Дизайн для Производительности

PASS SQL saturday

#178 | MOSCOW 2012

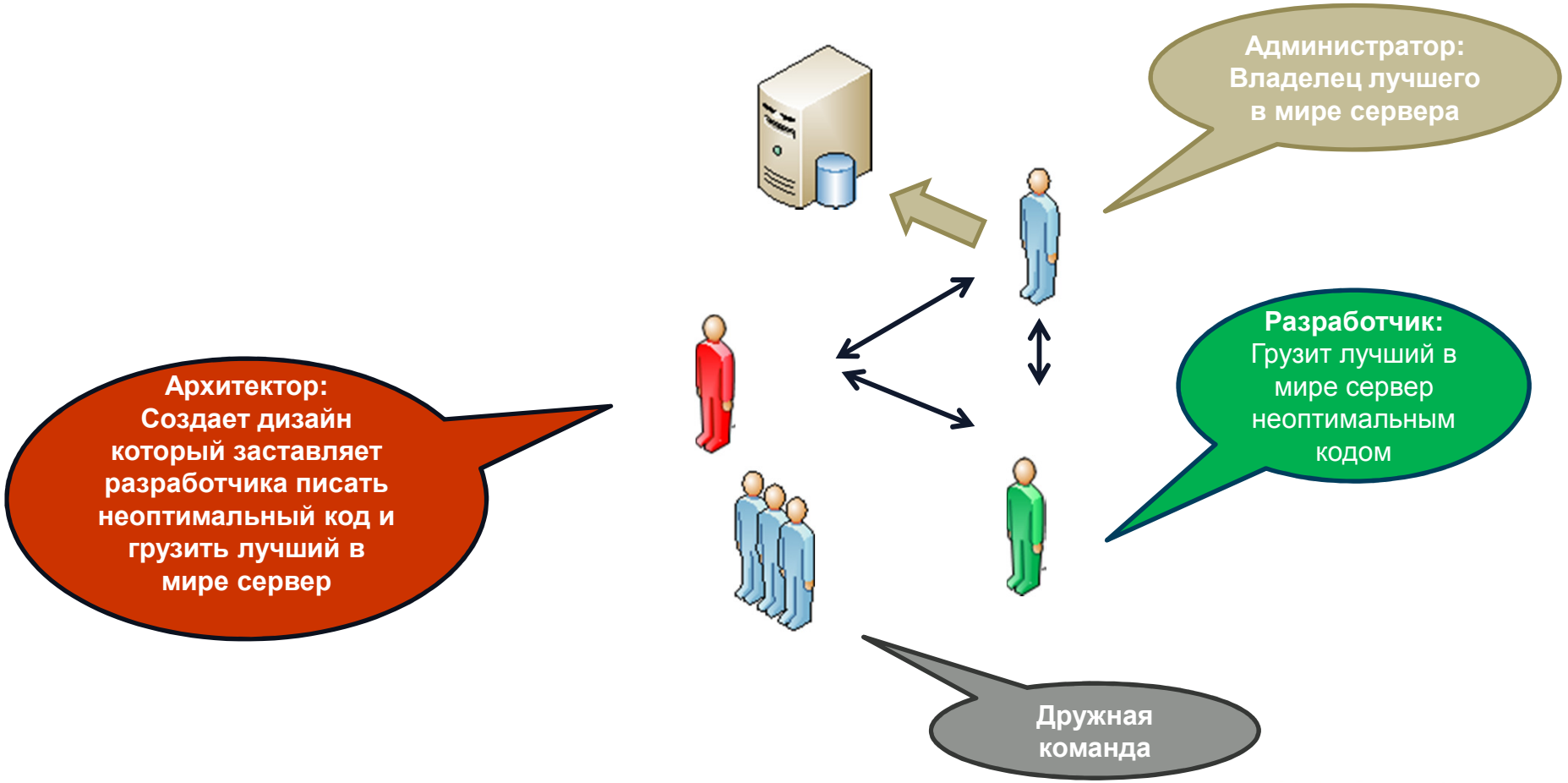
Обо мне

- 15+ лет в IT, включая 10+ лет работы с SQL Server
- Microsoft SQL Server MVP
- Microsoft Certified IT Professional
  - Database Developer
  - Database Administrator
- Microsoft Certified Professional Developer
  - Enterprise Application Developer

- Blog: http://aboutsqlserver.com
- Email: dmitri@aboutsqlserver.com

PASS SQL saturday

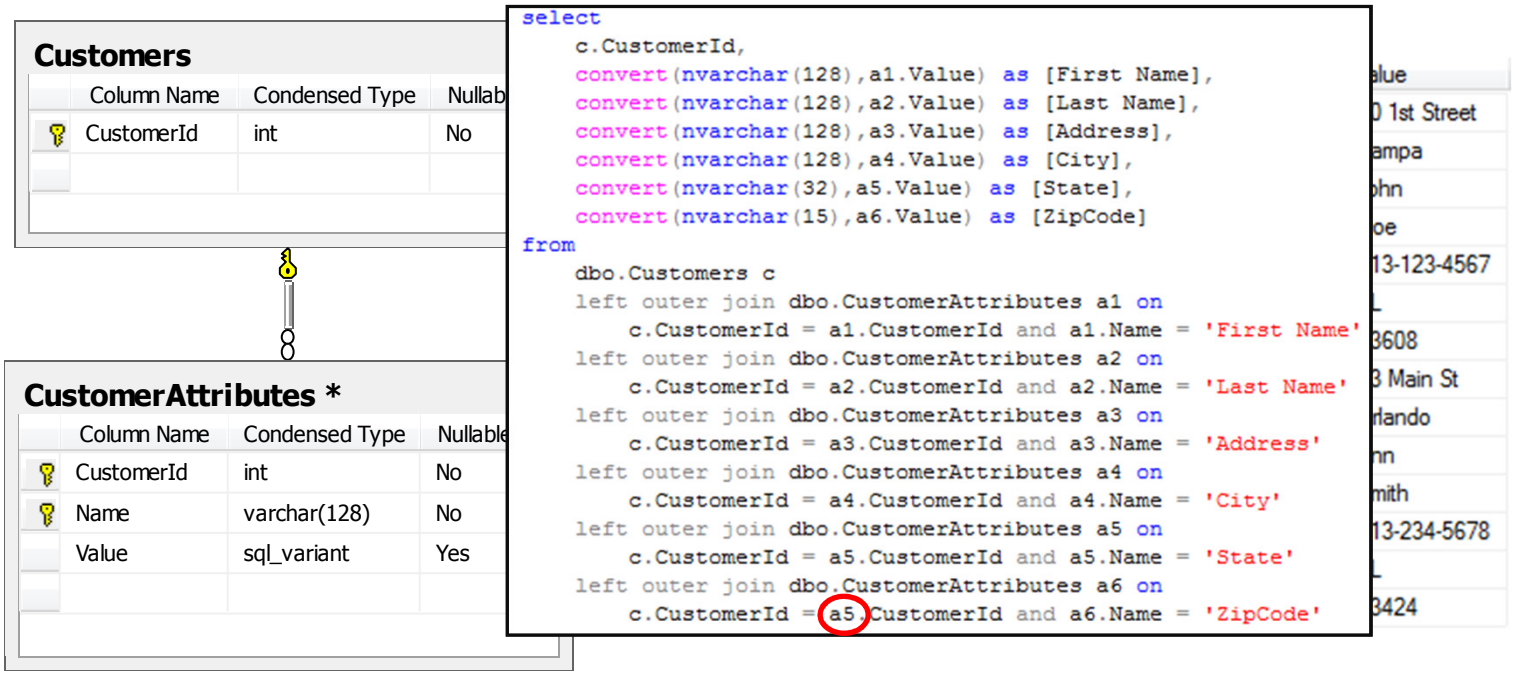
О чем эта сессия



Что такое «Плохой Дизайн»

Дизайн, созданный другим разработчиком ☺

Дизайн, который не принимает во внимание специфику и ограничения платформы



О чем мы будем говорить

- Как SQL Server хранит данные
- Типы объектов и их влияние на производительность
- Немного об архитектуре систем в целом

И перед тем как мы начнем

- Мы обсуждаем SQL Server 2005/2008/2008R2/2012
  - *Кое-что* справедливо для SQL Server 2000, Oracle, DB2, MySql…
- Мы обсуждаем OLTP системы
  - Data Warehouse / Reporting системы играют по несколько иным правилам
- Каждая система уникальна и может требовать специфических решений

Как SQL Server хранит данные (С высоты птичьего полета)

- Все объекты хранятся на страницах размером 8K
- Типы с фиксированной длиной
  - Tinyint (1 byte), Int (4 bytes), Char(N), …
  - Размер на диске не зависит от значения (включая NULL)
- Типы с переменной длиной
  - Varchar(N), Varbinary(N), …
  - Размер на диске зависит от объема данных
    - NOT NULL:Объем данных + 2 байта
    - NULL: (за редким исключением) 2 байта
- SQL Server 2008+ Enterprise может использовать компрессию

I/O – Наиболее важный фактор

- I/O является одним из наиболее важных факторов влияющих на производительность системы
  - Страницы данных должны быть загружены в память при чтении и записи
  - Физический доступ к диску является одной из самых дорогих операций

Демонстрация

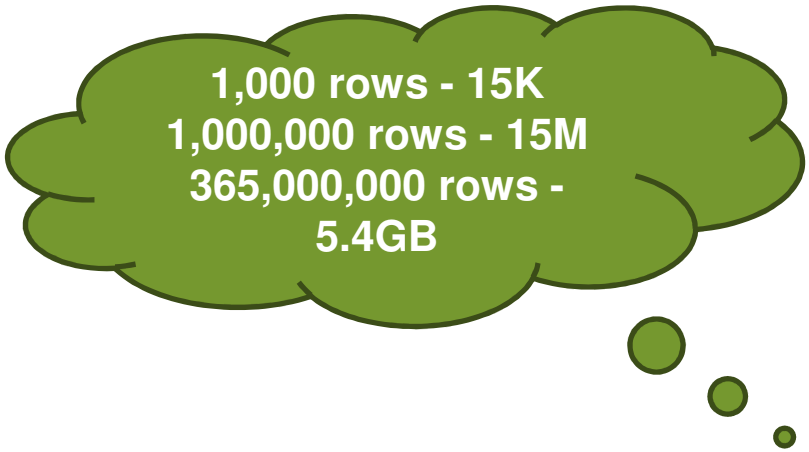
РАЗМЕР ЗАПИСИ И ПРОИЗВОДИТЕЛЬНОСТЬ

Золотое правило

- **Уменьшайте размер записей (таблицы и индексы)**
- Используйте «правильные» типы данных
  - Boolean: bit; not tinyint, smallint, int
  - DateTime & Double: какая точность необходима? (float, datetime2, money, ..)
  - Не используйте типы данных с фиксированной длиной (char, binary) за исключением ситуаций когда размер данных статичен

Золотое правило

- Думайте о будущем!
- **Пересоздавайте индексы после изменения структуры (alter table)**




```
create table Positions
(
    ATime datetime not null, -- 8 bytes
    Latitude float not null, -- 8 bytes
    Longitude float not null, -- 8 bytes
    IsValid int not null, -- 4 bytes
    IsAssistanceUsed int not null, -- 4 bytes
    -- total: 32 bytes
)
```

```
create table Positions2
(
    ATime datetime2(0) not null, -- 6 bytes
    Latitude decimal(9,6) not null, -- 5 bytes
    Longitude decimal(9,6) not null, -- 5 bytes
    IsValid bit not null, -- 1 byte
    IsAssistanceUsed bit not null, -- 0 bytes
    -- total: 17 bytes
)
```

Типы индексов

- Кластерный индекс
  - Определяет сортировку данных внутри таблицы
  - Один на таблицу
- Некластерный индекс
  - SQL Server 2005 – 249 на таблицу
  - SQL Server 2008+ – 999 на таблицу
- Пример - книга
  - Номер страницы – кластерный индекс
  - Аннотация в конце книги – некластерный индекс
- Heap таблицы – таблицы без кластерного индекса
  - Производительность *обычно* уступает таблицам с кластерным индексами
  - Идеальны для быстрой загрузки данных

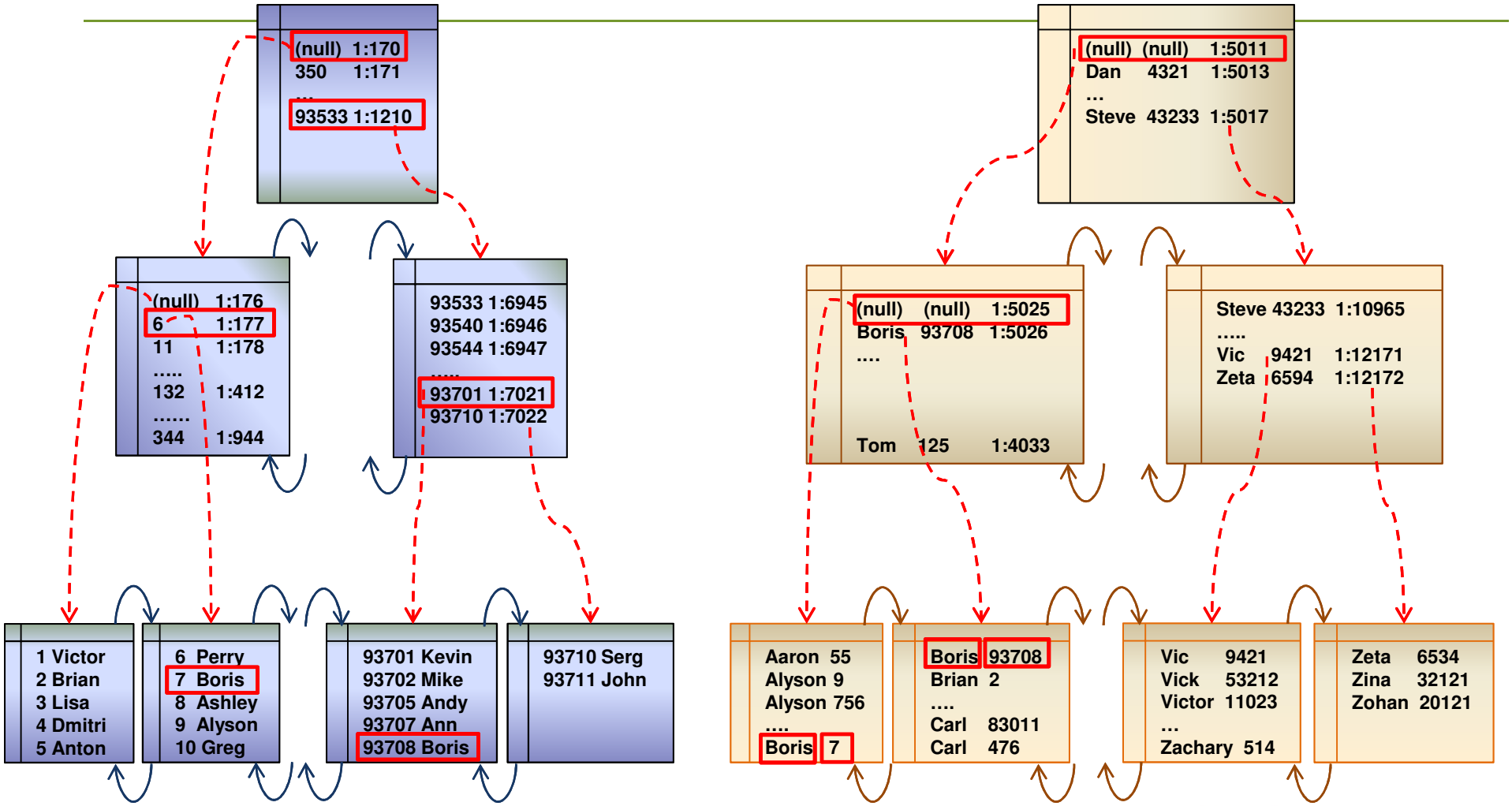
Структура и использование индексов

```
create table dbo.Customers
(
    CustomerId int not null,
    Name nvarchar(64) not null,
    EMail nvarchar(256) not null,
    Address1 nvarchar(100) not null,
    --...
)
go

create unique clustered index IDX_Customers_CustomerId
on dbo.Customers(CustomerId)
go

create nonclustered index IDX_Customers_Name
on dbo.Customers(Name)
go
```

Clustered Index

```
select *
from dbo.Customers
where Name = N'Boris'
```

Nonclustered Index

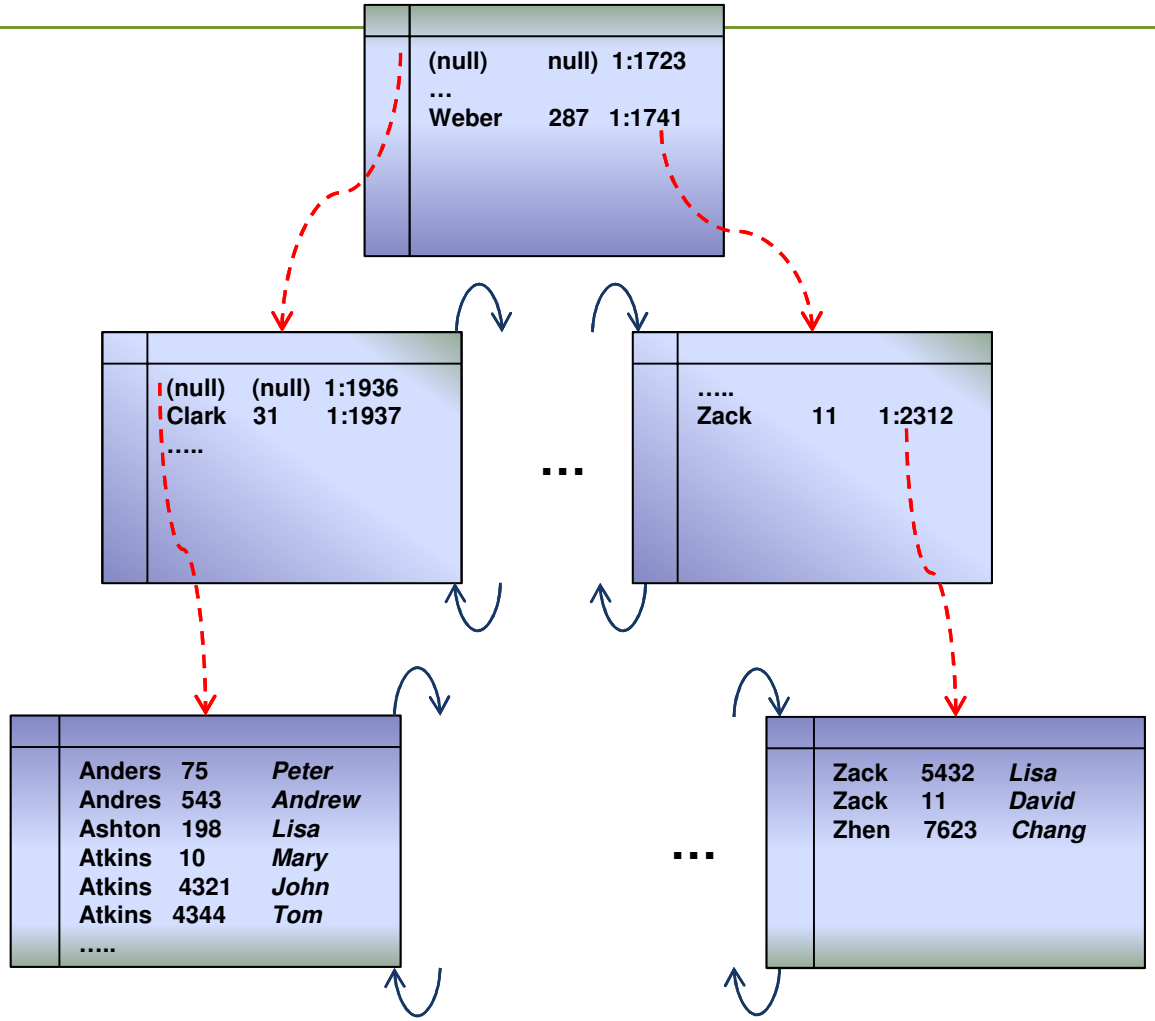


Демонстрация

ИСПОЛЬЗОВАНИЕ НЕКЛАСТЕРНЫХ ИНДЕКСОВ

Индексы с включенными полями

```
create table dbo.Customers
(
    CustomerId int not null,
    FirstName nvarchar(32) not null,
    LastName nvarchar(100) not null,
    EMail nvarchar(256) not null
)
go

create unique clustered index IDX_CI
on dbo.Customers(CustomerId)
go

create nonclustered index IDX_NCI
on dbo.Customers(LastName)
include(FirstName)
go
```



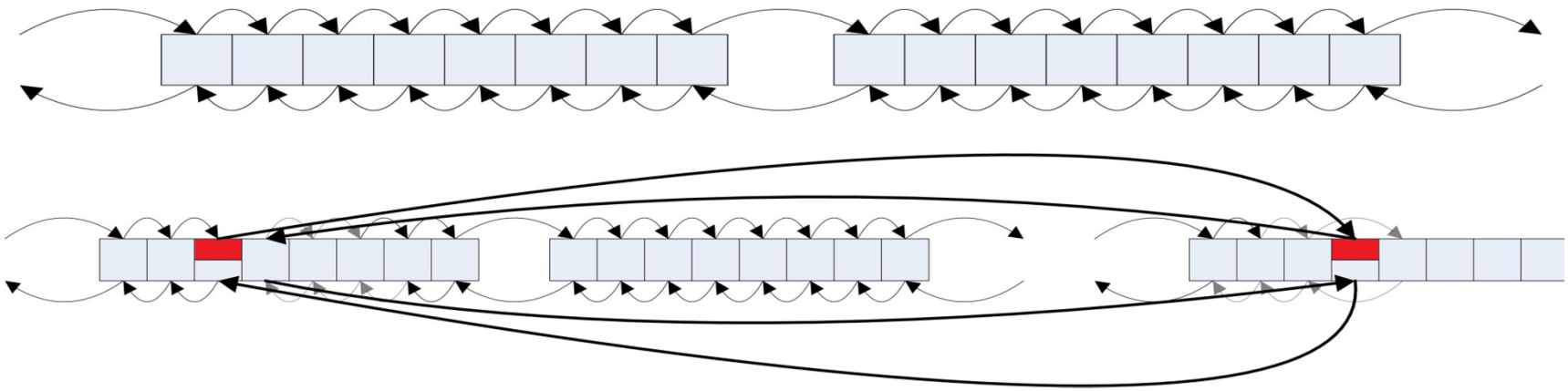
Кластерный индекс

- Идеальный кластерный индекс
  - Уменьшает фрагментацию
  - Оптимизирует наиболее важные запросы
- С точки зрения дизайна идеальный CI:
  - Статичен
    - При изменении CI данные переносятся в другое место и все NCI изменяются
  - Узок
    - Ключ присутствует во всех записях NCI
  - Уникален
    - Для неуникальных CI SQL Server добавляет *uniquifier int null*
      - Реальная избыточность варьируется от 0 до 8 байт

Identity / Sequence / Uniqueidentifier

- Уникальны, Статичны, Узки
- Потенциальные проблемы
  - «Горячие точки» – задержки во время создания новых страниц/участков (extents)
  - Редко помогают оптимизировать запросы
- Идеальное использование
  - Таблицы - справочники (Articles, Customers, и.т.д)
  - Отсутствие других кандидатов для CI
- Identity – 4/8 байт. Uniqueidentifiers – 16 байт
  - NEWID() – создает фрагментацию

Разрыв страниц (Page Split)



- SQL Server переносит ~50% данных на новую страницу
- FILLFACTOR позволяет зарезервировать место на странице
  - Работает только во время создания/пересоздания индексов (CREATE INDEX/ALTER INDEX REBUIL)
  - Уменьшает фрагментацию, но увеличивает размер индекса (как на диске, так и в памяти)

Паттерны приводящие к фрагментации

- Insert/Update – увеличение размера записи
- Read Committed Snapshot / Snapshot
- Триггеры
- Индексы на «случайные» значения
  - Uniqueidentifier
  - Hashbytes

Демонстрация

ФРАГМЕНТАЦИЯ

Внешние Ключи

- В большинстве случаев полезны
  - Гарантируют целостность данных
  - Помогают обнаружить ошибки во время разработки и тестирования
  - В некоторых случаях помогают оптимизатору
- Незначительно увеличивают нагрузку на систему
  - Будьте внимательны если данные часто модифицируются
  - Вставка в «Detail» таблицу влечет проверку на наличие «Master» записи
  - Удаление из «Master» таблицы влечет проверку на наличие записей в «Detail» таблице
    - **Всегда создавайте индекс в «Detail» таблице на столбец, участвующий во внешнем ключе**
- Несовместима с некоторыми функциями
  - Например не поддерживают переключение секций в секционированных таблицах

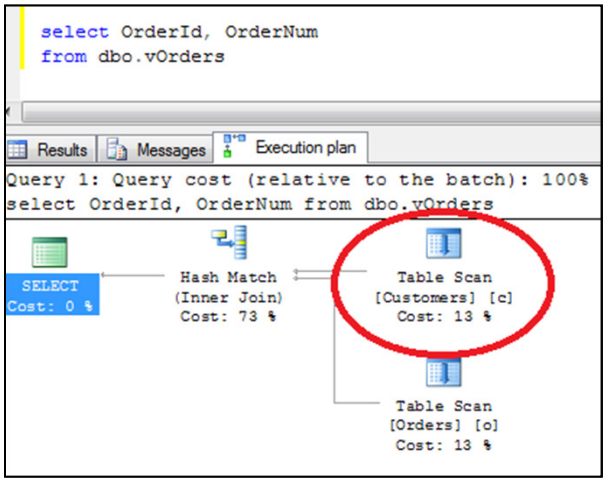
Unique & Check Constraints

- Unique Constraints
  - Уникальность значений в пределах столбца
  - Внутренняя имплементация через уникальный индекс
    - Unique constraint – концепция логического дизайна
    - Unique index – концепция физического дизайна
  - Используйте уникальный индекс
    - Оптимизация – можно добавить INCLUDE столбец
- Check Constraints
  - Поддерживают логическую целостность данных
  - Могут быть использованы вместе с UDF
  - Помогают оптимизатору запросов
  - Увеличивают нагрузку при модификации данных

Представления

```
create table dbo.Customers
(
    CustomerId int not null,
    Name nvarchar(32) not null,
)

create table dbo.Orders
(
    OrderId int not null,
    OrderNum int not null,
    CustomerId int not null
)

create view dbo.vOrders
as
    select
        o.OrderId, o.OrderNum,
        o.CustomerId,
        c.Name as CustomerName
    from
        dbo.Orders o join dbo.Customers c on
            o.CustomerId = c.CustomerId
```

```
select OrderId, OrderNum
from dbo.vOrders
```




```
create view dbo.vOrders
as
    select
        o.OrderId, o.OrderNum,
        o.CustomerId,
        c.Name as CustomerName
    from
        dbo.Orders o left outer join dbo.Customers c on
            o.CustomerId = c.CustomerId
```

```
alter table dbo.Orders
add constraint FK_Orders_Customers
foreign key(CustomerId)
references dbo.Customers(CustomerId)
```

Материализованные представления

- Данные «физически» материализованы подобно обычным таблицам и индексам
- Очень много ограничений при создании
- Standard Edition – необходимо использовать (NOEXPAND) хинт
- Дополнительная нагрузка при модификации данных
- Сценарии использования
  - Агрегаты на статичных данных
  - Оптимизация joins
  - Оптимизация стороннего кода

Демонстрация

МАТЕРИАЛИЗОВАННЫЕ ПРЕДСТАВЛЕНИЯ

Триггеры

- Ухудшают производительность
- Увеличивают фрагментацию данных
- Избегайте «длительных операций» внутри триггеров:
  - Доступ к внешним ресурсам (CLR)
  - Связанные сервера и распределенные транзакции
- CONTEXT_INFO позволяет «передавать параметры» в триггеры

XML

- Поддержка XML в SQL Server далека от идеала
  - Плохая производительность
  - Неоптимальные планы
  - Нагрузка на CPU
- OPENXML быстрее, однако использует 1/8ю памяти SQL Server
- Будьте осторожны
  - **Attribute-Centric XML быстрее чем Element-Centric XML**

CLR

TSQL

- Доступ и манипуляция данными
- Декларативные запросы (set-based)

CLR

- Работа со строками
- Regular Expressions
- Математика
- Доступ к файлам и внешним ресурсам
- Криптография
- Logging вне БД

CLR

- 32 бит ОС: Используется память процесса (не AWE)
- Смена контекста (context switching)
  - Операция может быть быстрее в CLR, однако в некоторых случаях T-SQL может быть выигрывать из-за отсутствия смен контекста
- Потенциальные проблемы с внедрением систем
- Некоторые сложности с технологиями высокой доступности

Ad-Hoc SQL

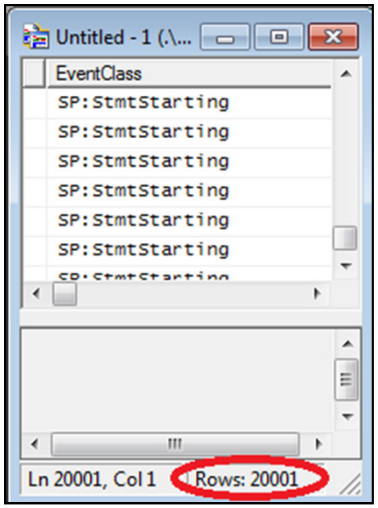
- Проблемы с SQL Injection
- Нагрузка на CPU во время рекомпиляции
- Расход памяти в кэше
  - SQL Server 2008+ - “Optimize for Ad-hoc workload” опция

Демонстрация

AD-HOC SQL

Функции

```
create function dbo.AvgItemCost(@OrderId int)
returns money
as
begin
    declare
        @Result money

    select @Result = Sum(Quantity * UnitPrice) / Sum(Quantity)
    from dbo.OrderItems
    where OrderId = @OrderId

    return @Result
end
go


select Sum(dbo.AvgItemCost(OrderId))
from dbo.Orders
```




```
create function dbo.AvgItemCostInline(@OrderId int)
returns table
as
return
(
    select Sum(Quantity * UnitPrice) / Sum(Quantity) as [AvgCost]
    from dbo.OrderItems
    where OrderId = @OrderId
)
go

select Sum(a.AvgCost)
from
    dbo.Orders o cross apply
        dbo.AvgItemCostInline(o.OrderId) a
```

Временные табличные переменные

- Миф: TTV не используют tempdb
- SQL Server не использует статистику с TTV
  - Может привести к сабоптимальным планам
    - В некоторых случаях можно использовать OPTION (RECOMPILE)
- TTV живут вне транзакционного контекста

Временные табличные переменные

```
create table #TT(ID int)
declare @TTV table(ID int)

begin tran
    insert into #TT values(1)
    insert into @TTV values(1)
rollback
select * from #TT
select * from @TTV
```

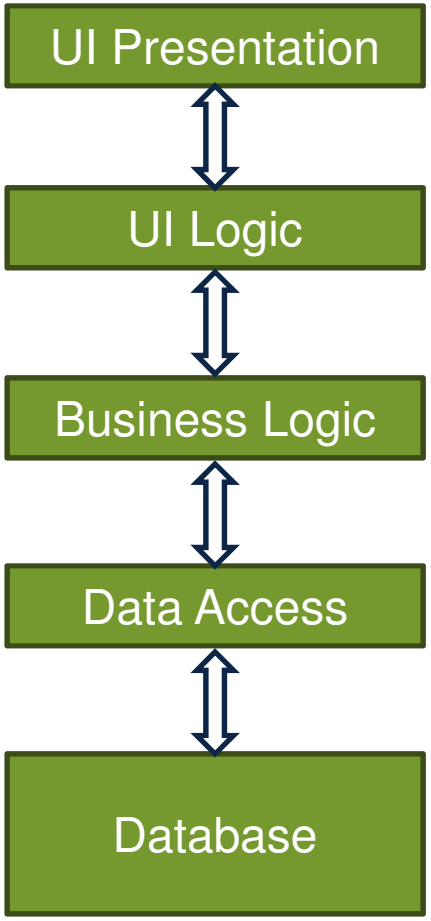
Results | Messages

ID

ID
1

И НЕМНОГО ОБ АРХИТЕКТУРЕ

N-Tier Архитектура



- Не имеет большого значения сколько физических уровней в системе
- Разделяйте *логические* уровни
- Изолируйте уровень доступа к данным (Data Access Tier)
  - Поддержка
  - Маштабирование
  - Стоимость внесения изменений

Уровень доступа к данным

- Представления (Views)
  - Помогают с безопасностью (security)
  - Обладают ограниченными возможностями
  - Потенциальные проблемы с joins
- Хранимые процедуры
  - Гибкость на стороне БД
  - Идеальны для рефакторинга при условии, что интерфейс не меняется
- ORM Frameworks
  - Зависит от..

ORM / Генераторы кода

- Замечательная идея. Облегчает разработку. Но:
  - Черный ящик, который мы не можем контролировать
  - Насколько хорош сгенерированный SQL?
- Будьте внимательны если производительность критична для системы
- Как минимум мы должны иметь возможность перехватить и изменить сгенерированный SQL

Параметры

```
var o = from c in db.Customers
        where c.City = "Tampa"
        select c;
```

```
exec sp_executesql N'select * from Customers where City=''Tampa'''
```

```
var o1 = from c in db.Customers
         where c.City = "Tampa"
         select c;

var o2 = from c in db.Customers
         where c.City = "St.Petersburg"
         select c;
```

```
exec sp_executesql N'select * from Customers where City=@P1', N'@P1 nvarchar(5)', @P1=N'Tampa'

exec sp_executesql N'select * from Customers where City=@P1', N'@P1 nvarchar(13)', @P1=N'St.Petersburg'
```

```
var o1 = from c in db.Customers
         where c.ID = 1
         select c;

var o2 = from c in db.Customers
         where c.ID = 1000
         select c;
```

```
exec sp_executesql N'select * from Customers where ID=@P1', N'@P1 tinyint', @P1=1

exec sp_executesql N'select * from Customers where ID=@P1', N'@P1 smallint', @P1=1000
```

Списки

```
select * from dbo.Orders
where OrderId in (123, 456, /* A few thousands more */)
```

```
select * from dbo.Orders
where OrderId in (@P1, @P2, /* A few thousands more */)
```

- Перекомпиляция и кэшированные планы
- Ухудшение производительности и сабоптимальные планы (>64 элементов)
- Идеальный вариант:
  - TVP (table-value parameters)

```
select * from dbo.Orders
where OrderId in (
    select ID from @TempTable
)
```

Пища для размышления

- Получение скалярного значения
  - Атрибут или объект?
- Lazy/Eager загрузка
  - Что является более предпочтительным?
  - Есть ли у нас контроль?
- Ссылочная целостность
  - Особенно в сложных случаях
- Интеграция с метадатой
  - Default & Check constraints, Domain Values, итд
- Оптимистичные блокировки
  - Совместима ли имплементация с системой?
- Разделение объектной модели и схемы БД
  - Возможно ли хранить Contact.HomeAddress.Street в dbo.[Contact].[HomeAddress_Street]?

Мы обсудили

- Как SQL Server хранит данные
- Типы объектов и их влияние на производительность
- Поговорили об архитектуре систем

- Вопросы?
  - dmitri@aboutsqlserver.com