Intro Into Index Analysis

Dmitri Korotkevitch
https://aboutsqlserver.com

Hello!

20+ years of experience in IT
15+ years of experience working with SQL Server
Microsoft Data Platform MVP
Microsoft Certified Master
Author

- Advanced SQL Server Troubleshooting and Performance Tuning
- Pro SQL Server Internals (v1-2)
- Expert SQL Server In-Memory OLTP (v1-2)
- Expert SQL Server Transactions and Locking

Blog: https://aboutsqlserver.com
Email: dk@aboutsqlserver.com
Slides / Code: https://github.com/aboutsqlserver/code
YouTube: AboutSQLServer

Agenda

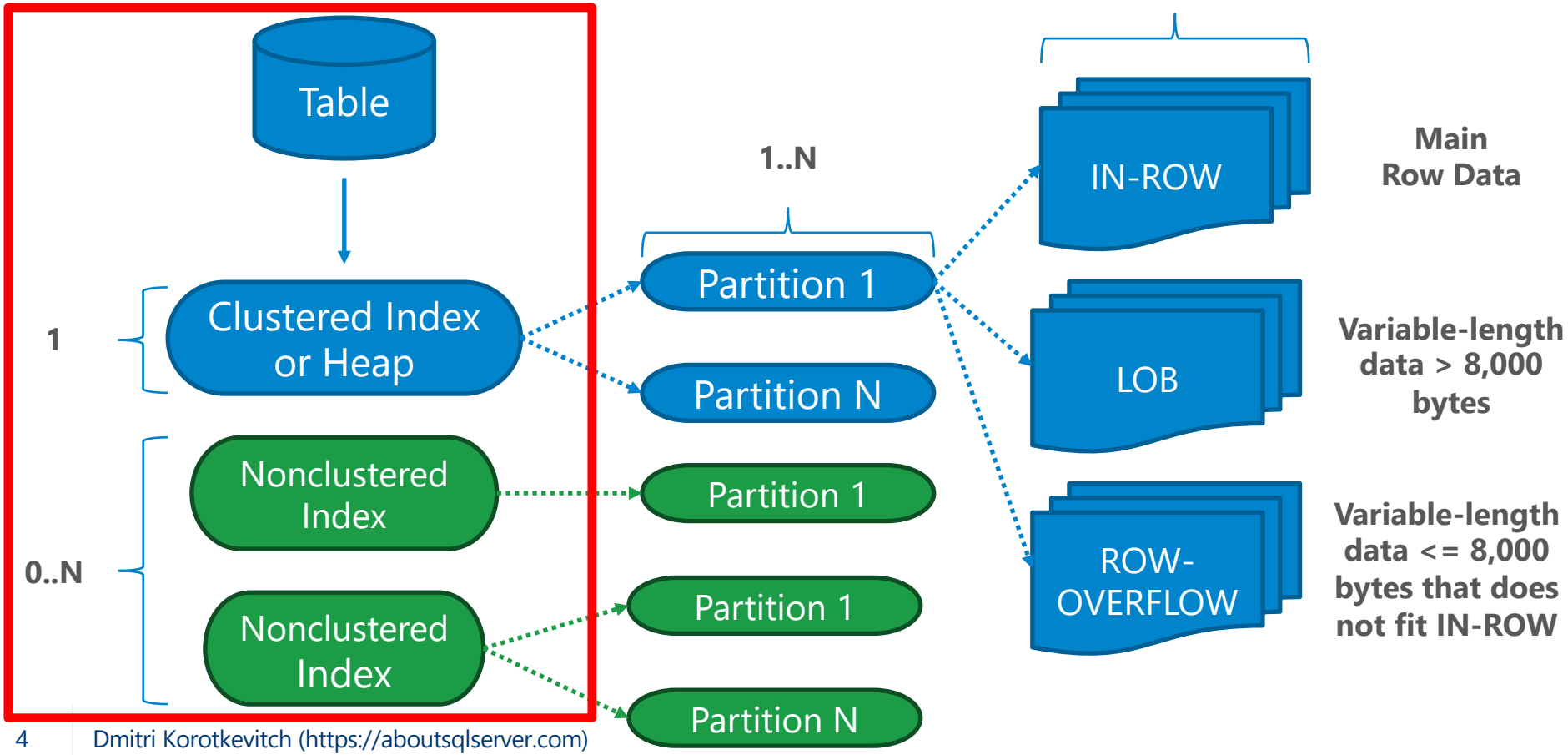
Data Storage 101

- B-Tree Format
- Access Patterns

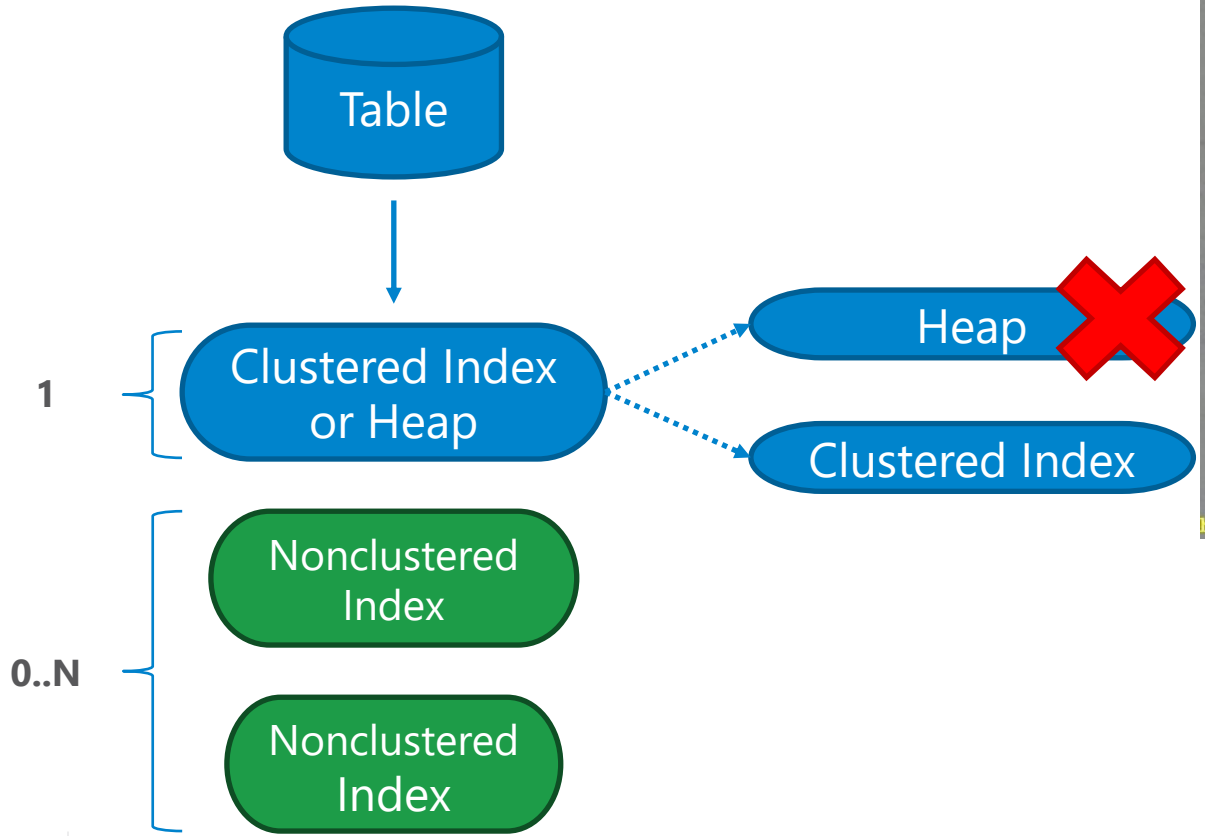
Index Analysis Techniques

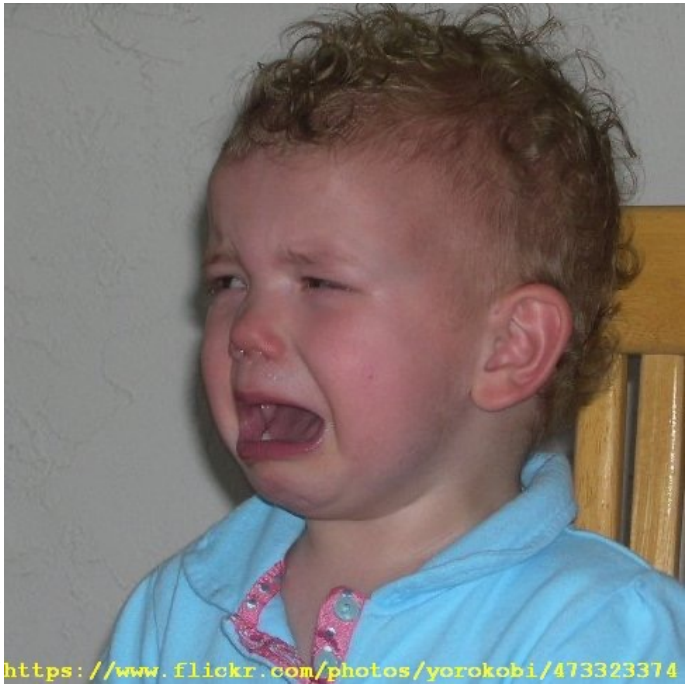
- Catalog views
- Usage and Operational statistics
- Physical statistics

Data Storage 101



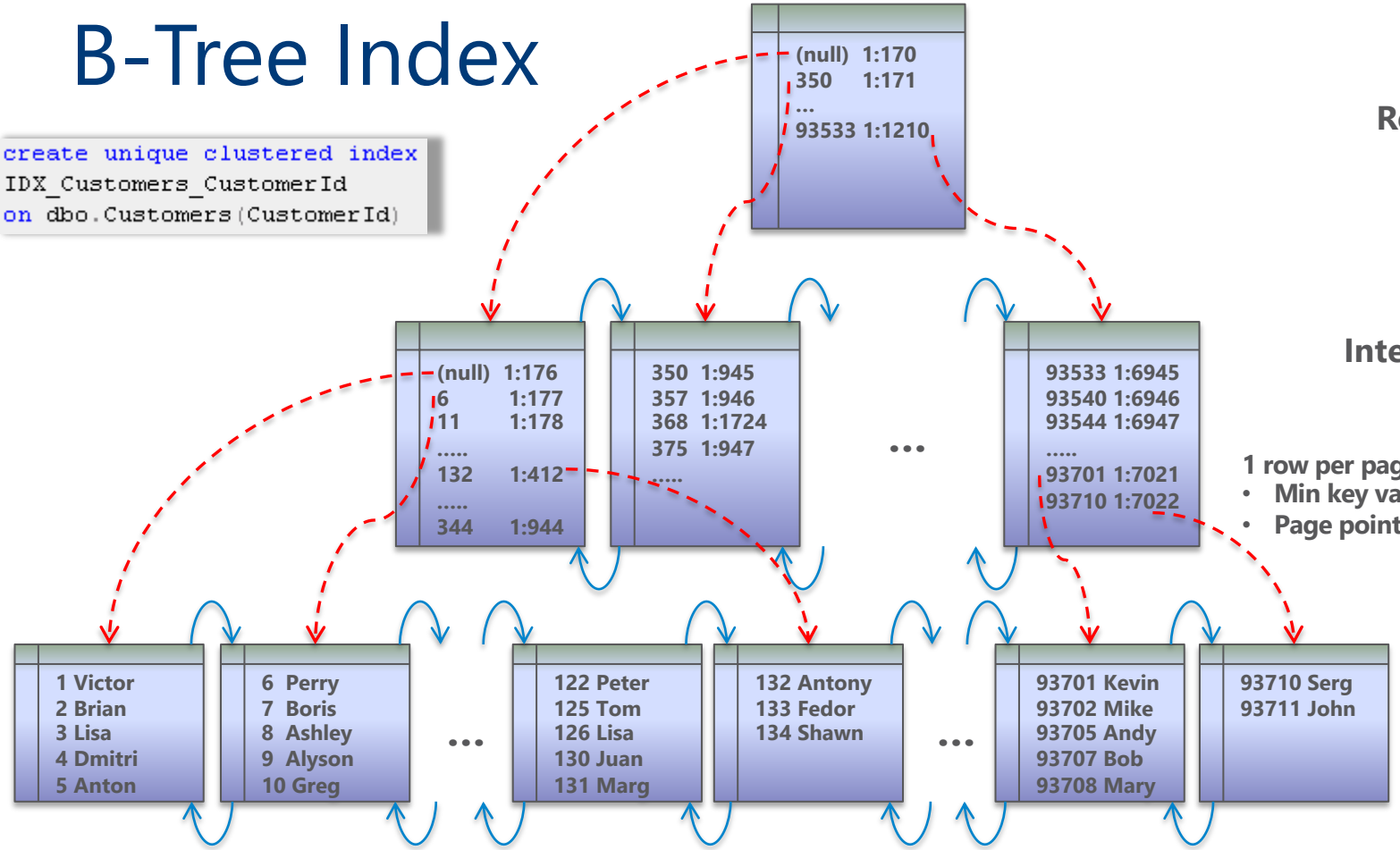
Data Storage 101





B-Tree Index

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



Root Level

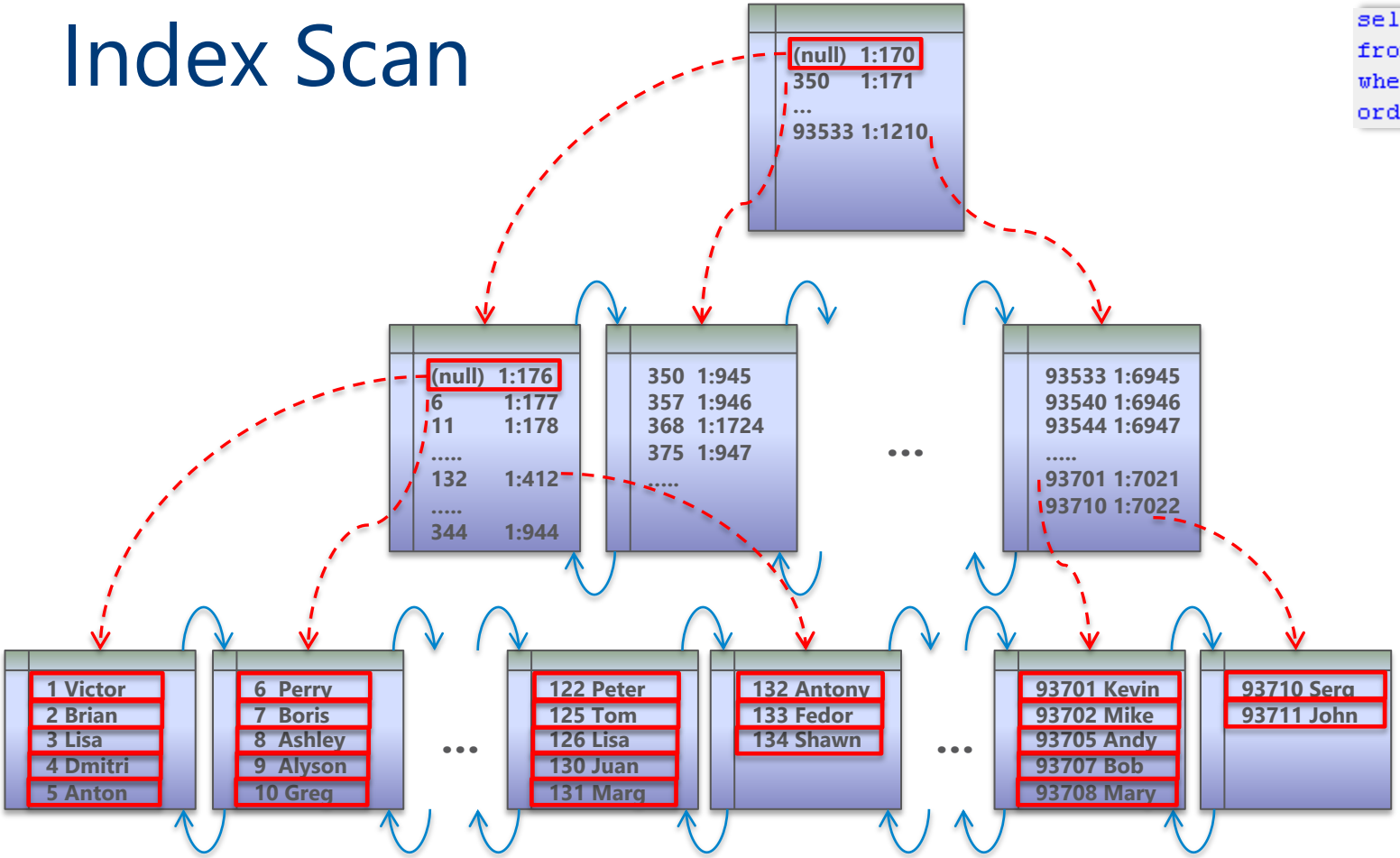
Intermediate Level

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

Leaf Level (actual data)

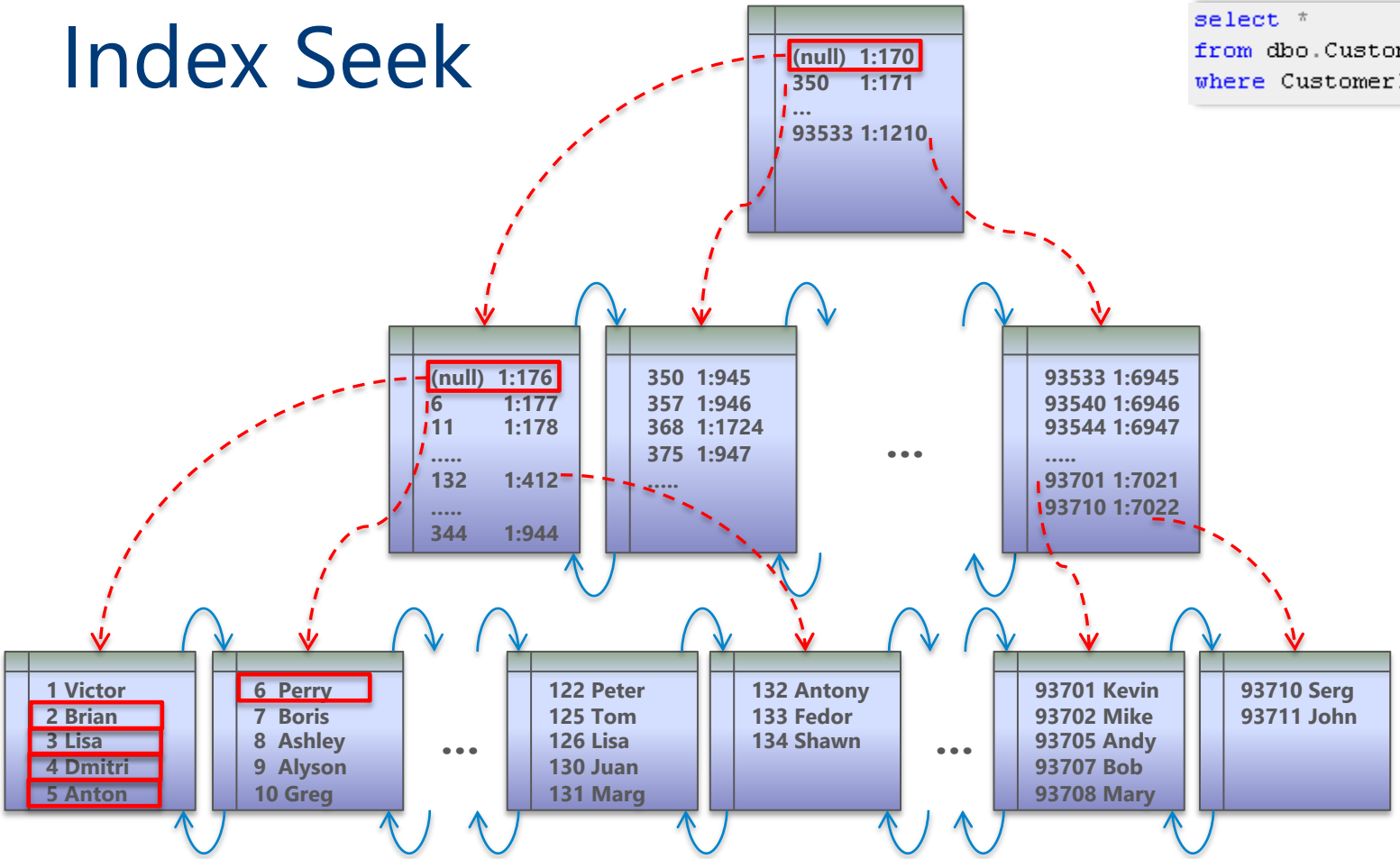
Index Scan

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



Index Seek

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



Seeks and Scans

Demo

Patterns to Avoid

Functions

Substring search

- Prefix search is OK

Data type conversion

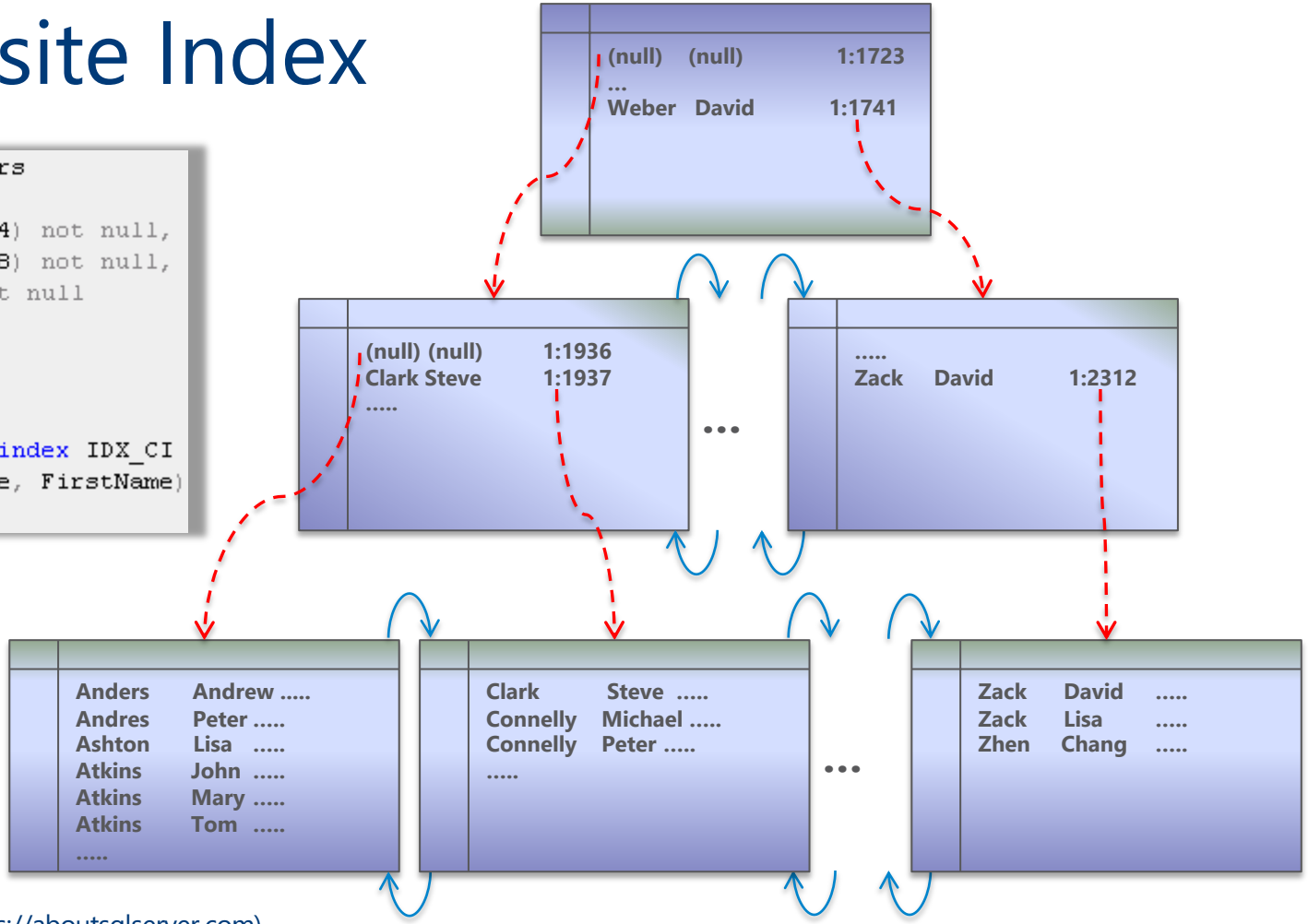
- Unicode parameters for varchar columns (some collations)

Foreign keys w/o indexes on *referencing* “detail” tables

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
```



Composite Indexes

```
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
```

- SARGable when query has SARGable predicates on leftmost column(s) of the index:

```
where LastName = N'Doe'
    and FirstName = N'John'

where LastName = N'Doe'
```

- Non-SARGable – all other cases:

```
where LastName = N'%oe%'
    and FirstName = N'John'

where FirstName = N'John'
```

Redundant Indexes

Look at the indexes with the same left-most column(s)

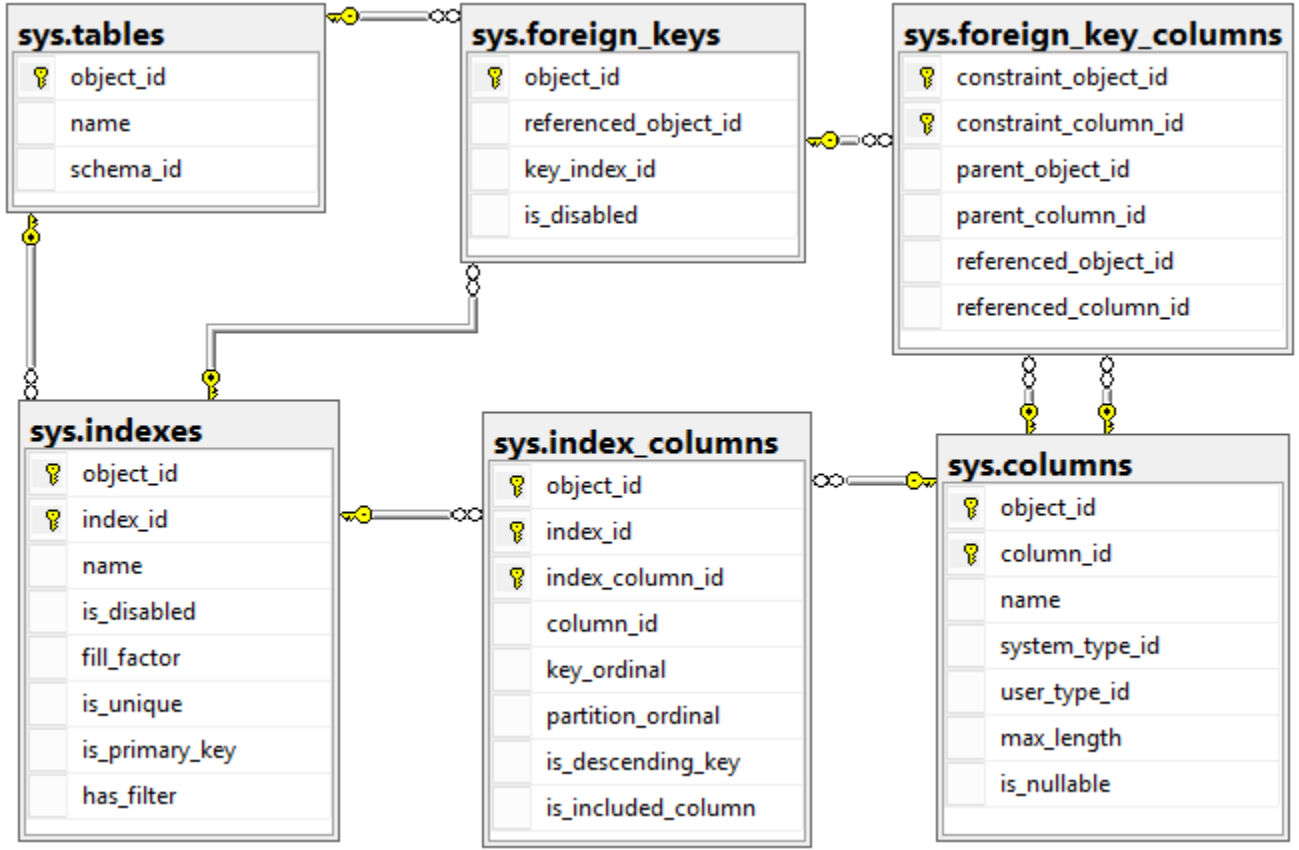
```
create nonclustered index IDX_1
on dbo.Customers(LastName, FirstName)
```

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

Multi-Tenant Systems:

- Make *TenantId* as the left-most column
- Factor it to redundant index analysis

Tales Told By Catalog Views



- Heap tables
- Redundant indexes
- Non-indexed FKs
- Wide CI
- Non-unique CI
- Uniqueidentifiers
- Untrusted FKs
- Suboptimal FILLFACTOR

Just be creative ☺

Catalog Views - Ideas

Heap tables

- sys.indexes (index_id = 0)

Redundant indexes

- Self-join sys.index_columns (=object_id, >index_id, =column_id, key_ordinal=1)
- Multi-tenant systems – key_ordinal > 1

Non-indexed foreign keys

- sys.foreign_key_columns (parent_object_id, parent_column_id, constraint_column_id) IN sys.index_columns(object_id, index_id, key_ordinal)

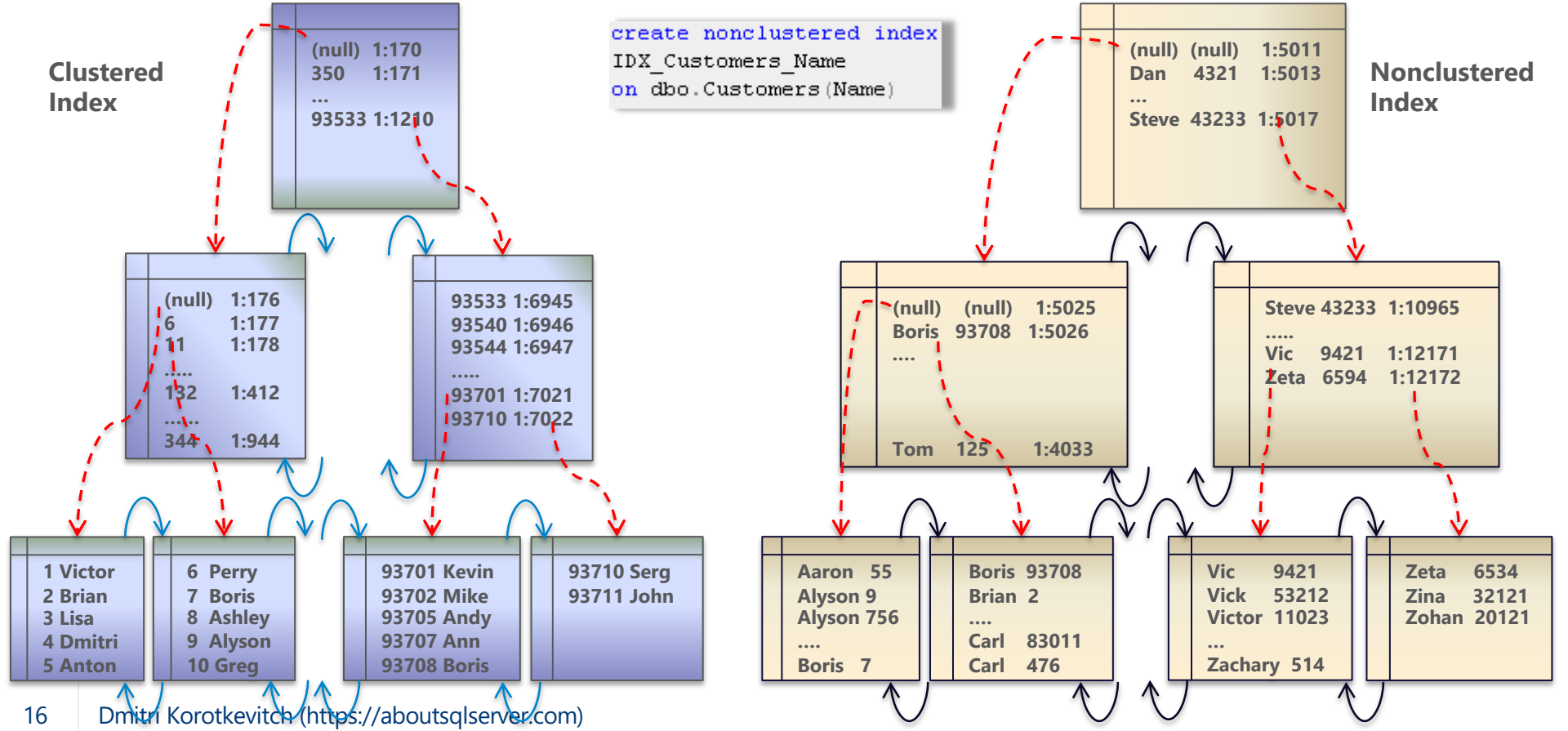
Uniqueidentifiers

- sys.index_columns (key_ordinal = 1) -> sys.columns.system_type_id =36

Wide clustered indexes

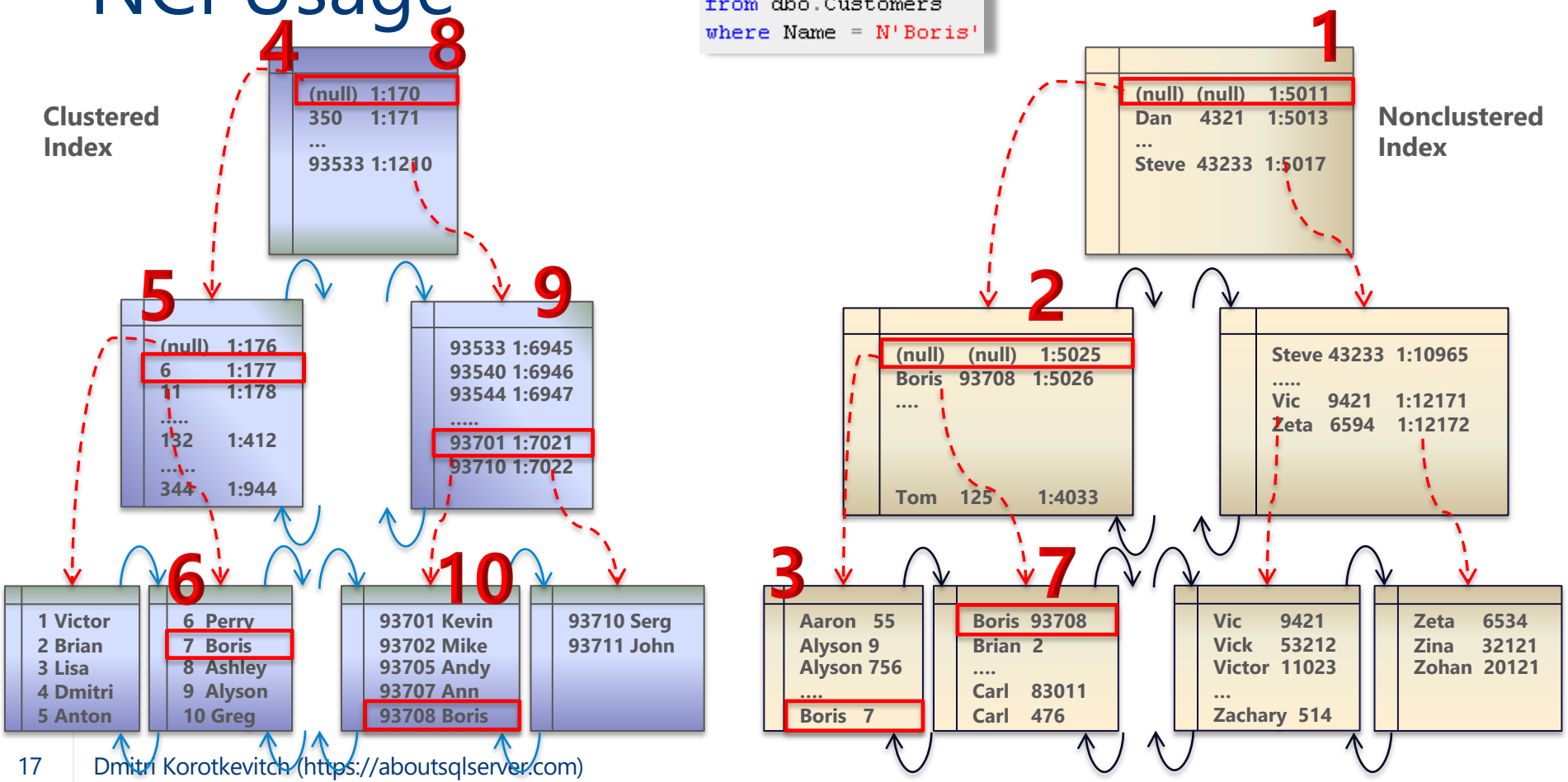
- sys.indexes (index_id = 1) -> SUM(sys.index_columns -> sys.columns.max_length)
- max_length for (MAX) columns =-1

Nonclustered Index



NCI Usage

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



NCI Usage

Demo

sys.dm_db_index_usage_stats

How often index appears in execution plans

- Seeks: Index Seek operation
- Scans: Index Scan operation
- Lookup: Key Lookup or RID Lookup operations
- Reads: Seeks + Scans + Lookups
- Updates: INSERT + UPDATE + DELETE

	Table	Index	Seeks	Scans	Lookups	Reads	Updates	Last Seek	Last Scan
1	[redacted]		74567916	0	755160	75323076	1510669	2016-11-18 07:32:15.437	NULL
2			57940278	0	0	57940278	1510669	2016-11-18 07:32:15.537	NULL
3			16724	0	0	16724	1510669	2016-11-18 07:00:20.827	NULL
4			1003195	1090508	0	2093703	1136428	2016-11-18 07:32:08.700	2016-11-1
5			544501	0	0	544501	1510669	2016-11-18 07:32:10.540	NULL
6			1	0	0	1	680021	2016-11-02 14:17:28.843	NULL
7			5210970	0	0	5210970	1510669	2016-11-18 07:32:14.423	NULL
8			0	0	0	0	1510669	NULL	NULL
9			7049584	0	0	7049584	2403273	2016-11-18 07:32:14.423	NULL
10			377572	0	0	377572	2403273	2016-11-18 07:32:08.750	NULL

sys.dm_db_index_operational_stats

Access methods, I/O, locking, latching activity

- Insert, Update, Delete counts (# of rows)
- singleton_lookup_count: Single-row Index Seek operations
- range_scan: Index Seeks on the range of rows + Index Scans
- LOB and ROW_OVERFLOW statistics (potential SELECT * patterns)
- Lock counts and waits on row- and page-levels (good for lock escalation analysis)
- Page latch count and waits
- Page IO latch count and waits
- And more..

	index_id	Table	Index	range_scan_count	singleton_lookup_count	row_lock_wait_in_ms	page_latch_wait_in_ms	page_io_latch_wait_in_ms
1	1	[redacted]	[redacted]	1411162	3638897399	0	71286	13634302
2	2	[redacted]	[redacted]	774	0	0	338	283589
3	3	[redacted]	[redacted]	760095	0	0	121	329284
4	4	[redacted]	[redacted]	32726	0	0	828	7878183
5	5	[redacted]	[redacted]	358	0	0	66	3138358
6	6	[redacted]	[redacted]	21	0	0	124	138602
7	8	[redacted]	[redacted]	0	0	0	519	68238
8	33	[redacted]	[redacted]	1	0	0	60	574234
9	34	[redacted]	[redacted]	11012	0	0	490	626016
10	35	[redacted]	[redacted]	0	0	0	126	23540

Usage and Operational Stats

Demo

Pattern: Inefficient Reads

High number of Scans in sys.dm_db_index_usage_stats

	IndexId	Table	Index	Seeks	Scans	Lookups	Updates
1	1			148266434	1500	251782	78441173
2	4			1207312	0	0	48080643
3	5			0	1354122	0	35993284
4	6			73969314	3	0	57187072
5	8			33931425	0	0	35589892
6	3			23019	2134296	0	70957328

Analyze index usefulness (# of seeks, operational overhead)
Find and optimize queries that use the index

Analyzing Plan Cache

Demo

Pattern: High Maintenance Cost

High Update vs. Read ratio

	IndexId	Table	Index	Seeks	Scans	Lookups	Reads	Updates
1	1			1	9	1121468	1121478	5243411
2	16			8	1	0	9	4940468
3	17			40023	0	0	40023	4940468
4	19			0	0	0	0	5243411
5	23			1089497	0	0	1089497	4940468

Drop unused indexes
Analyze index usefulness (# of seeks, access time, operational overhead)
Find and optimize queries that use the index

Pattern: Inefficient Clustered Indexes

High Lookup vs. Seek ratio

- Typical pattern: Clustered PRIMARY KEY constraint on identity column

	IndexId	Table	Index	Seeks	Scans	Lookups	Reads	Updates
1	1			1	9	1121468	1121478	5243411
2	16			8	1	0	9	4940468
3	17			40023	0	0	40023	4940468
4	19			0	0	0	0	5243411
5	23			1089497	0	0	1089497	4940468

Change clustered index in the table

Pattern: Noncovering NCIs

High Lookup vs. Seek ratio

- Typical pattern: Frequently executed queries use noncovered nonclustered index

	IndexId	Table	Index	Seeks	Scans	Lookups	Reads	Updates
1	1			148266434	1500	251782608	400050542	78441173
2	4			1207312	0	0	1207312	48080643
3	5			988005	1354122	0	988005	35993284
6	35			26734187	0	0	26734187	31211137
7	39			121404098	2	0	121404100	44104235
8	54			9075825	0	0	9075825	31210092
9	57			14180	0	0	14180	7537756
11	59			21019386	5319	0	21024705	48080643
13	61			39117309	6	0	39117315	43700843
14	3			23019	2134296	0	23019039	70957328
15	43			3094705	339090	0	3433795	66895706

Analyze queries with Key Lookups making NCIs covering when possible

Deeper Dive

Index size on disk and in buffer pool

- sys.allocation_units + sys.dm_os_buffer_descriptors (script is included)

Lock and latch contention in the system

- sys.dm_db_index_operational_stats

Disk (PAGEIOLATCH) contention

- sys.dm_db_index_operational_stats + sys.dm_db_index_usage_stats
  - page_io_latch_*
  - Seeks, Scans, singleton_lookup_count, range_scan_count

sp_Index_Analysis

Demo

Caveats

Statistics clears at:

- SQL Server restart
- ALTER INDEX REBUILD in:
  - SQL Server 2012 RTM - SP3 CU2 (Fixed in SP3 CU3)
  - SQL Server 2014 RTM – SP1 (Fixed in CU2)

Remember about Readable Secondaries!

	Server	IndexId	Table	Index	Seeks	Scans	Lookups	Reads	Updates
1	PRIMARY	1			1	9	1121468	1121478	5243411
2	PRIMARY	16			8	1	0	9	4940468
3	PRIMARY	17			40023	0	0	40023	4940468
4	PRIMARY	19			0	0	0	0	5243411

	Server	IndexId	Table	Index	Seeks	Scans	Lookups	Reads	Updates
1	SECONDARY	1			473	18939	257935	277347	0
2	SECONDARY	16			190645	1337	0	191982	0
3	SECONDARY	17			2860	0	0	2860	0
4	SECONDARY	19			173683	0	0	173683	0

sys.dm_db_index_physical_stats

	index_id	partition_number	alloc_unit_type_desc	index_level	page_count	record_count
1	1	1	IN_ROW_DATA	0	651333	12581042
2	1	1	IN_ROW_DATA	1	18746	651333
3	1	1	IN_ROW_DATA	2	202	18746
4	1	1	IN_ROW_DATA	3	1	202

avg_fragmentation_in_percent	avg_page_space_used_in_percent	version_ghost_record_count	forwarded_record_count
5.41	84.98	0	NULL
92.78	14.28	0	NULL
99.50	42.20	0	NULL
0.00	95.71	0	NULL

avg_fragmentation_in_percent: External fragmentation - # of fragments in DB

avg_page_space_used: Internal fragmentation – page space utilization

Tune FILLFACTOR. Do not use 100 with row versioning (RCSI, Snapshot, etc)

sys.dm_db_index_physical_stats

	index_id	partition_number	alloc_unit_type_desc	index_level	page_count	record_count	avg_fragmentation_in_percent	avg_page_space_used_in_percent	version_ghost_record_count	forwarded_record_count
1	1	1	IN_ROW_DATA	0	651333	12581042	5.41	84.98	0	NULL
2	1	1	IN_ROW_DATA	1	18746	651333	92.78	14.28	0	NULL
3	1	1	IN_ROW_DATA	2	202	18746	99.50	42.20	0	NULL
4	1	1	IN_ROW_DATA	3	1	202	0.00	95.71	0	NULL

Deferred *Ghost* and *Version Store Cleanup* due to row-versioning

Check long running RCSI/Snapshot transaction including readable secondaries

sys.dm_db_index_physical_stats

	index_id	partition_number	alloc_unit_type_desc	index_level	page_count	record_count
1	1	1	IN_ROW_DATA	0	651333	12581042
2	1	1	IN_ROW_DATA	1	18746	651333
3	1	1	IN_ROW_DATA	2	202	18746
4	1	1	IN_ROW_DATA	3	1	202

avg_fragmentation_in_percent	avg_page_space_used_in_percent	version_ghost_record_count	forwarded_record_count
5.41	84.98	0	NULL
92.78	14.28	0	NULL
99.50	42.20	0	NULL
0.00	95.71	0	NULL

Inefficient Heap tables (high rate of updates)

Rebuild and consider to remove heap tables

Key Points

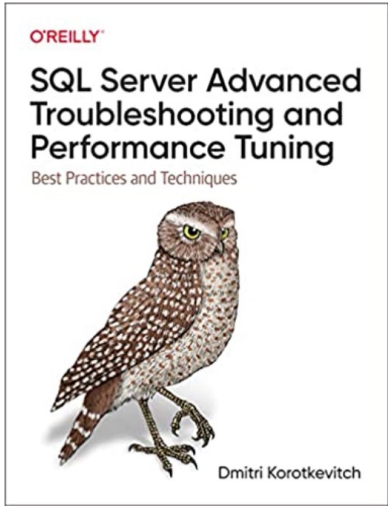
Remember about patterns that prevent *Index Seek*

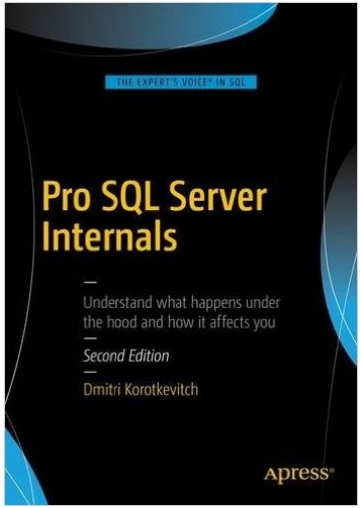
Use catalog views to identify schema issues

Use usage and operational statistics to identify inefficient indexes

Correlate data from multiple sources during analysis

Q&A





Blog: https://aboutsqlserver.com
E-mail: dk@aboutsqlserver.com
Slides / Code: https://github.com/aboutsqlserver/code

Thank You