстро, это не гарантирует проблемы при передаче очередной версии на тестирование.

Объективная оценка основывается на анализе плана выполнения запроса на самых ранних этапах. Будет лучше, если объёмы данных в среде разработки будут хотя бы одного порядка с предполагаемыми. Если это по каким-либо причинам невозможно, остановитесь на генерации тестовых данных разработчика порядка сотен тысяч строк — это объёмы, когда разница между плохим и хорошим планами будет уже заметна.

Как правило, СУБД предоставляет программисту возможность анализа планов двух видов:

- оценочный план, создаваемый оптимизатором до выполнения запроса;
- реальный план, получаемый по итогам запроса.

Наличие двух планов объясняется возможностями оптимизатора корректировать план по ходу выполнения. Например, если результат выборки подзапроса с фильтром на основании плотности значений в колонке оценивался в несколько тысяч записей, а в реальности было выбрано менее сотни, то дальнейшие способы соединения могут быть пересмотрены.

Реальный план сохраняется в кэше СУБД. Он будет заново использован при следующем запуске запроса, увеличивая производительность в общем случае.

Инструментарий СУБД может иметь функцию графического представления плана выполнения запроса в виде дерева, корнем которого является результирующий набор данных. Если план невелик, то удобнее просматривать его в виде масштабируемого изображения, но когда SQL-запрос сложен, включает в себя десятки таблиц и подзапросов, найти нужный фрагмент становится трудно.

Функция выдает план в текстовом виде и присутствует, как правило, всегда, но разбираться в ней трудно.

Возьмём для примера БД в главы «Ограниченные выборки», предварительно выполнивую дабыми, и выполним сложный запрос, подсчитывающий число продаж, количество товара в которых меньше среднего в 5 и более раз.

```
SELECT s.*
FROM sales s
  INNER JOIN (
    SELECT AVG(qty) / 5 AS avg_qty, id_product
    FROM sales s2
    GROUP BY id_product
  ) s3
  ON s.id_product = s3.id_product
WHERE s.qty < s3.avg_qty
```

Поскольку запрос достаточно простой, графическая интерпретация плана будет удобной для чтения.

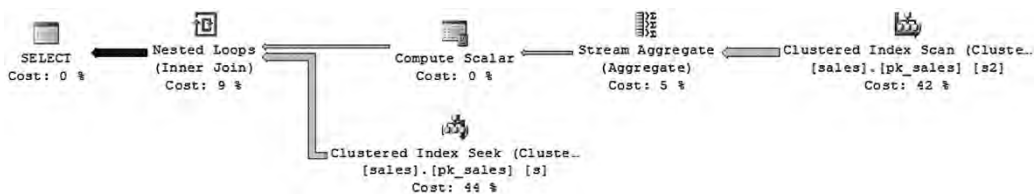


Рис.62. Схема плана выполнения запроса

Для получения текстового представления плана необходимо добавить перед запросом команду установки опции его показа (команда специфична для SQL Server ).

```
SET SHOWPLAN_TEXT ON
GO
```

Сокращённый фрагмент текста плана выполнения того же запроса выглядит следующим образом.

```
StmtText
-----
--
|--Nested Loops (Inner Join, OUTER REFERENCES ...)
|  |--Compute Scalar (DEFINE SCALAR EXPRESSION)
|  |  |--Stream Aggregate (GROUP BY: ([s2].[id_product] ...))
|  |  |--Clustered Index Scan (OBJECT: ([sales].[pk_sales] [s2]))
|  |--Clustered Index Seek (OBJECT: ([sales].[pk_sales] [s2]))
```

Из плана видно, что конечный результат получается в результате вложенного цикла (Nested Loops) по промежуточной выборке, подсчитывающей сгруппированное по товарам среднее количество их продаж (Compute Scalar, Stream Aggregate). В цикле на каждую запись промежуточной выборки осуществляется поиск по кластерному индексу (Clustered Index Seek). В свою очередь, промежуточная выборка сканирует всю таблицу sales, организованную в виде кластера (Clustered Index Scan).

Можно ли признать данный план эффективным? Для этого нужно знать, какие элементы плана потенциально могут быть узким местом.

Прежде всего, следует ориентироваться на оценку стоимости операций, выставляемую СУБД каждому элементу плана выполнения запроса. К сожалению, в текстовом виде у SQL Server эта информация отсутствует,

поэтому придётся прибегнуть к анализу графического представления или его «исходника» в виде XML.

```
SET SHOWPLAN_TEXT OFF
SET SHOWPLAN_XML ON
GO
```

Оптимизатор СУБД оценивает стоимость в процентах от общей. Соответственно, необходимо обратить внимание на максимальные значения.

Кроме стоимости, под подозрение попадают элементы следующих типов:

- table scan — последовательное сканирование таблицы, данные которой организованы в виде «кучи»;
- clustered index scan — сканирование таблицы, организованной в кластер;
- index scan — сканирование индекса;
- hash join — соединение с помощью хэширования.

В нашем примере наиболее дорогими этапами выполнения запроса является сканирование таблицы `products` (42 %) и поиск по кластерному индексу (44 %).

Из текста запроса видно, что предварительная выборка не содержит условий фильтрации, поэтому её полное сканирование для вычисления агрегата `avg()` является нормальным поведением и не может быть улучшено непосредственным образом.

Второй дорогостоящий элемент — поиск по кластерному индексу. На самом деле, поиск по кластеру является наиболее быстрым способом нахождения записи. Но в нашем случае соединение производится по единственной колонке `id_product`, входящий в составной кластерный индекс. Зная, что оптимизатор использует метод вложенного цикла, можно попытаться улучшить производительность, создав дополнительный индекс по колонкам `id_product` и `qty`.

Вначале замеряем время выполнения запроса в текущих условиях. Для этого в опциях SQL-консоли (Query options) нужно включить «SET STATISTIC TIME» и «SET STATISTIC IO».

Перед выполнением запроса рекомендуется сбросить кэш планов и данных, чтобы не вносить в сравнения возможные разночтения.

```
DBCC DROPCLEANBUFFERS
DBCC FREEPROCCACHE
```

Для существующего запроса получаем результат:

```
Table 'sales'. Scan count 1001, logical reads 110204,
physical reads 0, read-ahead reads 0, lob logical reads 0,
lob physical reads 0, lob read-ahead reads 0.
SQL Server Execution Times:
    CPU time = 4236 ms,  elapsed time = 14858 ms.
```

Теперь проверим первую гипотезу, строим дополнительный индекс со всеми опциями по умолчанию.

```
CREATE INDEX ix1_sales ON dbo.sales (id_product, qty)
```

Перезапускаем запросы-перехватчики, можно увидеть, что план немного изменился. Вместо Clustered Index Seek, теперь используется обычный Index Seek (Nonclustered), который в общем случае медленнее. Однако его стоимость по указанной выше причине (составной кластерный индекс) оказывается в разы ниже и падает до 12 %. Теперь наиболее дорогим элементом остаётся сканирование таблицы — целых 67 %.

Замер времени и ввода/вывода показывает небольшое улучшение.

```
Table 'sales'. Scan count 1001, logical reads 35549, physical
reads 0, read-ahead reads 1, lob logical reads 0, lob
physical reads 0, lob read-ahead reads 0.
SQL Server Execution Times:
    CPU time = 3244 ms,  elapsed time = 13113 ms.
```

Стоит ли радоваться улучшению? Не всегда. Надо понимать, что за полученный выигрыш придётся платить: за счёт нового индекса размер БД увеличился, а скорость вставки в таблицу снизилась. Локальное улучшение потенциально может ухудшить характеристики других запросов.

Избежать такой неоднозначной ситуации в случае достаточно сложной системы, разрабатываемой коллективом, можно двумя основными

способами, не исключаящими, а, скорее, взаимодополняющими друг друга:

- привлекать к работе администратора БД, который имеет общее видение работы всех приложений с СУБД и может на экспертном уровне оценить последствия добавлений новых индексов и других изменений;
- разработать систему автоматизированных тестов, прогон которых показывает возможную регрессию на отдельных операциях.

### ***Поиск узких мест***

Даже если автоматизированная информационная система находится в эксплуатации продолжительное время, в один не самый прекрасный день может наступить момент, когда пользователи начнут жаловаться на увеличение времени отклика. У системы, переданной в опытную эксплуатацию, вероятность такого поворота дел ещё выше, особенно при недостаточном уровне мониторинга (подробнее см. в следующей главе).

Как найти узкое место?

Прежде всего, следует определить, является ли такое узким местом собственно база данных, являющаяся конечным пунктом назначения запросов пользователей. Для такой диагностики промышленные СУБД обладают более или менее развитыми средствами мониторинга и трассировки.

В идеальном случае задержку можно стабильно воспроизвести. Тогда средства трассировки оказываются наиболее подходящими задачей. Весьма удобным инструментом является SQL Server Profiler, позволяющий в реальном масштабе времени наблюдать и регистрировать запросы, отфильтрованные по самым разнообразным критериям, например, по названию приложения или имени хост-машины.

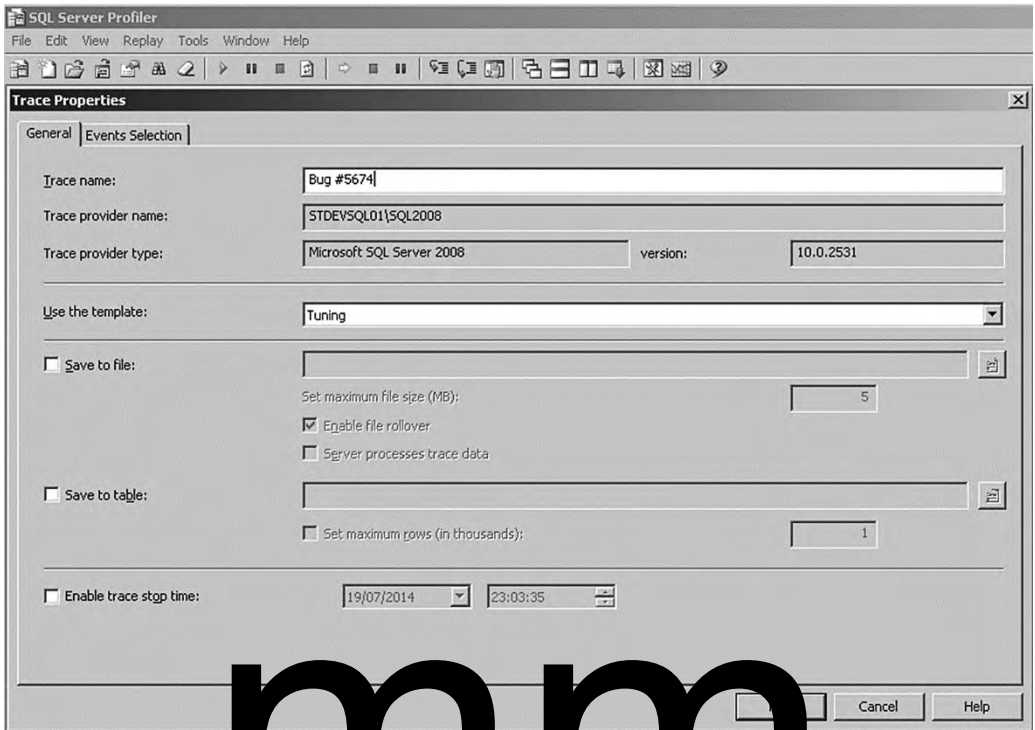


Рис.63. Установка свойств для трассировки

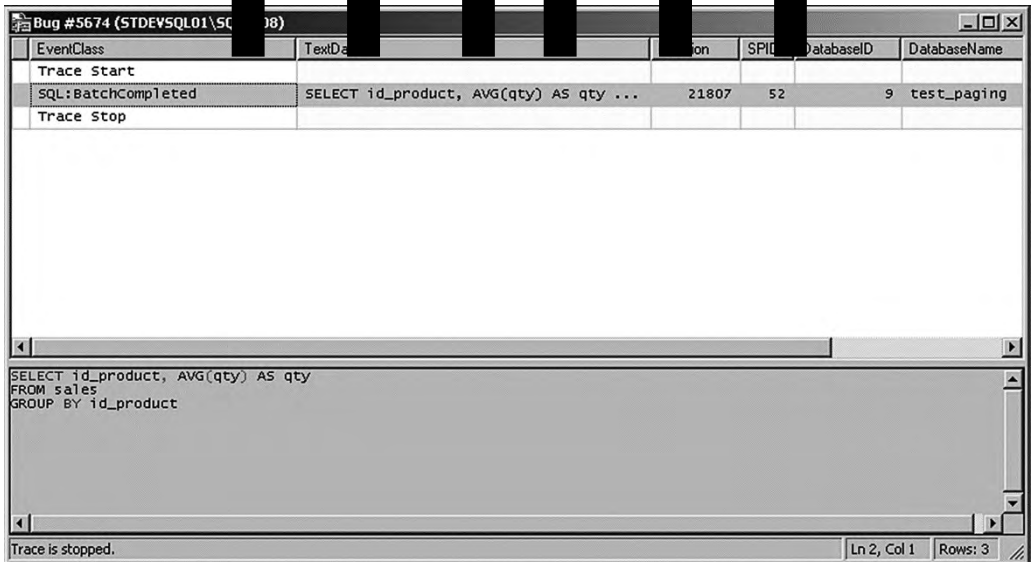


Рис.64. Результат трассировки средствами SQL Server Profiler

Итак, проблему можно воспроизвести в тестовой или эксплуатационной среде, запускаем утилиту трассировки с соответствующими настройками, на рабочем месте пользователя, на сервере пакетной обработки или ещё где-то на фронтальном уровне запускаем процесс и смотрим результат.

Из результата трассировки можно непосредственно определить время выполнения запроса или их серии, вызывающие нарекания по быстродействию. Если показанная цифра является гораздо меньшей, чем время отклика системы, то проблема кроется не в СУБД, а в вышестоящих слоях. Если же зафиксированное время соотносится с неприемлемо долгим ожиданием реакции системы на пользовательском уровне, то вам крупно повезло, с первой попытки найдено узкое место!

Теперь — дело техники, надо вычленить из трасс запросы, вызывающие проблемы, и провести их оптимизацию или переделку в режиме модульного тестирования и профилирования.

При условии, что запрос написан правильно с точки зрения программирования на SQL, наиболее частые причины его медленного выполнения таковы:

- недостаточное оперативной памяти на сервере или недостаточное количество памяти выделено для СУБД;
- не хватает статистики. Проверьте опции, касающиеся автоматического сбора и автоматического создания статистики по колонкам таблиц, если необходимо, пересоздайте её в ручном режиме;
- не хватает полезных индексов; проверьте в планах выполнения запросов использование индексов и в случае обнаружения прямого сканирования таблиц (table scan, cluster index scan) или хэшированных соединений (hash join) попробуйте добавить новые индексы сообразно условиям соединений или фильтрации запросов;
- не хватает полезных индексированных видов. Некоторые СУБД поддерживают индексированные виды, представляющие собой материализованное представление — слепок данных результата

некоторого запроса. Если в другом запросе частично или полностью может быть использован такой материализованный вид, то общее время выполнения снизится;

- проблемы физической архитектуры. Проверьте производительность дискового массива на характерных операциях чтения/записи (здесь может оказаться полезной утилита iometer). Проверьте, используется ли массив монополюс сервером СУБД или же доступ разделяется с другими серверами, возможно, виртуальными;
- отсутствует секционирование данных. При больших объёмах таблиц и запросах, затрагивающих лишь её небольшие фрагменты, следует подумать о секционировании.

К сожалению, такая ситуация является не самой частой на практике. Гораздо чаще задержки отклика системы являются спонтанными, воспроизвести их в одиночном режиме невозможно, а испытатель или приёмщик со стороны клиента описывает ситуацию как «в общем, система тормозит».

Не следует отнимать у нас помощь таблиц мониторинга, хранящие статистические данные о выполнении запросов системы. Хорошая СУБД сама заботится о сборе информации, необходимой для диагностики своих проблем.

Общий список видов и табличных функций мониторинга в SQL Server называется Dynamic Management Views and Functions, по этим словам информацию можно найти в MSDN. Перечень содержит несколько десятков функций, поэтому ниже мы рассмотрим только несколько наиболее часто употребляемых в диагностике.

Если «система тормозит», то логично начать поиск с наиболее медленных и наиболее «прожорливых» запросов. Частично эти множества могут и пересекаться, но, в общем случае, медленный запрос может загружать только процессор, а «прожорливый», напротив, заставлять шуршать диски массива на полную катушку.

Для выявления кандидатов на оптимизацию нам понадобятся всего несколько запросов.

```
/* Статвыборка 1: запросы, использующие процессорное время */
SELECT TOP 100
    qt.text,
    qs.execution_count,
    qs.total_worker_time / 1000000 AS total_worker_time_sec,
    CASE qs.total_worker_time
        WHEN 0 THEN NULL
        ELSE round(
            (CAST(qs.execution_count AS float) /
             qs.total_worker_time * 1000000)
            , 3)
    END AS avg_queries_per_sec,
    qs.total_elapsed_time
FROM sys.dm_exec_query_stats qs
    CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
    CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
ORDER BY total_worker_time DESC
```

В первую очередь следует отметить и отбросить все подозрительные запросы, наиболее использующих процессорное время (`total_worker_time`). Одновременно следует учитывать и число запусков этого запроса (`execution_count`) и среднюю производительность за секунду (`avg_queries_per_sec`). Например, если некоторый запрос выполняется в либрах, но при этом он выполняется часто и его производительность не вызывает сомнений, то исключите его из списка подозреваемых.

Для прояснения картины та же статистическая выборка №1 может быть отсортирована по соответствующим колонкам или дополнительно отфильтрована, например «все запросы с производительностью ниже 100 в секунду и числом запусков не менее 1000».

Отсортировав ту же выборку по колонке `execution_count`, мы получим наиболее частые запросы. Здесь также следует обратить внимание на производительность.

Теперь настало время вытащить на свет наиболее «прожорливые» создания программистских рук. Прежде всего, замедления происходят при обращении к дисковой подсистеме.

```

/* Статвыборка 2: запросы, приводящие к обращениям к дискам
*/
SELECT TOP 100
    qt.text,
    qs.execution_count,
    qs.total_physical_reads,
    qs.total_physical_reads / qs.execution_count
        AS avg_physical_reads,
    qs.max_physical_reads,
    (qs.total_elapsed_time - qs.total_worker_time) / 1000
        AS total_non_cpu_time_msec
FROM sys.dm_exec_query_stats qs
    CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
    CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
ORDER BY total_physical_reads DESC

```

Общее количество физических чтений (`total_physical_reads`), распределенное по числу запусков (`execution_count`) надо сопоставлять с их максимумом (`max_physical_reads`). Если две величины, `avg_physical_reads` и `max_physical_reads`, примерно равны на многократных запусках (запрос регулярный), это может свидетельствовать о долговременной медленной памяти, являющейся в свою очередь, проблемой нехватки выделенного для СУБД объёма ОЗУ.

В PostgreSQL также имеется набор видов таблиц мониторинга. Вы можете найти их в документации в главе «Monitoring Database Activity. The Statistics Collector». Трассировка может осуществляться настройкой журналов уровня сервера, обратите внимание на соответствующую опцию `log_statement` в файле конфигурации `postgresql.conf`.

В Firebird, начиная с версии 2.5, были введены виды и таблицы мониторинга, имеющие префикс «MON\$». Кроме того, добавлена утилита командной строки для трассировки запросов. С помощью этих видов и функций вы сможете произвести аналогичный анализ.

Разумеется, некоторая часть запросов может просочиться через статистическое сито, но основную их массу удаётся отловить, и тогда оставшиеся начинают проявлять себя более предсказуемо и стабильно, сводя поиск к трассировке, описанной в первом случае.

Удачной охоты!

## Основы нагрузочного тестирования

Если в предыдущей главе рассматривались способы обнаружения узких мест в приложении, связанных с СУБД, то цель нагрузочного тестирования — выявить эти узкие места ещё на стадии системных тестов до передачи продукта заказчику на приёмку или в опытную эксплуатацию.

### Инструменты и методы

Инструментов, реализующих нагрузочное тестирование для разных СУБД создано много, в том числе открытых. Поиск в Интернет по ключевым словам «название\_СУБД load testing» или «stress testing» выдаст достаточно много ссылок. Ниже мы рассмотрим утилиту SQL Load Generator, ориентированную на SQL Server, но при этом максимально простую и бесплатную, с открытым исходным кодом.

Прежде чем приступить к тестированию, следует составить представление о типовых сценариях работы пользователей с системой. Из них будут вытекать основные типы используемых запросов, во-вторых, оценка числа активных соединений.

Для примера возьмём БД, используемую в тестах постраничной выборки (см. главу «Постраничные выборки»). Там понадобится генератор псевдослучайных значений в диапазоне (0..1), создадим его как вид.

```
CREATE VIEW dbo.rand2 AS
SELECT rand(abs(convert(int, convert(varbinary, newid()))))
AS rand_value;
```

Пусть в системе будет 10 пользователей, выполняющих запросы типа

```
SELECT *
FROM sales
WHERE id_customer = N AND id_product = M
```

Здесь N и M — произвольным образом выбранные целочисленные идентификаторы, соответственно, клиента и товара. Из условий теста постраничной выборки, при заполнении соответствующих таблиц использовались значения 10000 для количества клиентов и 1000 для количества товаров. С применением генератора псевдослучайных чисел запрос примет следующий вид.

```
SELECT *
FROM sales
WHERE id_customer = (SELECT CAST((10000 * rand_value) AS int)
FROM rand2) AND
      id_product = (SELECT CAST((1000 * rand_value) AS int)
FROM rand2)
```

Одновременно, ещё 10 пользователей выполняют запросы вида

```
UPDATE sales
SET qty = X
WHERE id_customer = N AND id_product = M
```

Здесь X — случайное значение количества товара в диапазоне (0..1000).

```
UPDATE sales
SET qty = (SELECT CAST((1000 * rand_value) AS int) FROM
rand2)
WHERE id_customer = (SELECT CAST((10000 * rand_value) AS int)
FROM rand2) AND
      id_product =(SELECT CAST((1000 * rand_value) AS int)
FROM rand2)
```

Определим в SQL-пра... соедин... с СУБД Microsoft SQL Server.

Теперь добавим новый запрос (кнопка Add Query) и заносим туда первый из подготовленных выше. Для параметра числа одновременных активных соединений определяем значение «10» согласно сценарию. Аналогично поступаем со вторым запросом на обновление.

Осталось запустить оба сценария на выполнение и подождать некоторое время, например, до выполнения первой тысячи-двух запросов на модификацию данных.

После остановки запросов смотрим статистику выполнения по уже известной методике.

```
SELECT TOP 100
qt.TEXT,
qs.execution_count,
qs.total_worker_time / 1000000 AS total_worker_time_sec,
CASE qs.total_worker_time
WHEN 0 THEN NULL
ELSE
CAST(qs.execution_count AS float) /
```

```

        qs.total_worker_time * 1000000
    END AS queries_per_sec
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
WHERE qt.text LIKE '%rand_value%'
ORDER BY execution_count DESC

```

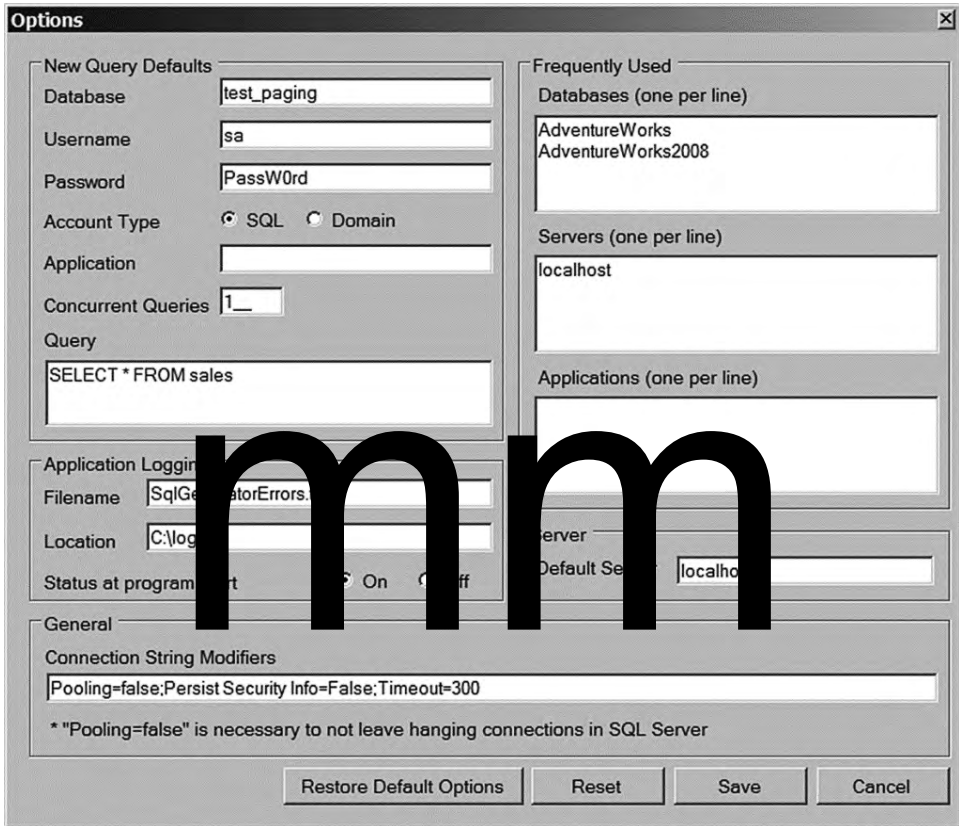


Рис.65. Установка опций соединения

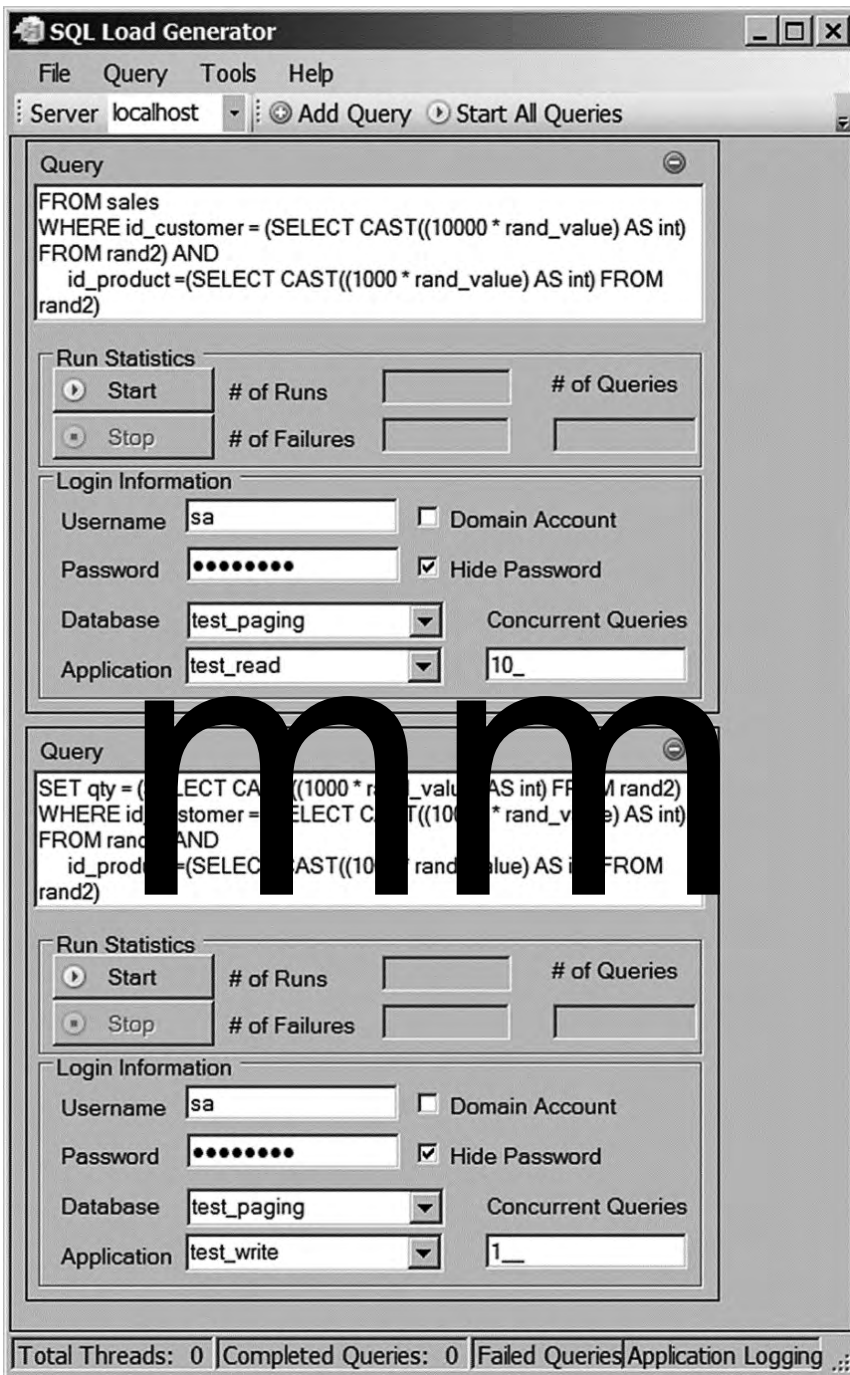


Рис.66. Создание тестовых запросов

Для запросов на чтение и запись получаем следующие показатели.

| Запрос             | Выполнено, раз | Потрачено времени, сек | Транзакций в секунду |
|--------------------|----------------|------------------------|----------------------|
| Чтение (запрос №1) | 8767           | 8                      | 1083                 |
| Запись (запрос №2) | 4286           | 7                      | 548                  |

Если полученные значения по количеству транзакций в секунду вас устраивают, то данный сценарий использования системы можно считать проверенным. Вас можно поздравить.

А если нет? Тогда вас тоже можно поздравить: узкое место выявлено на раннем этапе разработки, когда стоимость его ликвидации ещё относительно невысока. По крайней мере, заказчик не столкнётся с этой проблемой на собственной шкуре.

## Учёт степени параллелизма

В не то чтобы старые, но всё же достаточно давно времена один физический процессор на рабочем месте программиста или на сервере имел одно ядро. Поэтому данный вопрос не являлся таким значимым, как сейчас, когда количество ядер на одном «машине» позволяет СУБД распараллеливать выполнение запросов уже на самых первых этапах процесса разработки программ.

Допустим, вы отлаживаете SQL-запрос, получив на рабочем месте или на сервере разработки удовлетворительное время. Но попав на тестовый сервер да ещё и под нагрузку ваш запрос начинает показывать на порядок худшее время. В чем проблема?

При отладке запроса СУБД, как правило, позволяет выводить сопутствующую статистическую информацию и план его выполнения. Одним из ключевых показателей является время, затраченное непосредственно процессором. В SQL Server такая информация выводится в виде «CPU time = 123 ms», показывая, сколько времени запрос выполнялся процессором. В других СУБД также имеются возможности по выводу аналогичной статистики (EXPLAIN в PostgreSQL, SET PLAN в Firebird и т. д.).

Положим, SQL-запрос выводит общее количество заказов, количество товаров в которых превышает среднее количество во всех заказах.

```
/* Очистка кеша процедур и данных */
DBCC FREEPROCCACHE;
DBCC DROPCLEANBUFFERS;
/* Вывод статистики */
SET STATISTICS TIME ON;
SET STATISTICS IO ON;
/* Тестовый запрос */
SELECT COUNT(1)
FROM
  (SELECT id_product, AVG(qty) AS qty
   FROM sales
   GROUP BY id_product
  ) t1
WHERE t1.qty > (
  SELECT AVG(qty)
  FROM (SELECT id_product, AVG(qty) AS qty
        FROM sales
        GROUP BY id product) t2)
```

При очищении буфера и включённых опций SET STATISTICS TIME и SET STATISTICS IO сопряжённые с задачей результаты сообщения на MS SQL Server будут выглядеть следующим образом:

```
(1 row(s) affected)
Table 'sales'. Scan count 2, logical reads 106820, physical
reads 12, read-ahead reads 50383, lob logical reads 0, lob
physical reads 0, lob read-ahead reads 0.
```

```
SQL Server Execution Times:
  CPU time = 32627 ms, elapsed time = 47882 ms.
```

Запустим запрос ещё раз, но без очистки кеша, т. е. в «горячем» режиме.

```
(1 row(s) affected)
Table 'sales'. Scan count 2, logical reads 106820, physical
reads 0, read-ahead reads 0, lob logical reads 0, lob
physical reads 0, lob read-ahead reads 0.
```

```
SQL Server Execution Times:
  CPU time = 30323 ms, elapsed time = 35714 ms.
```

Статистика хорошо показывает, что запуск в холодном и горячем режимах ожидаемо влияет на общее время выполнения запроса (elapsed time), но в

гораздо меньшей степени — на процессорное время (CPU time). Это объясняется наличием большого числа операций физического и упреждающего чтения, сопровождающегося обменом с долговременной памятью (дисками).

Зная, например, что данный пример был выполнен на виртуальной машине с единственным одноядерным процессором, можно взять полученный результат за основу и предполагать, что с увеличением числа задействованных ядер и процессоров эта составляющая времени выполнения запроса будет пропорционально снижаться.

С той же целью для более точной оценки можно задействовать в запросе специфичные для СУБД опции, указывающие максимальную степень параллелизма, установив это значение в единицу. Например, в SQL Server этому служит опция MAXDOP.

```
/* запрос будет выполняться на одном процессоре/ядре */  
SELECT id_product, AVG(qty) AS qty  
FROM sales  
GROUP BY id_product  
OPTION(MAXDOP 1)
```

Верно и обратное. Положим, на нашем однопроцессорном четырёхядерном рабочем месте, где установлен локальный сервер СУБД, некоторый SQL-запрос выполняется за 500 миллисекунд, в том числе 300 миллисекунд занимает процессорное время. Значит на одном ядре запрос выполнялся бы примерно 1200 миллисекунд, в четыре раза дольше. Если на тестовом сервере установлены два процессора с четырьмя ядрами, то под имитацией нагрузки 15-20-ти пользователей ваш запрос начнёт выполняться в районе 2,5-3 секунд и выше. Время выросло почти на порядок!

Таким образом, при оценке соответствия производительности запросов требуемому времени отклика в режиме модульного тестирования и профилирования следует принять во внимание особенности конфигурации СУБД-серверов соответствующей среды, прежде всего, количество физических или виртуальных процессоров и их ядер.

## SQL Server и MongoDB на простом тесте

Описанное ниже испытание было проведено в рамках одного из предпроектных обследований. Хотя выбор SQL Server был фактически predetermined, целью тестирования было скорее понять, какие преимущества и проблемы могло бы принести использование в системе MongoDB.

Задача, поставленная в условиях теста, не является сильной стороной документ-ориентированных моделей, поскольку атрибутика данных достаточно жёсткая. Тем не менее, испытание показывает основные отличия при работе с СУБД разных моделей.

На сайте MongoDB была загружена последняя стабильная версия 2.0.2 для 64-разрядной Windows. В качестве альтернативы MongoDB выступал MS SQL Server 2008 R2 Developer Edition, тоже 64-разрядный. Если у вас такого нет, подойдёт и бесплатная редакция SQL Server Express 2005 и выше, которую также можно загрузить с сайта Microsoft. Компьютер для обоих тестов использовался относительно слабый, уровня обычной рабочей станции, под управлением ОС Windows 7, два ядра Intel 2,6 ГГц с одним достаточным «экономичным» диском 5400 оборотов/мин, 300 Гб, и шестью гигабайтами памяти. Поскольку целью является сравнение некоторых характеристик на одной и той же конфигурации, то аппаратная мощность компьютера не имеет значения.

### Тест вставки записей

Тест интенсивной вставки данных от множества датчиков ограничен 10 миллионами записей. Реальные объёмы на порядки выше, но и такое количество более чем достаточно для выявления потенциальных проблем.

Данные вставляются в коллекцию MongoDB и таблицу SQL Server, соответственно, имеющие одинаковую структуру.

К сожалению, MongoDB не поддерживает транзакционность на уровне группы операторов. Но в данном случае это должно дать выигрыш в производительности.

```
function pad(n) { return n < 10 ? '0' + n : n };  
function ISODateString(d) {
```

```

return d.getUTCFullYear() + '-'
    + pad(d.getUTCMonth() + 1) + '-'
    + pad(d.getUTCDate()) + 'T'
    + pad(d.getUTCHours()) + ':'
    + pad(d.getUTCMinutes()) + ':'
    + pad(d.getUTCSeconds());
};

print("Starting fill database: " + Date());
maxLines = 1000;
progressStep = maxLines / 100;
devicesCount = maxLines / 5;
for (var i = 1; i <= maxLines; i++) {
    db.measuresData.insert(
        {
            "measureId" : i,
            "deviceId" : Math.floor(Math.random() * devicesCount +
1),
            "stateId" : Math.floor(Math.random() * 2 + 1),
            "groupId" : Math.floor(Math.random() * 100 + 1),
            "measureDate": new Date(),
            "intVal" : Math.floor(Math.random() * 1000000 -
1000000),
            "floatVal" : Math.random() * 200000 - 100000
        });
    if (i % progressStep == 0) {
        print(i + " " + DateString(new Date()));
    }
};
print("Finished at: " + Date());

```

Для SQL-скрипта из главы «Загрузка данных» вы уже знаете, что приложение должно сначала накопить массив строк, а потом вставить их в таблицу в контексте одной транзакции. В тесте выбран размер пакета из 10 записей. Не забудьте изменить пути физического размещения файлов в соответствии с вашей конфигурацией.

```

SET NOCOUNT ON;

CREATE DATABASE testvol
ON PRIMARY (
    NAME = N'testvol',
    FILENAME = N'D:\MSSQL\DATA\testvol.mdf',
    SIZE = 200MB,
    MAXSIZE = UNLIMITED,

```

```

FILEGROWTH = 200MB
)
LOG ON (
    NAME = N'testvol_log',
    FILENAME = N'D:\MSSQL\DATA\testvol_log.ldf',
    SIZE = 100MB,
    MAXSIZE = UNLIMITED,
    FILEGROWTH = 100MB
)
GO

ALTER DATABASE [testvol] SET RECOVERY SIMPLE
GO

USE testvol
GO
CREATE TABLE dbo.measuresData (
    measureId    int NOT NULL PRIMARY KEY CLUSTERED,
    deviceId     int NOT NULL,
    stateId      int NOT NULL,
    groupId      int NOT NULL,
    measureDate  datetime NOT NULL,
    intVal       int NOT NULL,
    floatVal     float NOT NULL
)
GO

PRINT 'Starting fill database: ' + convert(nvarchar(32),
getdate(), 120);
DECLARE @maxLines int = 10000000;
DECLARE @progressStep int = @maxLines / 100;
DECLARE @devicesCount int = @maxLines / 5;
DECLARE @batchSize int = 10;
DECLARE @i int = 1;
BEGIN TRANSACTION;
WHILE @i <= @maxLines BEGIN
    INSERT INTO dbo.measuresData (
        measureId,
        deviceId,
        stateId,
        groupId,
        measureDate,
        intVal,
        floatVal
    )

```

```

VALUES (
    @i,
    floor(rand() * @devicesCount + 1),
    floor(rand() * 2 + 1),
    floor(rand() * 100 + 1),
    getdate(),
    floor(rand() * 2000000 - 1000000),
    rand() * 2000000 - 1000000
);

if @i % @progressStep = 0 BEGIN
    PRINT convert(nvarchar(16), @i) + ';' +
           convert(nvarchar(32), getdate(), 120);
END
if @i % @batchSize = 0 BEGIN
    COMMIT TRANSACTION;
    BEGIN TRANSACTION;
END
SET @i = @i + 1;
END
COMMIT TRANSACTION;
PRINT 'Finished' + convert(nvarchar(16), @i) + ';' +
      convert(nvarchar(32), getdate(),
120);
GO

```

По результатам теста получилось следующий график, показывающий, что несмотря на отсутствие транзакций, вставка пакетом по 1000 записей в таблицу SQL Server производительнее вставки соответствующих документов в коллекцию MongoDB.

Оценка размера базы данных также говорит в пользу SQL Server. Коллекция из 10 миллионов документов заняла 3,95 гигабайт, тогда как база данных SQL Server всего 0,5 гигабайт (без использования компрессии), то есть почти в 8 раз меньше. Избыточность требуемого размера хранения объясняется внутренним форматом JSON, используемом в MongoDB.

В отношении потребляемой оперативной памяти, MongoDB «отъела» практически всю свободную, заняв 4 гигабайта против всего 1 гигабайта у SQL Server при выставленном ему лимите в 3 гигабайта.

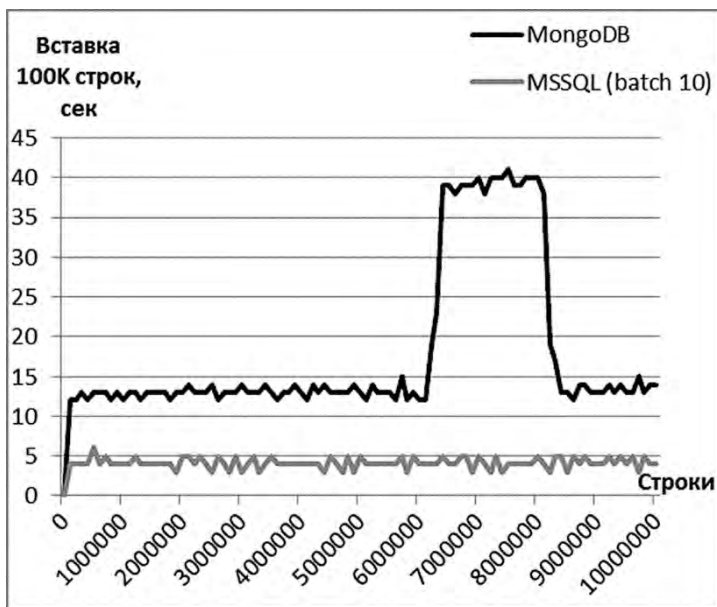


Рис.67. Время вставки 100 тысяч записей (пачками по 10) и документов

К сожалению, ограничить объем используемой оперативной памяти у MongoDB на данном этапе её развития было нельзя. Запрос на изменения висел больше года на сайте, не встретив у разработчиков понимания. В общем случае решение состоит в создании виртуального сервера для СУБД, которой отдаётся вся имеющаяся память.

## Запросы и хронометраж

Запросы в CRUD-логике тестировать не имело большого смысла, потому что сразу после вставки таблица должна была служить для отчётности и оперативной аналитики. То есть, запросы предполагаются ещё не настоящие тяжёлые аналитические, но они могут вызывать сканирование всей таблицы.

Для тестов приходится делать перезапуск процесса `mongod` для очистки памяти. В SQL Server для аналогичного эффекта просто очищаем буфера и кэш соответствующими командами DBCC.

Первым впечатлением от языковых средств MongoDB было ощущение возврата куда-то в начало 1990-х годов: навигационный подход в стиле программ на Clipper (Harbour), с его бесконечными обходами таблиц в

циклах, явным выбором текущего индекса для поиска, наложениями фильтров, ручным суммированием и прочими давно забытыми в SQL-среде особенностями.

В MongoDB отсутствуют встроенные функции агрегации. Для нахождения сумм или средних величин нужно фактически написать свой аналог стандартных в SQL функций SUM() или AVG(), только на языке JavaScript, или использовать более общий метод MapReduce.

Справедливости ради, надо сказать, что начиная с версии SQL Server 2005, программисты могут писать свои нестандартные функции агрегации на C#. Но я посоветовал бы вначале хорошо подумать о необходимости этого шага.

В случае, если запрос с агрегацией возвращает более 10 тысяч документов, использование MapReduce становится обязательным, иначе система выдаёт ошибку. Таким образом, из встроенных функций остаётся только подсчёт числа элементов, в том числе уникальных. Но использовать подсчёт можно только в применении к целым коллекциям, то есть вне контекста запросов с группировкой. Поэтому в общем случае программист должен писать рутинный типовой код на JavaScript и для функции агрегации COUNT().

Оказалось невозможным и отсортировать результаты запроса с группировкой. Разработчики на форуме поддержки предложили выбрать результат в приложение и сделайте сортировку в нём.

*Q: How can I sort on the resultset of the group by operation?*

*A: Group currently just returns an object, so you should be able to sort easily client side*

В результате код запросов с кратким и ясным синтаксисом декларативного языка SQL превращается в длинную «простыню» из императивных инструкций и параметров-деклараций. Разницу можно проследить на последующих примерах.

Запрос №1. Поиск минимального и максимального значения дат в таблице/коллекции.

## SQL Server

```
SELECT MIN(measureDate), MAX(measureDate)
FROM dbo.measuresData
```

## MongoDB

```
var minDate = new Date(1900,1,1,0,0,0,0);
var maxDate = new Date(2100,1,1,0,0,0,0);
db.measuresData.group(
{
  key: { },
  reduce: function(obj, prev)
  {
    if (obj.measureDate.getTime() < prev.minMsec)
      prev.minMsec = obj.measureDate.getTime();
    else if (obj.measureDate.getTime() > prev.maxMsec)
      prev.maxMsec = obj.measureDate.getTime();
  },
  initial: { minMsec: maxDate.getTime(), maxMsec:
minDate.getTime() },
  finalize: function(out)
  {
    out.minMeasureDate = new Date(out.minMsec);
    out.maxMeasureDate = new Date(out.maxMsec);
  }
})
.forEach(printjson);
```

Вариант MongoDB использует представление дат в виде числа миллисекунд, отсчитываемых от 01/01/1970, поэтому прямое их сравнение в функции вызывало ошибку `TypeError: this["get" + UTC + "FullYear"] is not a function nofile_b:2`.

Опыт работы с навигационными СУБД пригодился и в MongoDB. Если построить индекс по элементу `measureDate`, то можно найти крайние значения относительно простым способом: отсортировать по индексу в порядке возрастания и взять первый элемент, повторив процедуру в порядке убывания значений.

```
// построение индекса
db.measuresData.ensureIndex({measureDate: 1});
// первый элемент
db.measuresData.find({}, {measureDate: 1})
.sort({measureDate: 1}).limit(1);
```

```
// последний элемент
db.measuresData.find({}, {measureDate: 1})
.sort({measureDate: -1}).limit(1);
```

Запрос №2. Подсчёт сумм целых и вещественных значений по состоянию и группе устройств. Запрос для MongoDB не выполняет сортировку.

### SQL Server

```
SELECT SUM(intVal), SUM(floatVal), stateId, groupId
FROM dbo.measuresData
GROUP BY stateId, groupId
ORDER BY stateId, groupId
```

### MongoDB

```
db.measuresData.group(
{
  key: { stateId: true, groupId: true },
  reduce: function(obj, prev)
  {
    prev.sumIntVal += obj.intVal;
    prev.sumFloatVal += obj.floatVal;
  },
  initial: { sumIntVal: 0, sumFloatVal: .0 }
})
.forEach(printJson);
```

Запрос №3. Подсчёт общего числа устройств и количества уникальных устройств по состоянию и их группе.

### SQL Server

```
SELECT COUNT(deviceId), COUNT(DISTINCT deviceId), stateId,
groupId
FROM dbo.measuresData
GROUP BY stateId, groupId
ORDER BY stateId, groupId
```

### MongoDB (не выполняет сортировку)

```
db.measuresData.group(
{
  key: { stateId: true, groupId: true },
  reduce: function(obj, prev)
  {
    prev.count++;
  },
  initial: { count: 0 },
```

```

finalize: function(out)
{
    out.distCount = db.measuresData
        .distinct("deviceId", {stateId: out.stateId, groupId:
out.groupId})
        .length;
}
})
.forEach(printjson);

```

«Бортовой» журнал СУБД показывает примерно 35 секунд для функции `finalize` на каждый цикл вычисления `distinct` из примерно 200 строк. Такая производительность совершенно неприемлема, запрос будет крутиться в течение почти 2 часов. Прерываем выполнение и приступаем к оптимизации. Необходимо построить индекс по элементам `stateId` и `groupId`.

```
db.measuresData.ensureIndex({stateId: 1, groupId: 1})
```

Индекс строился 141 секунду, тогда как аналогичный в SQL Server — за 20 секунд. Размер файла вырос до 4,2 МБ.

Перезапускаем запрос. Цикл вычисления `distinct` снижается до 300 миллисекунд, в 100 раз! В третьем перезапуске время цикла выросло до 1,2 секунды, потом начало снижаться до 100 миллисекунд. Общее время — более 15 минут! Возникают подозрения, что СУБД практически не использует кэш данных.

Запрос №4. Подсчёт сумм целых и вещественных значений по состоянию и группе устройств для заданного диапазона дат. В качестве диапазона выбрана одна минута примерно в середине интервала между минимальным и максимальным значениями дат. Так как распределение тестовых данных близко к равномерному, каждый диапазон включает в себя около 500 тысяч строк.

### SQL Server

```

SELECT SUM(intVal), SUM(floatVal), stateId, groupId
FROM dbo.measuresData
WHERE measureDate BETWEEN '20120208 22:54' AND '20120208
22:55'
GROUP BY stateId, groupId

```

```
ORDER BY stateId, groupId
```

### MongoDB (не выполняет сортировку)

```
var date1 = new Date(ISODate("2012-02-08T15:20:00.000Z"));
var date2 = new Date(ISODate("2012-02-08T15:21:00.000Z"));

db.measuresData.group(
{
  key: { stateId: true, groupId: true },
  cond: {measureDate: { $gte: date1, $lt: date2 } },
  reduce: function(obj, prev)
  {
    prev.sumIntVal += obj.intVal;
    prev.sumFloatVal += obj.floatVal;
  },
  initial: { sumIntVal: 0, sumFloatVal: 0.0 }
})
.forEach(printjson)
```

Запрос №5. Подсчёт общего числа устройств и количества уникальных устройств по состоянию и их группе для того же заданного диапазона дат.

### SQL Server

```
SELECT COUNT(deviceId), COUNT(DISTINCT deviceId, stateId,
groupId)
FROM dbo.measuresData
WHERE measureDate BETWEEN '20120208 22:54' AND '20120208
22:55'
GROUP BY stateId, groupId
ORDER BY stateId, groupId
```

### MongoDB (не выполняет сортировку)

```
var date1 = new Date(ISODate("2012-02-08T15:20:00.000Z"));
var date2 = new Date(ISODate("2012-02-08T15:21:00.000Z"));

db.measuresData.group(
{
  key: { stateId: true, groupId: true },
  cond: {measureDate: { $gte: date1, $lt: date2 } },
  reduce: function(obj, prev)
  {
    prev.count++;
  },
  initial: { count: 0 },
  finalize: function(out)
```

```

{
  out.distCount = db.measuresData
    .distinct("deviceId", {stateId: out.stateId, groupId:
out.groupId})
    .length;
}
})
.forEach(printjson);

```

Проведённый хронометраж осуществлялся по каждому запросу в три последовательных запуска, первый из них — в «холодном» режиме.

Для MongoDB результаты без оптимизации выглядят неприемлемо долгими. После оптимизации, где она возможна, результаты можно оценить, как неудовлетворительные.

Со стороны SQL Server оптимизация сводится к построению кластерного индекса по колонке measureDate. Соответственно, имеющийся первичный ключу по measureId объявляется некластерным. В запросах №2 и №3 необходимо полное сканирование данных, поэтому оптимизация индексацией для SQL Server не производилась.

Первый запуск — так называемый «холодный», вторые и третьи запуски — «горячие», они короче, полностью игнорируя работу кэша СУБД.

Табл. 23. Хронометраж запросов

| Запуски                       | SQL Server       |    |    | MongoDB |     |     |
|-------------------------------|------------------|----|----|---------|-----|-----|
|                               | 1                | 2  | 3  | 1       | 2   | 3   |
| Без оптимизации, время, сек   |                  |    |    |         |     |     |
| Запрос №1                     | 7                | 1  | 1  | 190     | 185 | 189 |
| Запрос №2                     | 8                | 3  | 3  | >7200   |     |     |
| Запрос №3                     | 26               | 20 | 20 | 345     | 341 | 349 |
| Запрос №4                     | 8                | 1  | 1  | 22      | 22  | 22  |
| Запрос №5                     | 15               | 10 | 12 | 141     | 141 | 141 |
| После оптимизации, время, сек |                  |    |    |         |     |     |
| Запрос №1                     | 0                | 0  | 0  | 0       | 0   | 0   |
| Запрос №2                     | 8                | 3  | 3  | 322     | 800 | 619 |
| Запрос №3                     | не производилась |    |    |         |     |     |
| Запрос №4                     | 1                | 1  | 1  | 12      | 12  | 13  |
| Запрос №5                     | 7                | 7  | 7  | 85      | 84  | 85  |

## Выводы

Несмотря на не вполне подходящие условия задачи для использования NoSQL-СУБД, можно сделать некоторые выводы в отношении использования. С другой стороны, делать скидку на «не вполне подходящие условия», значит прямо признать, что СУБД является специализированной и имеет узкий сегмент применения.

Из необходимости построения индекса в случае запроса №2 следует, что без оптимизации, требующего вмешательства администратора БД, время отклика простого ад-хок-запроса становится неприемлемым. С точки зрения пользователя это означает, что «запрос не работает».

Недостатки:

- сложность программирования запросов, являющихся обычными в языке SQL;
- неудовлетворительное время отклика при выполнении запросов;
- избыточность физического хранения;
- проблемы с ограничением использования оперативной памяти;
- неэффективное использование кластера данных и запросов.

Преимущества:

- гибкая структура данных, не требующая предварительного определения схемы.

Перечисленное даёт возможность утверждать, что выбор подобного решения должен быть обоснован и подтверждён сравнительными тестами с универсальными РСУБД, в том числе с использованием их встроенной поддержки неполно структурированных данных.

## **Тестовые и демонстрационные базы данных**

Многие поставщики СУБД собственными силами и при помощи сообщества пользователей предоставляют для испытаний и ознакомления так называемые тестовые и демонстрационные базы данных.

В некоторых случаях, например, Firebird, небольшая демонстрационная база входит в дистрибутив установки.

Для Microsoft SQL Server в соответствующем разделе сайта сообщества разработчиков (<http://msftdbprodsamples.codeplex.com>) можно найти реляционные демобазы для транзакционных и аналитических приложений, а также многомерные базы для Analysis Services.

На странице документации сообщества пользователей PostgreSQL ([http://wiki.postgresql.org/wiki/Sample\\_Databases](http://wiki.postgresql.org/wiki/Sample_Databases)) приводится множество ссылок на базы данных, которые можно загрузить и установить для проведения тестов.

Разработчики Oracle также предоставляют всем интересующимся примеры не только собственно баз данных, но и программную реализацию некоторых техник работы с продуктом, например, использования API для добычи данных (Data Mining). Подробнее см. [http://docs.oracle.com/cd/E11882\\_01/install.112/e24501/toc.htm](http://docs.oracle.com/cd/E11882_01/install.112/e24501/toc.htm).

**mm**

## Заключение

Теперь, отложив книгу в сторону, можно немного поразмыслить. Насколько целостной была подаваемая информация? Удалось ли узнать что-то новое? Остались ли за рамками издания вопросы, которые хотелось бы рассмотреть? Станет ли чтение толстых монографий из списка литературы более осмысленным? Ваши отзывы могут быть учтены в новом дополненном переиздании.

Одной из целей книги было уменьшение числа ситуаций, когда программистам требуется помощь специалистов по СУБД. Потому что работа эксперта должна, в идеале, быть направлена на совместный синтез проектных решений, а не на ликвидацию последствий реализации непродуманных подходов, когда, зачастую, наилучшим выходом является уже полная переделка частей системы.

Я не опасаясь, что, приумножив собственные знания в области СУБД, вы станете обходиться без помощи со стороны других специалистов, но искренне надеюсь, что будете вовлекать их в проект на более ранних этапах, предотвращая грядущие проблемы.

*Ipsa scientia potestas est. Knowledge itself is power.* Франсис Бэкон.

*Знание — сила.* Фрэнсис Бэкон.

# Литература

1. Дейт К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.: ил.
2. Карпова Т. С. Базы данных: модели, разработка, реализация. — СПб.: Питер, 2002. — 304 с., ил.
3. Грабер М. Введение в SQL. — М.: Лори, 1996.
4. Ульман Дж. Основы систем баз данных. — М.: Финансы и статистика, 1983. — 334 с, ил.
5. Мартин Дж. Организация баз данных в вычислительных системах. — М.: Мир, 1978.— 616 с.
6. Пржиялковский В. В. Абстракции в проектировании баз данных — Журнал «СУБД» №1-2/1998
7. Когаловский М. Р. Абстракции и модели в системах баз данных. — Журнал «СУБД» №4-5/1998
8. Чен, Петер Пин-Шен. Модель «сущность-связь» - шаг к единому представлению данных. Журнал «СУБД» №1-2/1995.
9. Хендерсон К. Профессиональное руководство по SQL Server: хранимые процедуры; пер. с англ. — СПб.: Питер, 2005.
10. Хендерсон К. Профессиональное руководство по SQL Server: структура и реализация; пер. с англ. — М.: Издательский дом «Вильямс», 2006.
11. Мирошниченко Г. А. Реляционные базы данных: практические приёмы оптимальных решений. — СПб.: БХВ-Петербург, 2005. — 400 с.: ил.
12. Чёрч А. Введение в математическую логику.: Пер. с англ. — Т. 1. — М., 1960.
13. Тарасов С. Дефрагментация мозга. Софтостроение изнутри.— СПб.: Питер, 2013.
14. Усов А. Ключ или отмычка?—2001 г., статья на веб-сайте: <http://alexus.ru/russian/articles/dbms/keys/index.htm>
15. Тенцер А. Естественные ключи против искусственных ключей. — 1999 г., статья на веб-сайте: <http://ibase.ru/devinfo/NaturalKeysVersusArtificialKeysByTentser.html>

16. Емельянов Н. Е. Введение в СУБД ИНЕС. — М.:Наука, 1988, 256 с. (Библиотечка программиста).
17. Кодд Э. Ф. Реляционная модель данных для больших совместно используемых банков данных. Перевод Communications of the ACM, Volume 13, Number 6, June, 1970.— Журнал «СУБД», №1/1995.
18. Stonebraker, Michael. Readings in Database Systems (2nd edition). San Mateo, Calif.: Morgan Kaufmann, 1994. Introduction to Chapter 1.
19. Codd, E.F. Is Your DBMS Really Relational?; Computerworld, October 14th, 1985.
20. Codd, E.F. Does Your DBMS Run By The Rules?; Computerworld, October 21st, 1985.
21. Codd, E.F. The Relational Model For Database Management Version 2. Reading, Mass.: Addison-Wesley, 1990.
22. ИНТУИТ. Модели и смыслы данных в Cache и Oracle. Лекция на веб-сайте дистанционного обучения (<http://www.intuit.ru>).
23. Raima Database Manager 11.0. SQL Language Guide. Официальное руководство (английский язык) на сайте производителя СУБД (<http://www.raima.com>).
24. Д. Кнут. Искусство программирования (в 3 томах). Издание 2. Пер. с англ. М.: Вильямс, 2001.
25. Н. Вирт. Алгоритмы + Структуры данных = Программы. М.: Мир, 1985.
26. Joe Celko, Trees and Hierarchies. Morgan-Kaufmann, 2004.
27. Гамма Э. и др. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. Пер. с англ. СПб.: Питер, 2006.
28. Фаулер М. и др. Архитектура корпоративных программных приложений. Пер с англ. М.: Вильямс, 2006.
29. Тарасов С. Разработка ядра информационной системы. Часть 1. Журнал «Мир ПК» №7 2007 г.
30. Тарасов С. В., Бураков В. В. Способы реляционного моделирования иерархических структур данных. Журнал «Информационно-управляющие системы» №6 2013 г.

Тарасов Сергей Витальевич

# СУБД для программиста

## Базы данных изнутри

Ответственный за выпуск: В. Митин

Верстка: СОЛОН-Пресс

Обложка: СОЛОН-Пресс

*ООО «СОЛОН-Пресс»*

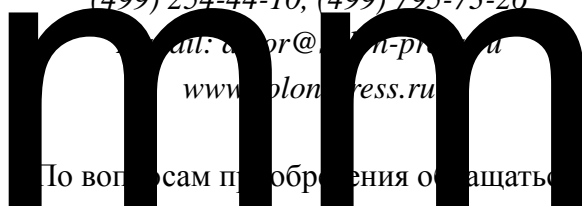
*123001, г. Москва, а/я 82*

*Телефоны:*

*(499) 254-44-10, (499) 795-73-26*

*e-mail: solon@solon-press.ru*

*www.solon-press.ru*



По вопросам приобретения обращаться

**ООО «ПЛАНЕТА АЛЬЯНС»**

Тел: (499) 782-38-89, [www.aliants-kniga.ru](http://www.aliants-kniga.ru)

По вопросам подписки на журнал «Ремонт & Сервис» обращаться:

**ООО «СОЛОН-Пресс»**

тел.: (499) 795-73-26, e-mail: [rem\\_serv@solon-press.ru](mailto:rem_serv@solon-press.ru)

[www.remserv.ru](http://www.remserv.ru)

**ООО «СОЛОН-Пресс»**

115142, г. Москва, Кавказский бульвар, д. 50

Формат 70×100/16. Объем 20 п. л. Тираж 200 экз.

Заказ №