EVERYTHING YOU WANTED TO KNOW ABOUT INDEXES

...but were afraid to ask

- Presentation by Dmitri Korotkevitch
  dmitri@aboutsqlserver.com

PASS SQLRally

2012 MAY, 10-11
DALLAS, TX

About me

- 10+ years of experience working with SQL Server
- Microsoft SQL Server MVP
- MCITP (DBA, DB Developer), MCPD
- Blog: http://aboutsqlserver.com
- Email: dmitri@aboutsqlserver.com
- Demos are available for download
  - SQL Rally web site
  - http://aboutsqlserver.com/presentations

Microsoft CERTIFIED IT Professional

Microsoft CERTIFIED Professional Developer

Agenda

- Indexes structure and usage
  - Physical structure
  - Access methods and statistics
- Additional features
  - Indexes with included columns
  - Filtered indexes
  - Partitioned tables and indexes
- Fragmentation
- Indexing Strategies
- Optimization Strategies

Before we begin..

- Everything works differently in production
  - So we need to test, test, test and then test again
- There is only one right answer: “It Depends”
  - So we need to test, test, test and then test again
- Every system is unique – don’t be afraid to think out of the box
  - And test, test, test and then test again

Index Structure and Usage

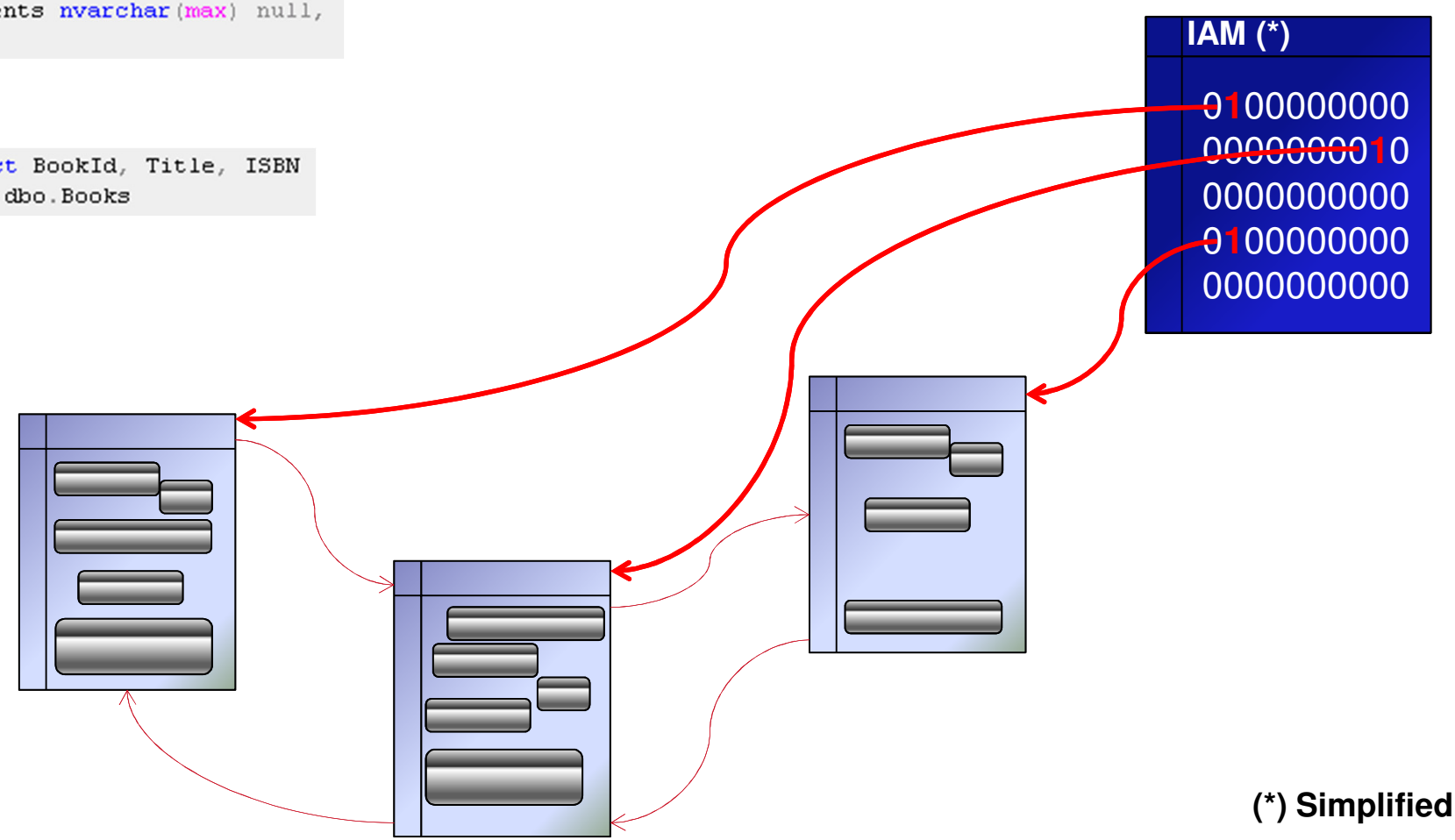
In the beginning was the Page

- Everything stores at 8K pages in double linked list
  - 8060 bytes available to user
- 8 pages group to “Extent”
  - Mixed extents store data belongs to different objects
  - Unified extents store data belongs to the same object
- First 8 object pages stored in the mixed extents. After that only unified extents are used.
- SQL Server uses special pages – “Index Allocation Map” (IAM) to track extents that belong to the object

Heap Tables

```
create table dbo.Books
(
    BookId int not null,
    Title nvarchar(256) not null,
    ISBN char(14) not null,
    Comments nvarchar(max) null,
)
```

```
select BookId, Title, ISBN
from dbo.Books
```



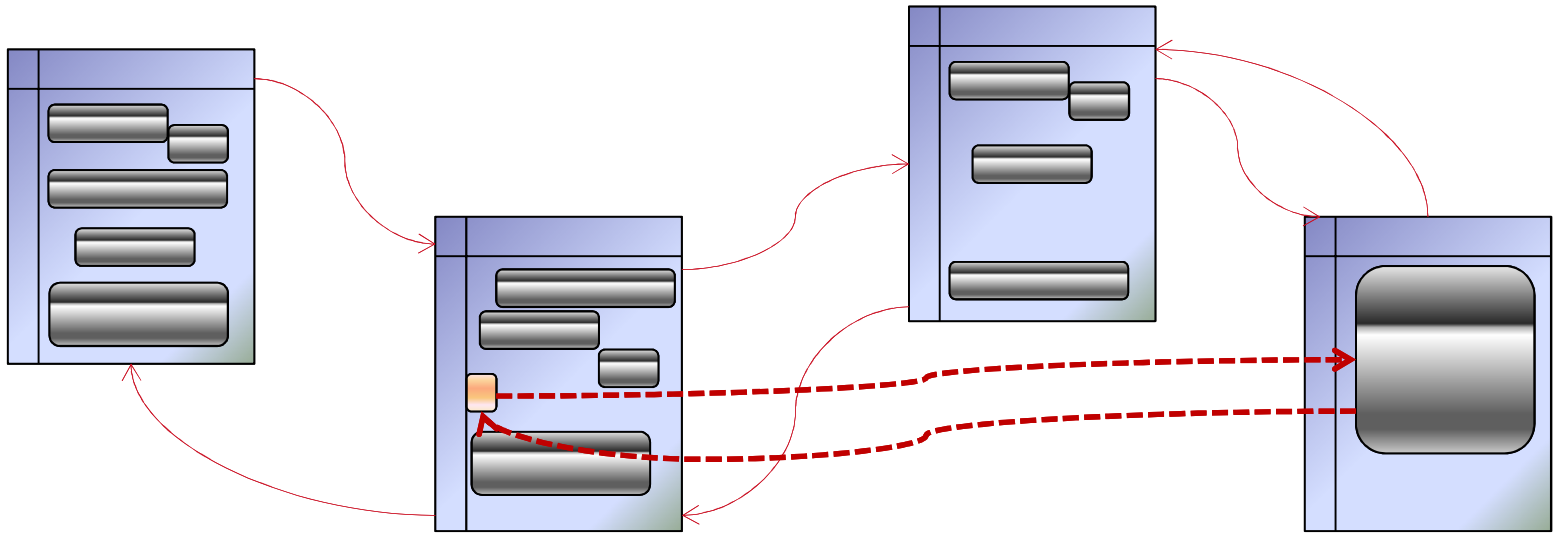
(*) Simplified

Heap Tables

```
create table dbo.Books
(
    BookId int not null,
    Title nvarchar(256) not null,
    ISBN char(14) not null,
    Comments nvarchar(max) null,
)
```

```
update dbo.Books
set
    Comments = N'Very Long Text'
where
    BookId = 123
```

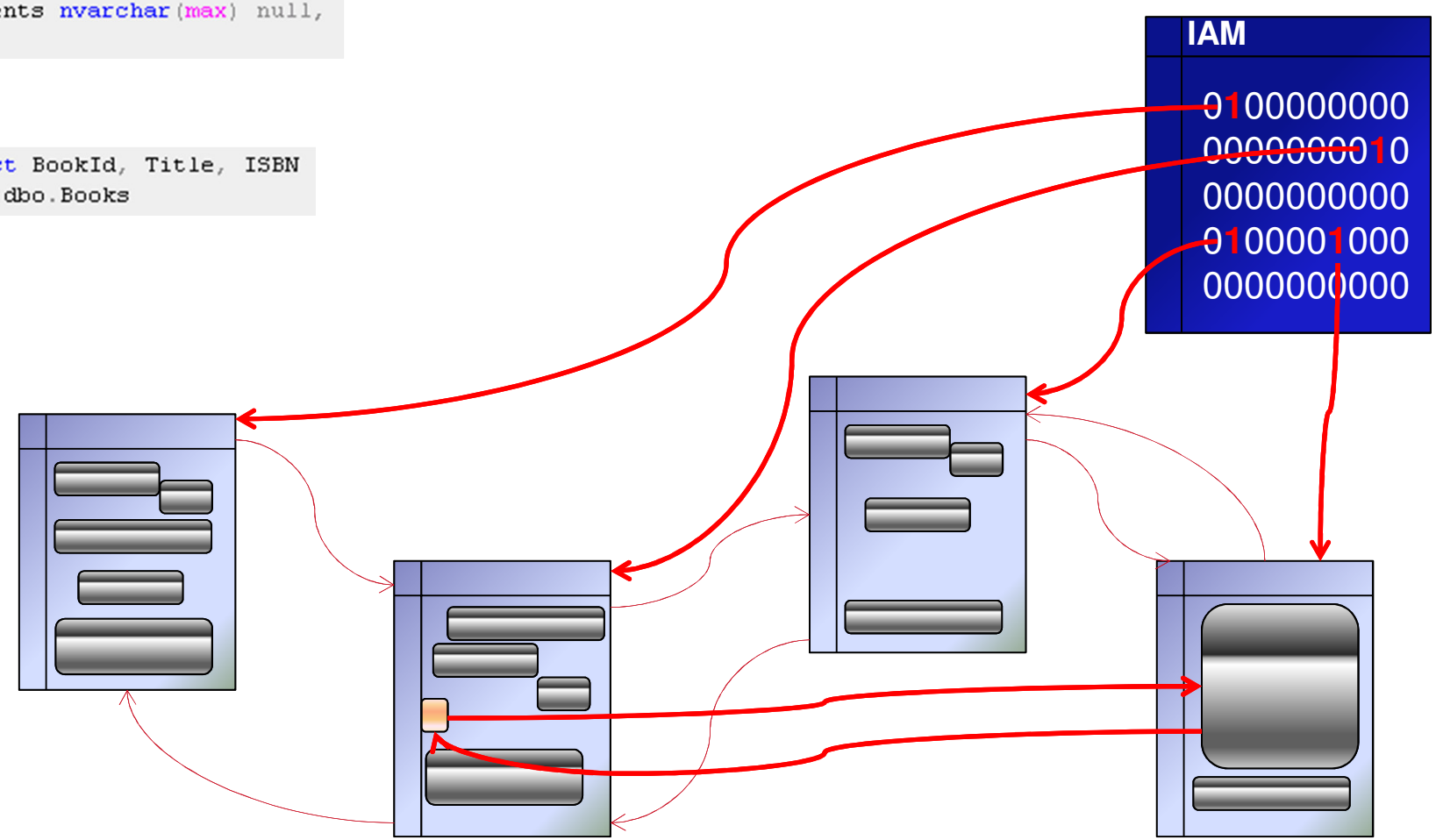
IAM
0100000000
0000000010
0000000000
0100001000
0000000000



Heap Tables

```
create table dbo.Books
(
    BookId int not null,
    Title nvarchar(256) not null,
    ISBN char(14) not null,
    Comments nvarchar(max) null,
)
```

```
select BookId, Title, ISBN
from dbo.Books
```

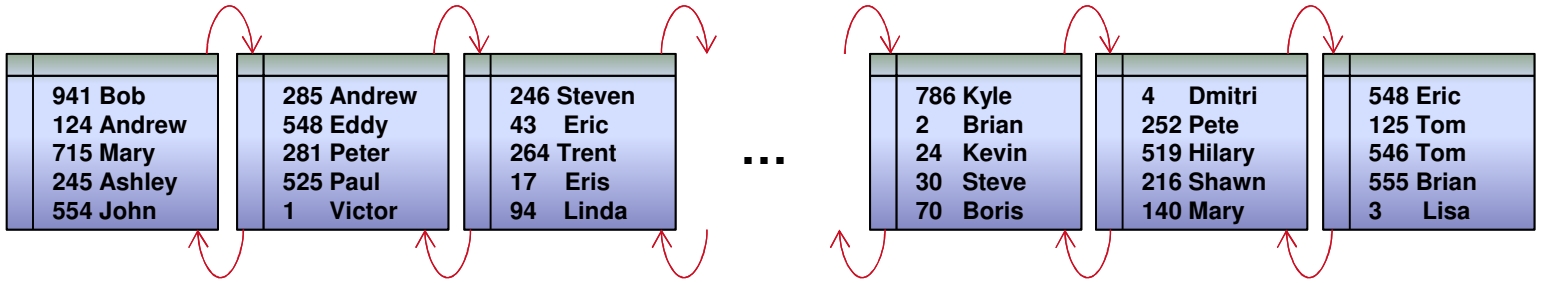


Heap Tables

- Potential Problems:
  - Extra I/O because of forwarding pointers
  - Suboptimal control of free space
- Potential Usage:
  - Staging environment where fast data load is the key

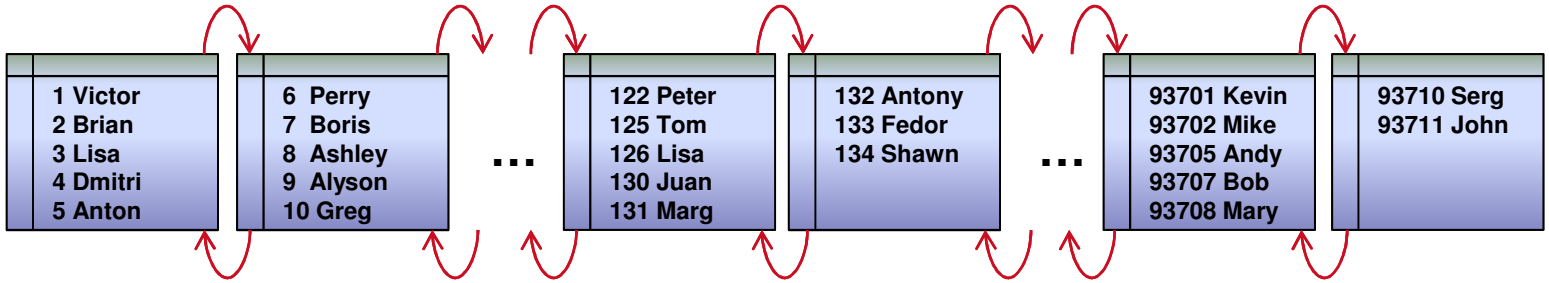
Clustered Index

```
create table dbo.Customers
(
    CustomerId int not null,
    Name nvarchar(128) not null,
    Phone varchar(32) not null,
    DateOfBirth date null
)
```



Heap Table

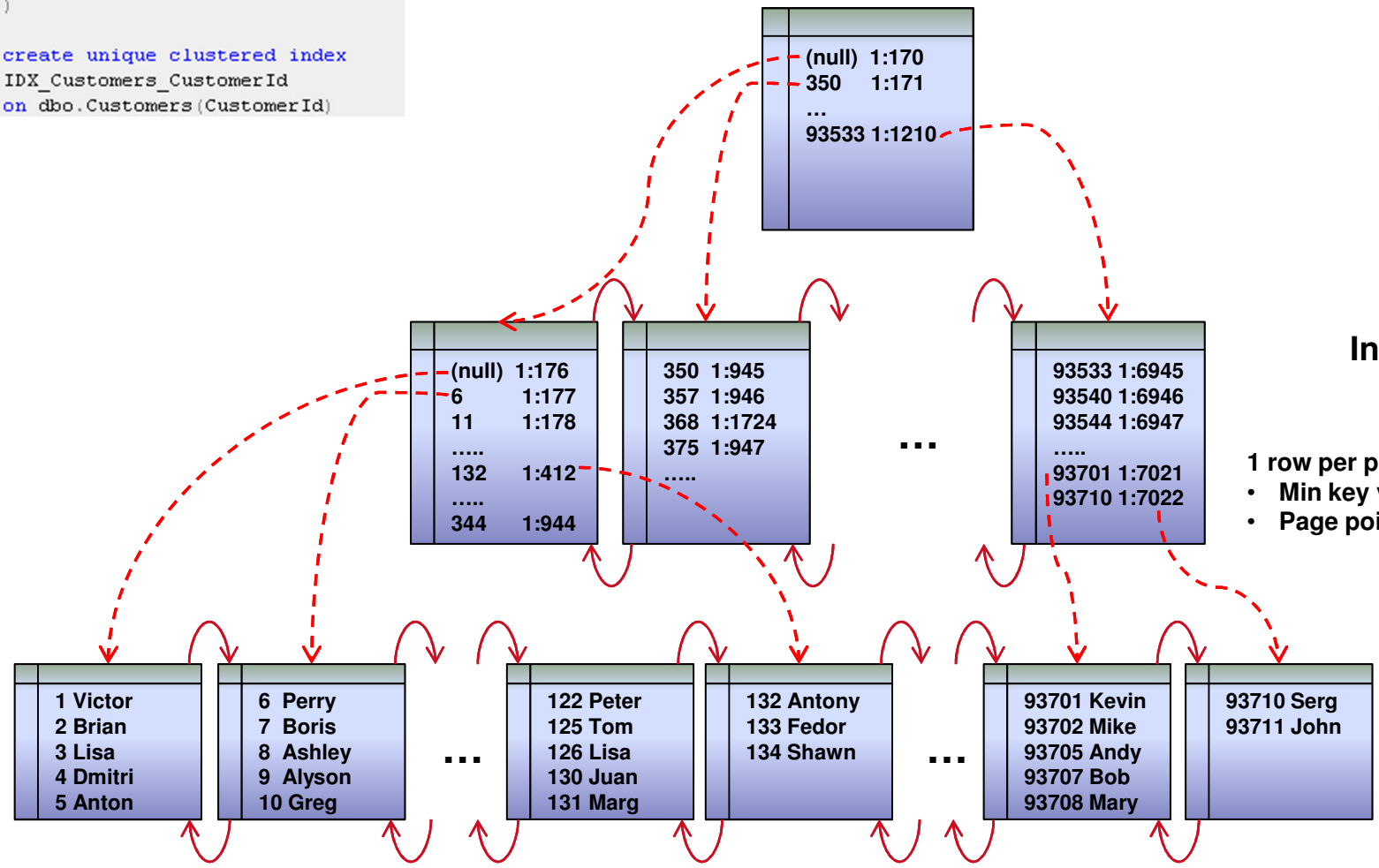
```
create unique clustered index
IDX_Customers_CustomerId
on dbo.Customers(CustomerId)
```



Leaf Level (actual data)

```
create table dbo.Customers
(
    CustomerId int not null,
    Name nvarchar(128) not null,
    Phone varchar(32) not null,
    DateOfBirth date null
)

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

Clustered Index



Root Level

Intermediate Level

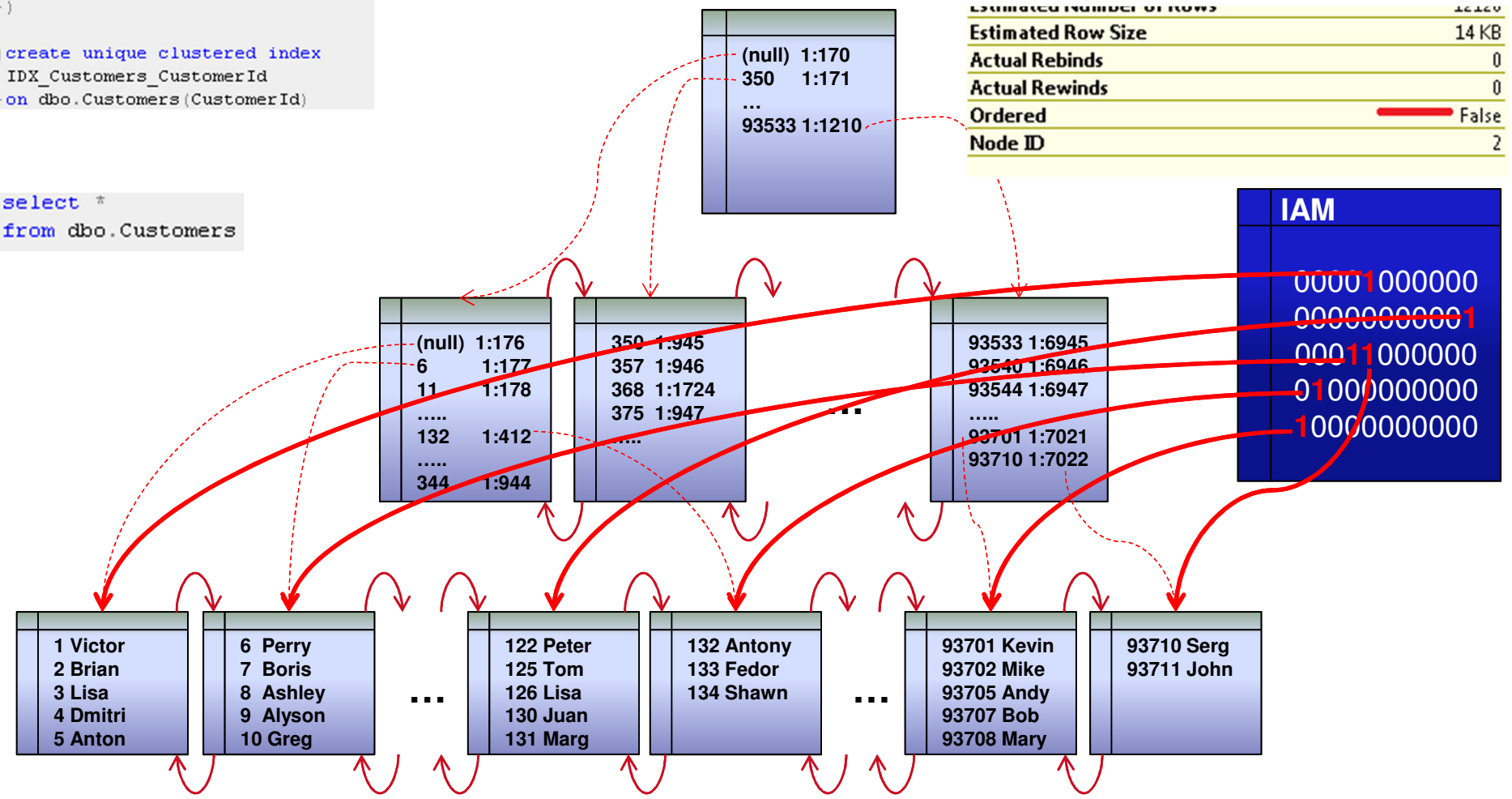
1 row per page on the "next level"
- **Min key value from the page**
- **Page pointer (file:page)**

Leaf Level (actual data)

Index Usage
IAM Scan

```
create table dbo.Customers
(
    CustomerId int not null,
    Name nvarchar(128) not null,
    Phone varchar(32) not null,
    DateOfBirth date null
)

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

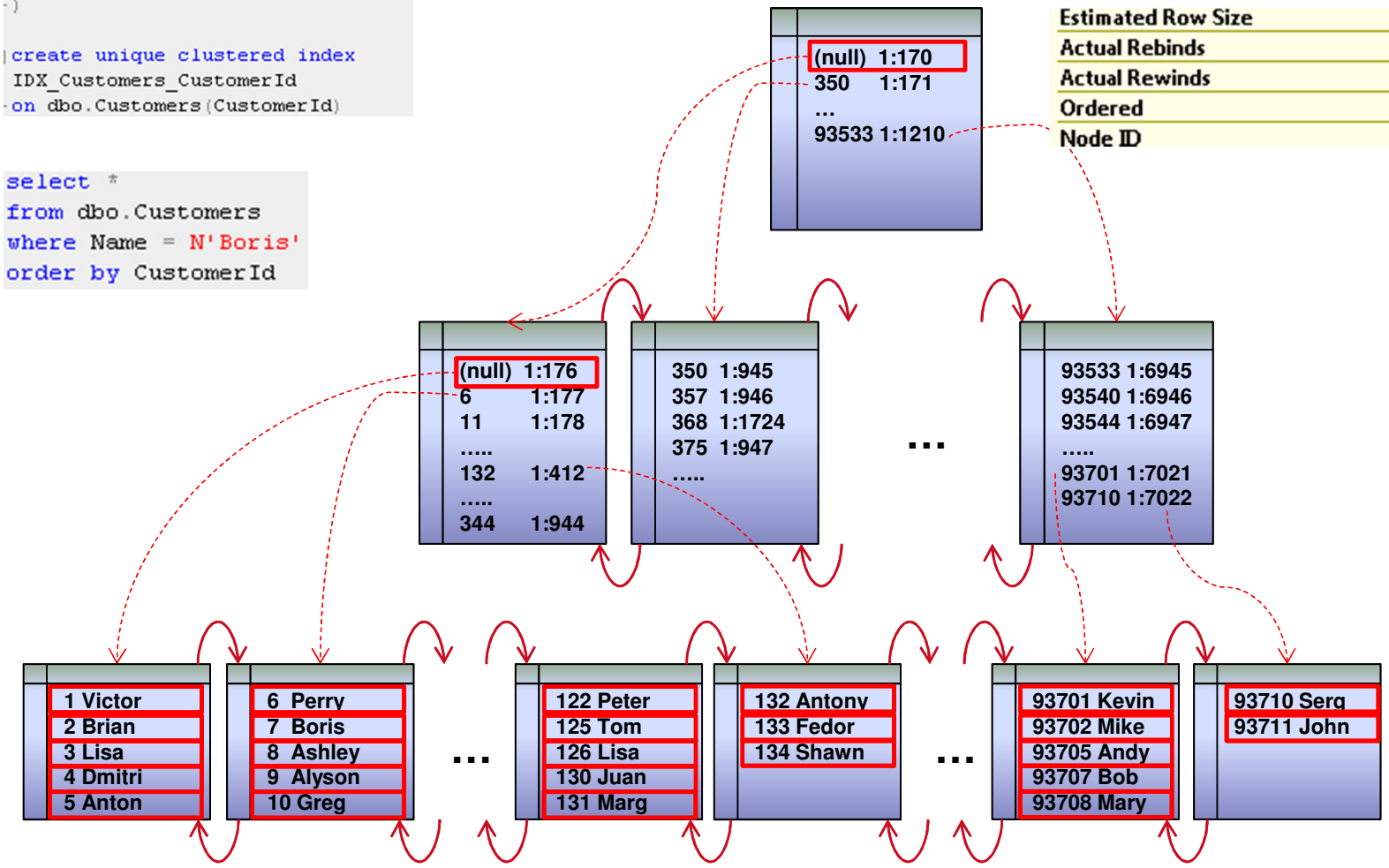
```
select *
from dbo.Customers
```



Index Usage
Ordered Scan

```
create table dbo.Customers
(
    CustomerId int not null,
    Name nvarchar(128) not null,
    Phone varchar(32) not null,
    DateOfBirth date null
)

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

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

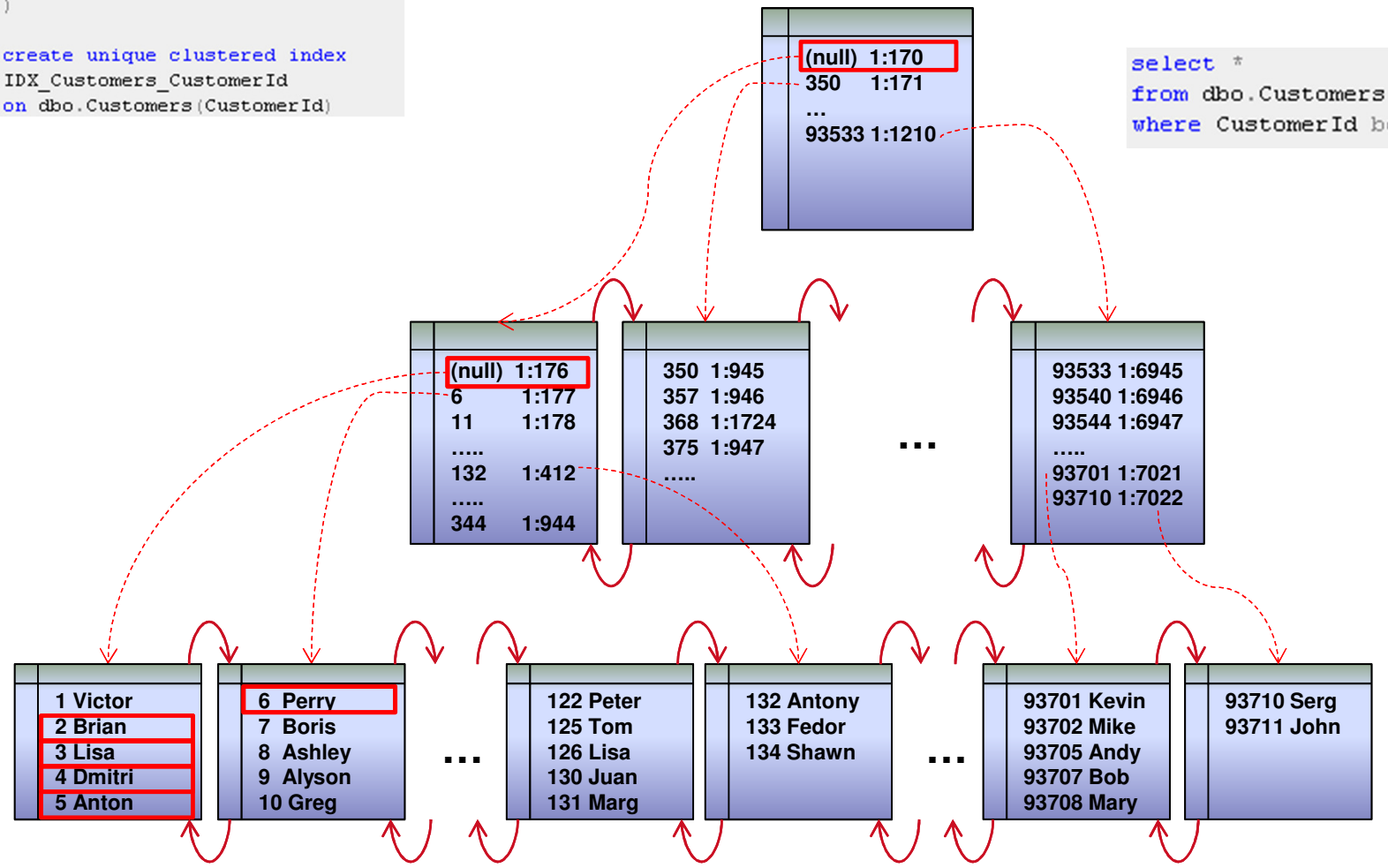


Estimated Row Size	14 KB
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	2

```
create table dbo.Customers
(
    CustomerId int not null,
    Name nvarchar(128) not null,
    Phone varchar(32) not null,
    DateOfBirth date null
)

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

Index Usage
Index Seek

```
select *
from dbo.Customers
where CustomerId between 2 and 6
```



SARG (Seekable/Searchable Arguments)

- SARG predicates lead to Index SEEK

```
where CustomerId = 1

where CustomerId > 1000

where CustomerId in (1, 2)

where VarCharCol like 'A%'

where DateCol between '2012-01-01' and '2012-02-01'
```

- Non-SARG predicates – Index SCAN only

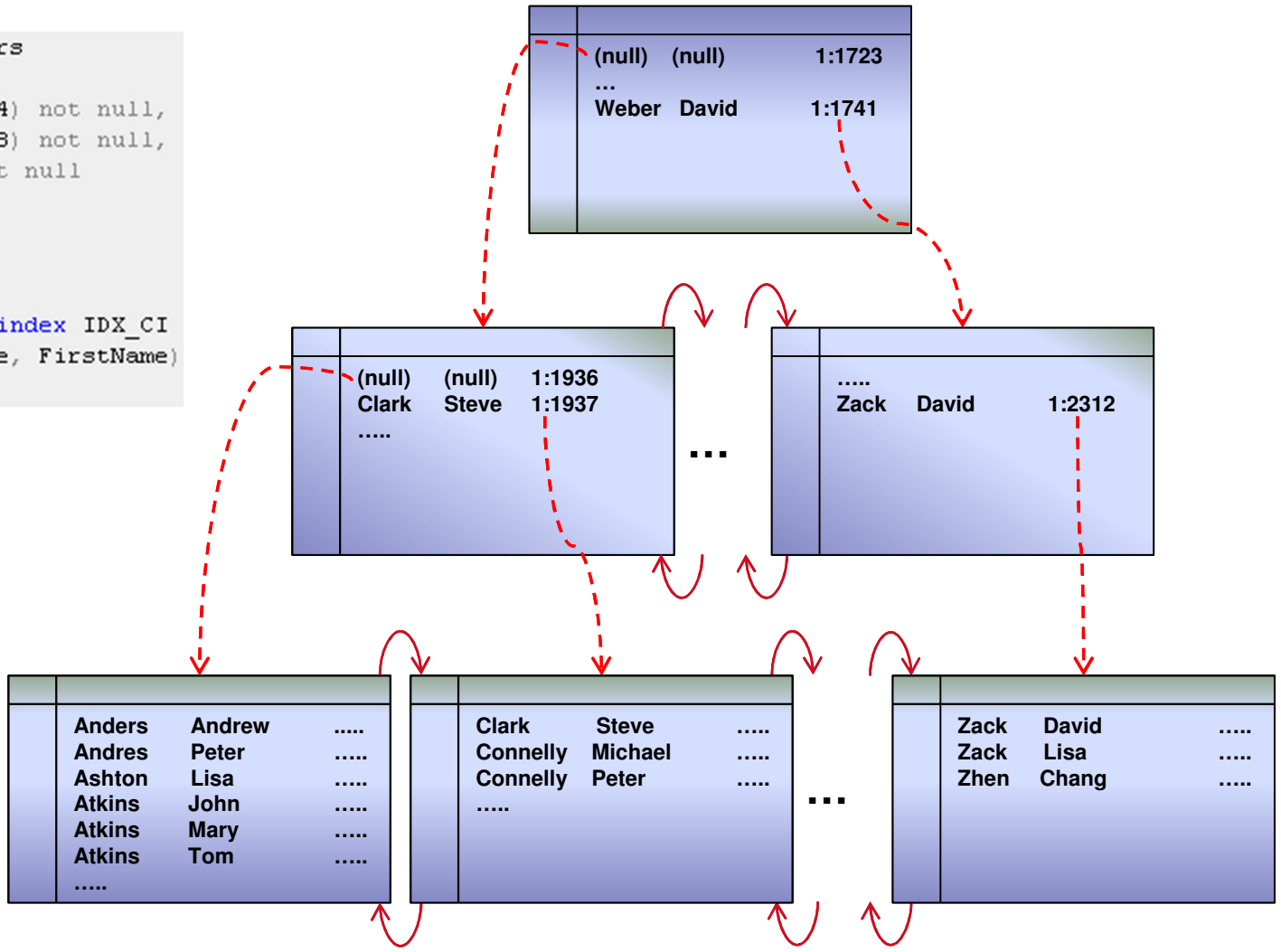
```
where IntCol + 1 = 10

where ABS(IntCol) = 1

where DATEPART(YEAR,DateTimeCol) = 2012

where DATEADD(DAY,7,DateTimeCol) > GETDATE()

where VarCharCol like '%A%'
```

```
where IntCol = 9

where IntCol in (-1, 1)

where DateTimeCol >= '2012-01-01' and DateTimeCol < '2013-01-01'

where DateTimeCol > DATEADD(DAY,-7,GETDATE())
```

```
where VarCharCol = N'nvarchar parameter'
```

Composite Index

```
create table dbo.Customers
(
    FirstName nvarchar(64) not null,
    LastName nvarchar(128) not null,
    Phone varchar(32) not null
    -- ...
)
go

create unique clustered index IDX_CI
on dbo.Customers(LastName, FirstName)
go
```



SARG for Composite Index

```
create table dbo.Customers
(
    FirstName nvarchar(64) not null,
    LastName nvarchar(128) not null,
    Phone varchar(32) not null
    -- ...
)
go

create unique clustered index IDX_CI
on dbo.Customers(LastName, FirstName)
go
```

- SARG predicates:

```
where LastName = N'Sanders' and FirstName = N'Tom'

where LastName = N'Sanders'
```

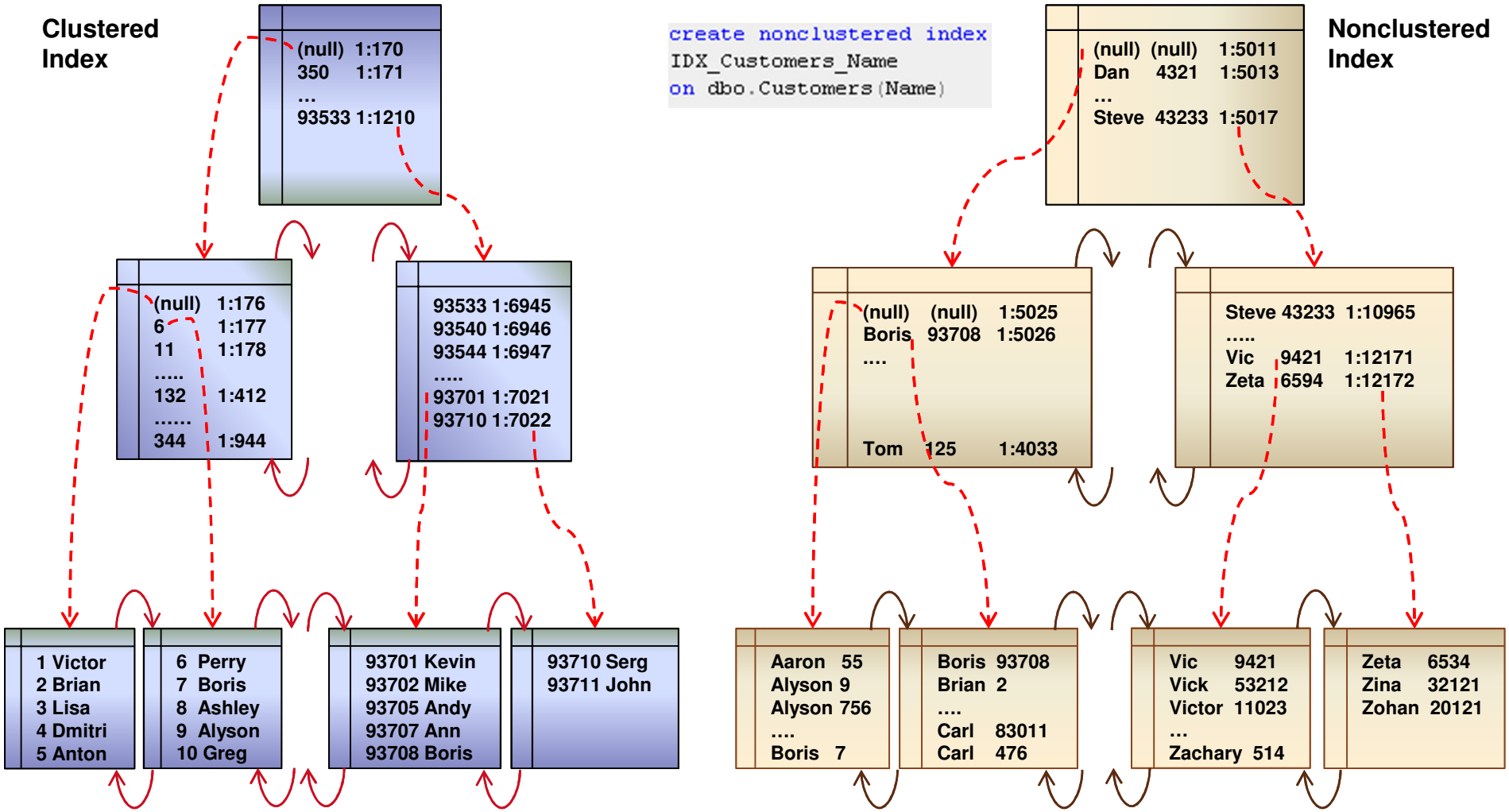
SARG predicates on the “left” column(s) of the index

- Non-SARG predicates:

```
where LastName like N'%A%' and FirstName = N'Tom'

where FirstName = N'Tom'
```

All other cases

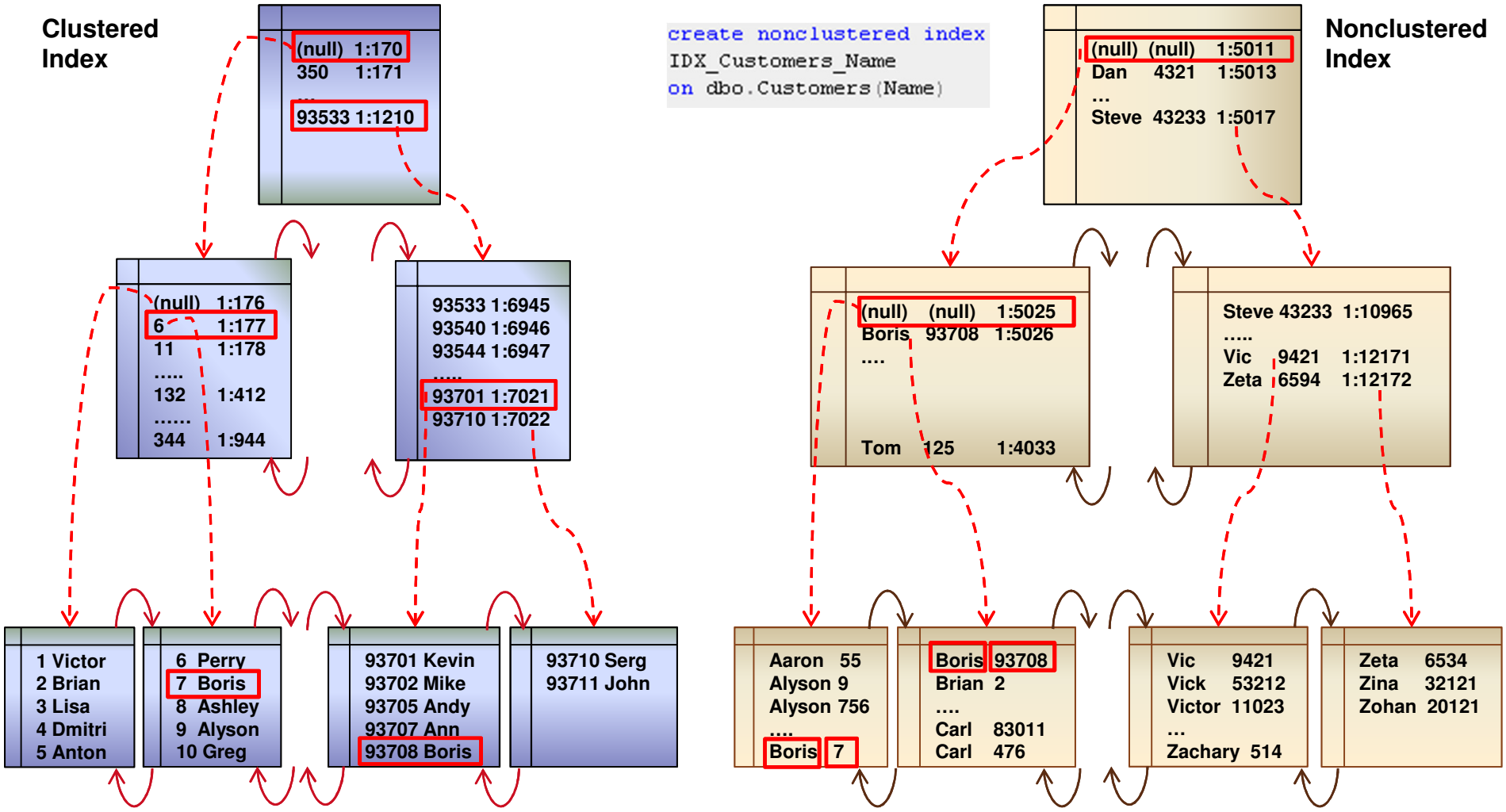
Nonclustered Index



Nonclustered Index

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

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



Nonclustered Index Structure

- Leaf level: Key value + row id
- Row id:
  - Heap Tables: row id = file:page:slot
  - Tables with Clustered Index: row id = CI key value
- Intermediate and Root levels
  - Unique indexes: Page pointer + minimal key value from the page
  - Non unique indexes: Page pointer + minimal key value from the page + row id for that key

Nonclustered Index Usage Cost

- Approximate cost of read operation:

  # of levels in nonclustered index +

  # of pages scanned at the leaf level +

  *# of rows found * # of levels in the clustered index*

Query Optimizer

- Optimizer does not look for the *best* plan. The goal is to find *good enough* plan *fast enough*
- Generally Speaking:
  - Table/Clustered Index SCAN is always an option
  - Other options require cost estimate for NCI usage
    - One of the main criteria – how many rows would be found → how many Key Lookup operations required
    - SQL Server uses Statistics for the estimation

Statistics

- Histogram keeps 200 steps at most
  - More rows in the table leads to less accurate statistics
- Estimation errors progressing through the plan
- Histogram kept for the first (left) index column only
- Automatic statistics update based on the number of changes of the first (left) index column
  - Add rows to the empty table
  - <= 500 rows → 500 changes
  - > 500 rows → 500 + 20% of number of rows in the table

Statistics and Composite Indexes

```
create table dbo.Customers
(
    CustomerId int not null,
    FirstName nvarchar(64) not null,
    LastName nvarchar(128) not null,
    Phone varchar(32) not null
)
go

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

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

```
select *
from dbo.Customers
where FirstName = @FirstName
```

- Options:
  - Clustered index scan
  - Nonclustered index scan + Key lookup
- Cost depends on # of rows found
- Solution: column level statistics
  - In some cases SQL Server creates it automatically

```
create statistics CLS_FirstName
on dbo.Customers(FirstName)
go
```

Statistics and Composite Indexes

```
create table dbo.Positions
(
    CompanyId int not null,
    UTCTimeTag datetime2(0) not null,
    RecId bigint not null,
    DeviceId int not null,
    Latitude decimal(9,6) not null,
    Longitude decimal(9,6) not null
)
go

create unique clustered index IDX_CI
on dbo.Positions
(CompanyId, UTCTimeTag, RecId)
go

create nonclustered index IDX_NCI
on dbo.Positions
(CompanyId, DeviceId, UTCTimeTag)
```

- Histogram kept for the first (left) index column only
- DeviceId is unique within the company
- What index to choose?

```
select *
from dbo.Positions
where
    CompanyId = @CompanyId and
    UTCTimeTag between @StartTime and @StopTime and
    DeviceId in
    (
        select DeviceId
        from #Devices
    )
```

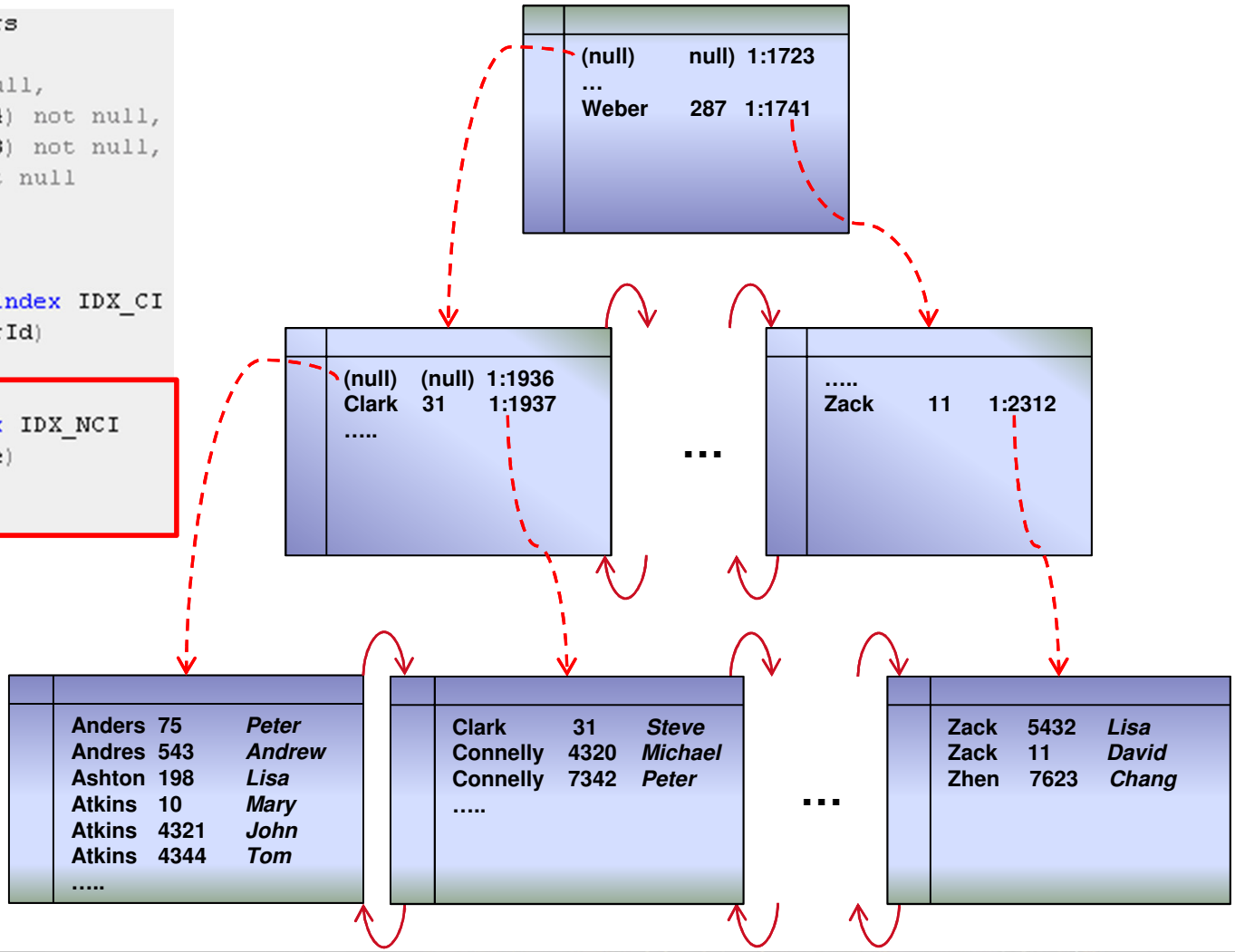
Summarize

- Key lookup is very expensive operation
  - SQL Server does not use nonclustered index in case it expects # of key lookups more than (very low) threshold
- SQL Server uses statistics to estimate the number of the rows
  - Histogram is kept for the first (left) index column only and has no more than 200 steps
  - Statistics is inaccurate in case of the large tables
  - Automatic statistics update requires an update of ~20% of the rows in the table

Additional Features

Indexes with included columns

```
create table dbo.Customers
(
    CustomerId int not null,
    FirstName nvarchar(64) not null,
    LastName nvarchar(128) not null,
    Phone varchar(32) 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
```



Indexes with Included Columns

- Included columns:
  - Stored at leaf level only
  - Unsorted
  - Do not count towards 900 bytes key size limitation
- Helps to remove Key lookups
- Helps with index consolidation

Indexes with Included Columns

- Downsides:
  - Bigger row size on the leaf level
  - Possible overhead during data modifications
  - Additional maintenance cost due larger index size

Filtered Indexes

```
create table dbo.Positions
(
    CompanyId int not null,
    UTCTimeTag datetime2(0) not null,
    RecId bigint not null,
    DeviceId int not null,
    Latitude decimal(9,6) not null,
    Longitude decimal(9,6) not null,
    Processed bit not null default 0
)
go

create unique clustered index IDX_CI
on dbo.Positions
(CompanyId, UTCTimeTag, RecId)
go

create unique nonclustered index
IDX_Positions_Processed_Filtered
on dbo.Positions(RecId)
where Processed = 0
go

select top 5000 *
from dbo.Positions
where Processed = 0
order by RecId
go
```

- Indexing of the part of the data
  - Smaller index size and maintenance cost
- Potential problems
  - Restrictions for the filters
  - Optimizer does not always choose filtered indexes
- Use Cases:
  - Indexing for special values of the key
  - Sparse columns
  - Uniqueness of (not null) values

Partitioned Tables

```
create table dbo.Data
(
    ID int not null,
    DateCreated datetime not null,
    DateModified datetime not null,
    -- ...
)
```

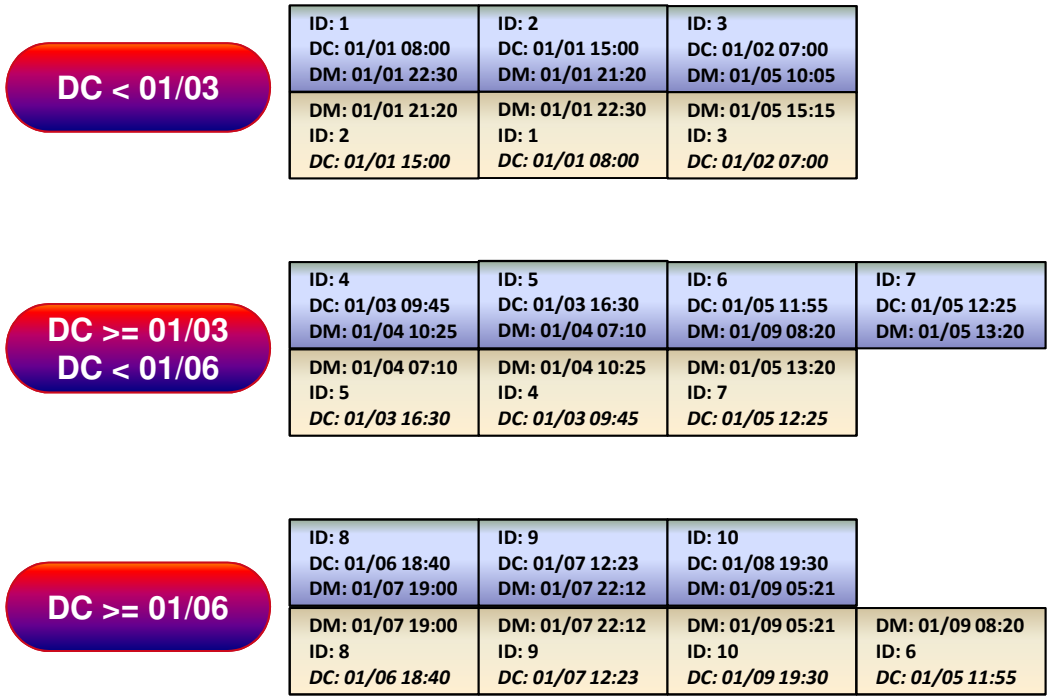
```
create unique clustered index IDX_CI
on dbo.Data(ID)
```

ID: 1 DC: 01/01 08:00 DM: 01/01 22:30	ID: 2 DC: 01/01 15:00 DM: 01/01 21:20	ID: 3 DC: 01/02 07:00 DM: 01/05 15:15	ID: 4 DC: 01/03 09:45 DM: 01/04 10:25	ID: 5 DC: 01/03 16:30 DM: 01/04 07:10	ID: 6 DC: 01/05 11:55 DM: 01/09 08:20	ID: 7 DC: 01/05 12:25 DM: 01/05 13:20	ID: 8 DC: 01/06 18:40 DM: 01/07 19:00	ID: 9 DC: 01/07 12:23 DM: 01/07 22:12	ID: 10 DC: 01/08 19:30 DM: 01/09 05:21

```
create nonclustered index IDX_NCI
on dbo.Data(DateModified)
include(DateCreated)
```

DM: 01/01 21:20 ID: 2 *DC: 01/01 15:00*	DM: 01/01 22:30 ID: 1 *DC: 01/01 08:00*	DM: 01/04 07:10 ID: 5 *DC: 01/03 16:30*	DM: 01/04 10:25 ID: 4 *DC: 01/03 09:45*	DM: 01/05 13:20 ID: 7 *DC: 01/05 12:25*	DM: 01/05 15:15 ID: 3 *DC: 01/02 07:00*	DM: 01/07 19:00 ID: 8 *DC: 01/06 18:40*	DM: 01/07 22:12 ID: 9 *DC: 01/07 12:23*	DM: 01/09 05:21 ID: 10 *DC: 01/08 19:30*	DM: 01/09 08:20 ID: 6 *DC: 01/05 11:55*

Partitioned Tables




```
create partition function pfData(datetime)
as range right
for values('2012-01-03','2012-01-06')
go

create partition scheme psData
as partition pfData
all to ([primary])
go

create table dbo.Data
(
    ID int not null,
    DateCreated datetime not null,
    DateModified datetime not null,
    -- ...
)
go

create unique clustered index IDX_CI
on dbo.Data(ID,DateCreated)
on pfData(DateCreated)
go

create nonclustered index IDX_NCI
on dbo.Data(DataModified(DateCreated)
on pfData(DateCreated)
```

Partitioned Tables

- Clustered and aligned nonclustered indexes consist of multiple “physical” partitions
- Data (keys) is sorted within partition
  - That leads to quite interesting issues

Partitioned Tables

```
create table dbo.Data
(
    ID int not null,
    DateCreated datetime not null,
    DateModified datetime not null,
    -- ...
)
```

```
create unique clustered index IDX_CI
on dbo.Data(ID)
```

ID: 1 DC: 01/01 08:00 DM: 01/01 22:30	ID: 2 DC: 01/01 15:00 DM: 01/01 21:20	ID: 3 DC: 01/02 07:00 DM: 01/05 15:15	ID: 4 DC: 01/03 09:45 DM: 01/04 10:25	ID: 5 DC: 01/03 16:30 DM: 01/04 07:10	ID: 6 DC: 01/05 11:55 DM: 01/09 08:20	ID: 7 DC: 01/05 12:25 DM: 01/05 13:20	ID: 8 DC: 01/06 18:40 DM: 01/07 19:00	ID: 9 DC: 01/07 12:23 DM: 01/07 22:12	ID: 10 DC: 01/08 19:30 DM: 01/09 05:21

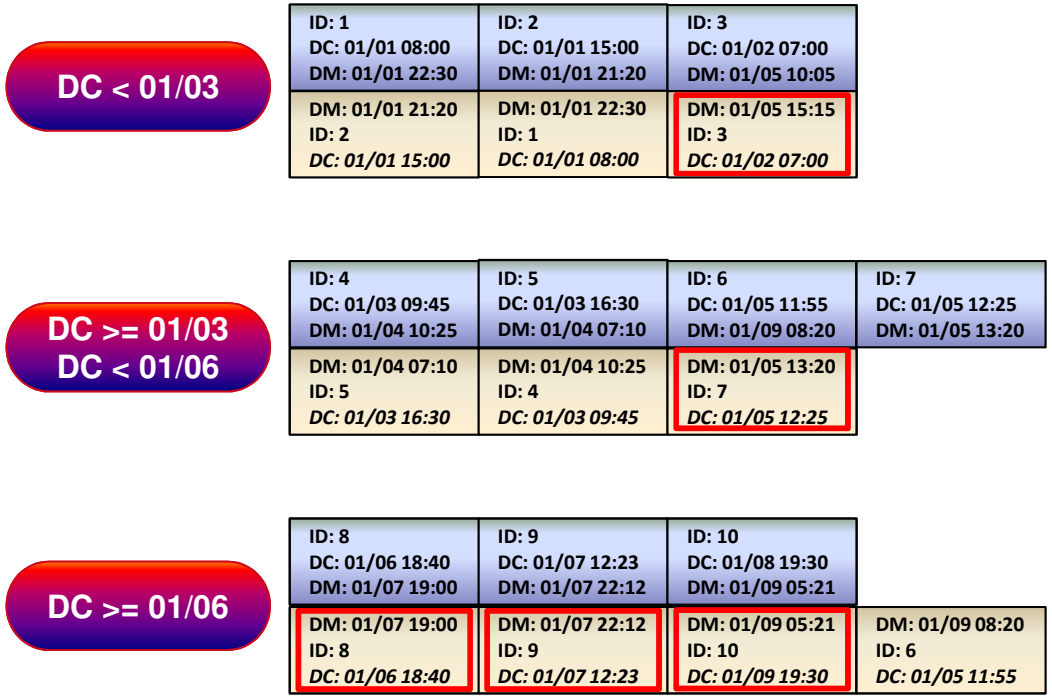
```
create nonclustered index IDX_NCI
on dbo.Data(DateModified)
include(DateCreated)
```

```
select top 3 *
from dbo.Data
where DateModified > '2012-01-05'
order by DateModified, ID
```

DM: 01/01 21:20 ID: 2 *DC: 01/01 15:00*	DM: 01/01 22:30 ID: 1 *DC: 01/01 08:00*	DM: 01/04 07:10 ID: 5 *DC: 01/03 16:30*	DM: 01/04 10:25 ID: 4 *DC: 01/03 09:45*	DM: 01/05 13:20 ID: 7 *DC: 01/05 12:25*	DM: 01/05 15:15 ID: 3 *DC: 01/02 07:00*	DM: 01/07 19:00 ID: 8 *DC: 01/06 18:40*	DM: 01/07 22:12 ID: 9 *DC: 01/07 12:23*	DM: 01/09 05:21 ID: 10 *DC: 01/08 19:30*	DM: 01/09 08:20 ID: 6 *DC: 01/05 11:55*

Partitioned Tables

```
select top 3 *
from dbo.Data
where DateModified > '2012-01-05'
order by DateModified, ID
```



Partitioned Tables

```
select top 3 *
from dbo.Data
where DateModified > '2012-01-05'
order by DateModified, ID
```

DC < 01/03

ID: 1 DC: 01/01 08:00 DM: 01/01 22:30	ID: 2 DC: 01/01 15:00 DM: 01/01 21:20	ID: 3 DC: 01/02 07:00 DM: 01/05 10:05
DM: 01/01 21:20 ID: 2 *DC: 01/01 15:00*	DM: 01/01 22:30 ID: 1 *DC: 01/01 08:00*	DM: 01/05 15:15 ID: 3 *DC: 01/02 07:00*

DC >= 01/03
DC < 01/06

ID: 4 DC: 01/03 09:45 DM: 01/04 10:25	ID: 5 DC: 01/03 16:30 DM: 01/04 07:10	ID: 6 DC: 01/05 11:55 DM: 01/09 08:20	ID: 7 DC: 01/05 12:25 DM: 01/05 13:20
DM: 01/04 07:10 ID: 5 *DC: 01/03 16:30*	DM: 01/04 10:25 ID: 4 *DC: 01/03 09:45*	DM: 01/05 13:20 ID: 7 *DC: 01/05 12:25*	

DC >= 01/06

ID: 8 DC: 01/06 18:40 DM: 01/07 19:00	ID: 9 DC: 01/07 12:23 DM: 01/07 22:12	ID: 10 DC: 01/08 19:30 DM: 01/09 05:21	
DM: 01/07 19:00 ID: 8 *DC: 01/06 18:40*	DM: 01/07 22:12 ID: 9 *DC: 01/07 12:23*	DM: 01/09 05:21 ID: 10 *DC: 01/09 19:30*	DM: 01/09 08:20 ID: 6 *DC: 01/05 11:55*

DM: 05/01 13:20 ID: 7 *DC: 05/01 12:25*	DM: 05/01 15:15 ID: 3 *DC: 02/01 07:00*	DM: 07/01 19:00 ID: 8 *DC: 06/01 18:40*	DM: 07/01 22:12 ID: 9 *DC: 07/01 12:23*	DM: 09/01 05:21 ID: 10 *DC: 08/01 19:30*

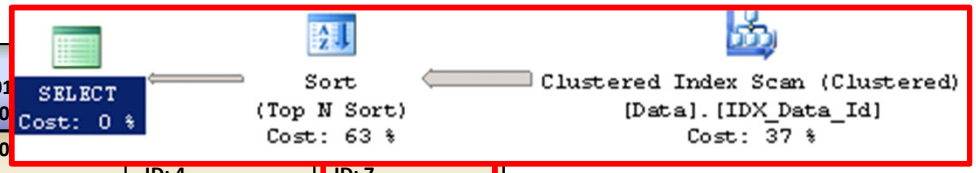
Partitioned Tables

```
select top 3 *
from dbo.Data
where DateModified > '2012-01-05'
order by DateModified, ID
```

DC < 01/03

ID: 1 DC: 01/01 08:00 DM: 01/01 22:30	ID: 2 DC: 01/01 15:00 DM: 01/01 21:20	ID: 3 DC: 01/02 07:00 DM: 01/05 10:05
DM: 01/01 21:20 ID: 2 *DC: 01/01 15:00*	DM: 01/01 22:30 ID: 1 *DC: 01/01 08:00*	DM: 01/05 15:15 ID: 3 *DC: 01/02 07:00*

DC >= 01/03
DC < 01/06



ID: 4 DC: 01 DM: 0		
DM: 0 ID: 5 *DC: 01/03 16:30*	ID: 4 *DC: 01/03 09:45*	ID: 7 *DC: 01/05 12:25*

DC >= 01/06

ID: 8 DC: 01/06 18:40 DM: 01/07 19:00	ID: 9 DC: 01/07 12:23 DM: 01/07 22:12	ID: 10 DC: 01/08 19:30 DM: 01/09 05:21	
DM: 01/07 19:00 ID: 8 *DC: 01/06 18:40*	DM: 01/07 22:12 ID: 9 *DC: 01/07 12:23*	DM: 01/09 05:21 ID: 10 *DC: 01/09 19:30*	DM: 01/09 08:20 ID: 6 *DC: 01/05 11:55*

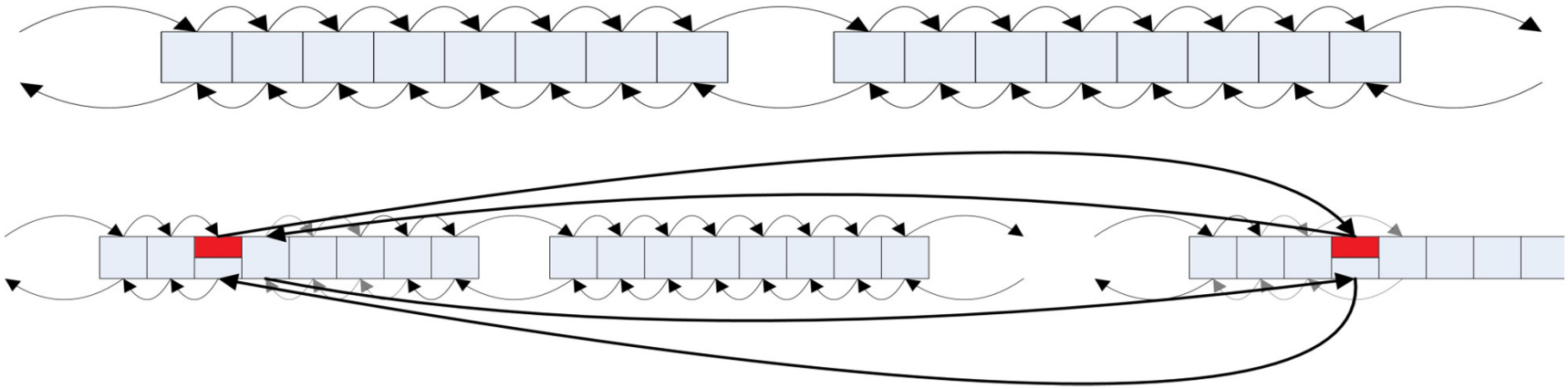
Partitioned Tables

```
;with Boundaries(boundary_id)
as
(
    select Num
    from (values(1),(2),(3),(4),(5),(6)) N(Num)
)
,Top100(ID, DateModified)
as
(
    select top 100 d.ID, d.DateModified
    from
        Boundaries b
            cross apply
            (
                select top 100 ID, DateModified
                from dbo.Data
                where
                    DateModified > @ModDate and
                    $Partition.pfData(DateCreated)   = b.boundary_id
                order by DateModified, ID
            ) d
    order by DateModified, ID
)
select d.*
from dbo.Data d join Top100 a on d.Id = a.Id
order by d.DateModified, d.ID
```

- Some cases can be optimized with *$Partition* function
- Download demo scripts: http://aboutsqlserver.com/presentations

Fragmentation

Page Split



- SQL Server moves about 50% of the data to another page
- FILLFACTOR allows to reserve some space on the page
  - Works only on CREATE INDEX/ALTER INDEX REBUILD stage
  - Decreases Page Splits / Fragmentation but increases index size

Patterns increasing fragmentation

- Insert/Update – increasing size of the row after insert
- Read Committed Snapshot / Snapshot
- Indexes on “Truly Random” values
  - Uniqueidentifier
  - Hashbytes

Fragmentation and System

- Fragmentation affects system during SCANS
  - Data Warehouse / Decision Support Systems
  - Poorly optimized Queries
- Fragmentation has less affect during index SEEK
  - OLTP
- Fragmentation decreases performance of batch operations
- Smaller percent of space usage on the page increases size of the index and uses more memory in the Buffer Pool.

Index Maintenance

- sys.dm_db_index_physical_stats
  - **avg_fragmentation_in_percent**
  - **avg_page_space_used_in_percent**
- ALTER INDEX..REBUILD
  - Recreate the index (and statistics)
  - Can be online operation with Enterprise Edition
  - Recommended(*) if fragmentation > 30%
- ALTER INDEX..REORGANIZE
  - Logical defragmentation
  - Online operation
  - Recommended(*) if fragmentation between 5% and 30%

(*) Books online

Factors to consider designing Maintenance Plan

- System blueprint (OLTP, DW, Mixed)
- Is system running 24x7x365?
- SQL Server Edition – can I use online index rebuild
- Are there other DBs on the server?
- What HA/DR solutions are in place?

Indexing Strategies

Clustered Index

- Primary Key != Clustered Index
  - Primary key → logical design concept
  - Clustered Index → physical design concept
- Ideal Clustered index
  - Reduces Page Splits / Fragmentation
  - Optimizes most important queries

Clustered Index

- Ideally Clustered Index would be
  - Static
    - Data moved to the different place in the data file and all NCI are updated when CI key changed
  - Narrow
    - Data present in every NCI key
  - Unique
    - For non-unique clustered indexes SQL Server adds 4 bytes *uniquifier int null* column
      - Real overhead is between 0 and 8 bytes

Identity / Uniqueidentifiers

- Unique, Static, Narrow
- Potential problems:
  - Hot Spots – serialization during page/extent allocations
  - Rarely helps to optimize queries
- Use cases
  - Referenced tables (Articles, Customers, etc.)
  - No other candidates for CI (in some cases)
- Identity – 4/8 bytes. Uniqueidentifiers – 16 bytes
  - NEWID() – produces fragmentation

Logical Partitioning

```
create table dbo.Positions
(
    CompanyId int not null,
    UTCTimeTag datetime2(0) not null,
    RecId bigint not null,
    DeviceId int not null,
    Latitude decimal(9,6) not null,
    Longitude decimal(9,6) not null
)
go

create unique clustered index IDX_CI
on dbo.Positions
(CompanyId, UTCTimeTag, RecId)
go

create nonclustered index IDX_NCI
on dbo.Positions
(CompanyId, DeviceId, UTCTimeTag)
```

- Leftmost column identifies partition
  - Localizes I/O within the partition
  - Could reduce fragmentation in case if right index columns monotonously increase
- Downsides
  - Potential issues with statistics

Nonclustered indexes - Uniqueness

- Unique Constraint != Unique Index
  - Unique Constraint → Logical design concept
  - Unique Index → Physical design concept
  - Internally SQL Server uses unique index in both cases
- Unique index has more effective structure
  - Intermediate and Root levels don't include *row id*
- Uniqueness helps Query Optimizer

Nonclustered indexes - Usage

- SQL Server rarely uses more than one nonclustered index per table in the same statement (index intersection)
- Composite index with included columns usually more effective than set of narrow one column indexes
  - Don’t forget about data modification and maintenance cost

Nonclustered indexes
How many indexes per table?

- OLTP systems
  - Many short transactions running simultaneously
  - Data updates all the time
  - Simple and optimized queries
    - Minimally required set of indexes
- Data Warehouse systems
  - Small number of simultaneous complex queries (usually scans)
  - Data updated based on schedule
    - Number of indexes depend on the queries and update schedule
      - Dropping and re-creating indexes during data update
      - ColumnStore indexes

Keys greater than 900 bytes

- Index on persisted calculated column (CHECKSUM, HASHBYTES)
- Always add original column to *where* clause

Indexed Views

- Regular Views – Metadata Only
- Indexed Views – Data materialized similarly to the table
- Potential issues:
  - A lot of restrictions
  - Maintenance cost
  - Behavior varies based on SQL Server edition
    - Standard Edition requires (NOEXPAND) hint
- Use Cases
  - Aggregation (Data Warehouse)
  - Join elimination
  - Vendors code optimization (Enterprise Edition only)

Optimization Strategies

Optimization Strategies
New Systems

- Step 1 – creation of *minimally required* set of indexes:
  - Primary keys and clustered indexes
  - Uniqueness support (unique constraints or indexes)
  - Referential integrity support
  - Indexes for most important queries (when known)

Optimization Strategy
Existing Systems

- Step 1
  - Removing redundant indexes
  - Removing unused indexes
  - Trivial consolidation
  - Analysis of problematic indexes

Optimization Strategy
Existing Systems

- Step 1
  - **Removing redundant indexes**
  - Removing unused indexes
  - Trivial consolidation
  - Analysis of problematic indexes

```
create nonclustered index IDX_1
on dbo.Customers(LastName, FirstName)
```

```
create nonclustered index IDX_2
on dbo.Customers(LastName)
```

- There are always exceptions to the rule
  - Sometimes key size does matter

Optimization Strategy
Existing Systems

- Step 1
  - Removing redundant indexes
  - **Removing unused indexes**
  - Trivial consolidation
  - Analysis of problematic indexes

```
select *
from sys.dm_db_index_usage_stats us
where
    us.database_id = DB_ID() and
    us.object_id = OBJECT_ID(N'dbo.Orders')
order by
    us.index_id
```

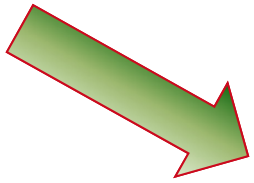
index_id	user_seeks	user_scans	user_lookups	user_updates	last_user_seek	l
1	27341	2578	28123	9921	2012-03-23 15:27:10.270	:
2	5368	0	0	299	2012-03-23 15:26:55.697	I
3	302	0	0	46	2012-03-23 15:21:06.557	I
5	1677	0	0	49	2012-03-23 15:27:05.317	I
8	21065	0	0	49	2012-03-23 15:27:08.737	I
50	0	0	0	28	NULL	I
51	0	2382	0	60	NULL	:

- sys.dm_db_index_operational_stats – more info
- Data clears when server restarts

Optimization Strategy
Existing Systems

- Step 1
  - Removing redundant indexes
  - Removing unused indexes
  - **Trivial consolidation**
  - Analysis of problematic indexes

```
create nonclustered index IDX_1
on dbo.Customers(LastName, FirstName)
include(DateOfBirth)

create nonclustered index IDX_2
on dbo.Customers(LastName)
include(Phone)
```



```
create nonclustered index IDX_3
on dbo.Customers(LastName, FirstName)
include(Phone, DateOfBirth)
```

Optimization Strategy
Existing Systems

- Step 1
  - Removing redundant indexes
  - Removing unused indexes
  - **Trivial consolidation**
  - Analysis of problematic indexes

```
create nonclustered index IDX_4
on dbo.Orders(OrderDate, Total)

create nonclustered index IDX_5
on dbo.Orders(OrderDate, WarehouseId)
```

```
create nonclustered index IDX_6
on dbo.Orders(OrderDate, Total)
include(WarehouseId)
```

```
create nonclustered index IDX_7
on dbo.Orders(OrderDate, WarehouseId)
include(Total)
```

Optimization Strategy
Existing Systems

- Step 1
  - Removing redundant indexes
  - Removing unused indexes
  - Trivial consolidation
  - **Analysis of problematic indexes**

```
select *
from sys.dm_db_index_physical_stats(DB_ID(),Object_Id(N'dbo.Orders'),null, null, 'DETAILED')
```

index_id	avg_fragmentation_in_percent	fragment_count	avg_fragment_size_in_pages	page_count	avg_page_space_used_in_percent
3	98.5034013605442	732	1.00409836065574	735	64.8512972572276
51	3.53982300884956	19	23.7894736842105	452	97.8963800345935

Optimization Strategy

Existing Systems

- Step 1
  - Removing redundant indexes
  - Removing unused indexes
  - Trivial consolidation
  - **Analysis of problematic indexes**

```
select *
from sys.dm_db_index_operational_stats(DB_ID(),Object_Id(N'dbo.Users'),null, null)
```

index_id	leaf_insert_count	leaf_update_count	singleton_lookup_count	range_scan_count	row_lock_count	page_io_latch_wait_count	page_io_latch_wait_in_ms
1	18	27457	326612	0	88905	172	2782
2	307	0	0	58715693	1158	207	2075
3	18	0	304	0	36	17	673
5	23	0	0	1767	51	9	329
8	20	3	1471	257	148	11	516
50	0	0	0	0	0	0	0
51	125	0	0	0	357	4	62
100	307	0	0	42526	903	7	514

Optimization Strategy

- Pinpoint problematic code
- Analysis of problematic code
- Adding new indexes

Optimization Strategy

- **Pinpoint problematic code**
  - Data Management View
  - SQL Profiler
  - Performance Data Collectors / MDW
  - 3rd Party tools
  - Etc.
- Analysis of problematic code
- Adding new indexes

sys.dm_exec_query_stats

```
SELECT TOP 50
    SUBSTRING(qt.TEXT, (qs.statement_start_offset/2)+1,
        ((
            CASE qs.statement_end_offset
                WHEN -1 THEN DATALENGTH(qt.TEXT)
                ELSE qs.statement_end_offset
            END - qs.statement_start_offset)/2)+1) as [SQL],
    qs.execution_count,
    (qs.total_logical_reads + qs.total_logical_writes)
        / qs.execution_count as [Avg IO],
    qp.query_plan,
    qs.total_logical_reads, qs.last_logical_reads,
    qs.total_logical_writes, qs.last_logical_writes,
    qs.total_worker_time,
    qs.last_worker_time,
    qs.total_elapsed_time/1000 total_elapsed_time_in_ms,
    qs.last_elapsed_time/1000 last_elapsed_time_in_ms,
    qs.last_execution_time
FROM
    sys.dm_exec_query_stats qs
        OUTER APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
        OUTER APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
ORDER BY
    [Avg IO] DESC
```

	SQL	execution_c...	Avg IO	query_plan	total_logical_reads	last_logical_reads	total_logical_writes	last_logical_writes	total_worker_time	l
1	select top 1000 RecId, UtcTimeTag,...	1	1324346	<ShowPlanXML xmlns="http:/...	1324346	1324346	0	0	2254882	2
2	select NEWID() as ID, p.COMPA...	3	218543	<ShowPlanXML xmlns="http:/...	655631	218544	0	0	37335939	1
3	select count(*) from GPS.V$NEX...	1	191254	<ShowPlanXML xmlns="http:/...	191254	191254	0	0	2510740	2
4	with MinMax (MinRec, MaxRec) as ...	1	147437	<ShowPlanXML xmlns="http:/...	147435	147435	2	2	1310547	1
5	with MinMax (MinRec, MaxRec) as ...	1	118663	<ShowPlanXML xmlns="http:/...	118663	118663	0	0	772461	1

Optimization Strategy

- Pinpoint problematic code
- **Analysis of problematic code**
  - Can code be simplified and/or refactored?
  - Is Statistics up to date?
  - Does recompilation help (parameter sniffing)?
- Adding new indexes

Tools

- Missing Indexes DMVs
  - Recommend only indexes for *specific* query and *specific* plan
  - Do not consider any other factors
- Database Tuning Advisor
  - Quality largely depend on what is in the trace
- Could help to pinpoint the problems (as the helper tools)

Q&A

- Thank you very much for attending!

- Email: dmitri@aboutsqlserver.com
- Blog: http://aboutsqlserver.com
- Demo scripts:
  - SQL Rally Web Site
  - http://aboutsqlserver.com/presentations

PASS