Adopt, Discover and Improve

Who is this guy with heavy accent?

- Director of Development at Actsoft.
  - I'm responsible for DB Design and Development of OLTP system handling ~2500 TPS during peak time
- 9+ years of experience with Microsoft SQL Server
- MCITP
  - SQL Server Database Developer 2005, 2008
  - SQL Server Database Administrator 2008
- MCPD
  - Enterprise Application Developer
- Blog: http://aboutsqlserver.com
  - Session will be available for download
- Email: dmitri@aboutsqlserver.com

What is it all about?

- **Session goals:**
  - Share the experience
  - Demonstrate the set of techniques that helps to analyze system state and identify performance bottlenecks in **OLTP blueprint systems**
  - Show a few things you need to be aware when designing index strategy on the large databases
  - Demonstrate a few common patterns and anti-patterns
- **What is out of scope:**
  - We don't want to miss after-event party, do we?
  - How to configure and maintain SQL Server instances
  - **Troubleshooting of Data Warehouse / Reporting blueprint systems**
  - Advanced troubleshooting and monitoring techniques
    - Extended events
    - Management Data Warehouse

System Blueprints

OLTP

- High volume of small identical transactions
- Queries are identical
- Data constantly changed

Data Warehouse

- Low volume of large complex transactions
- Complex and different queries
- Data rarely changes
  - Batch refreshes

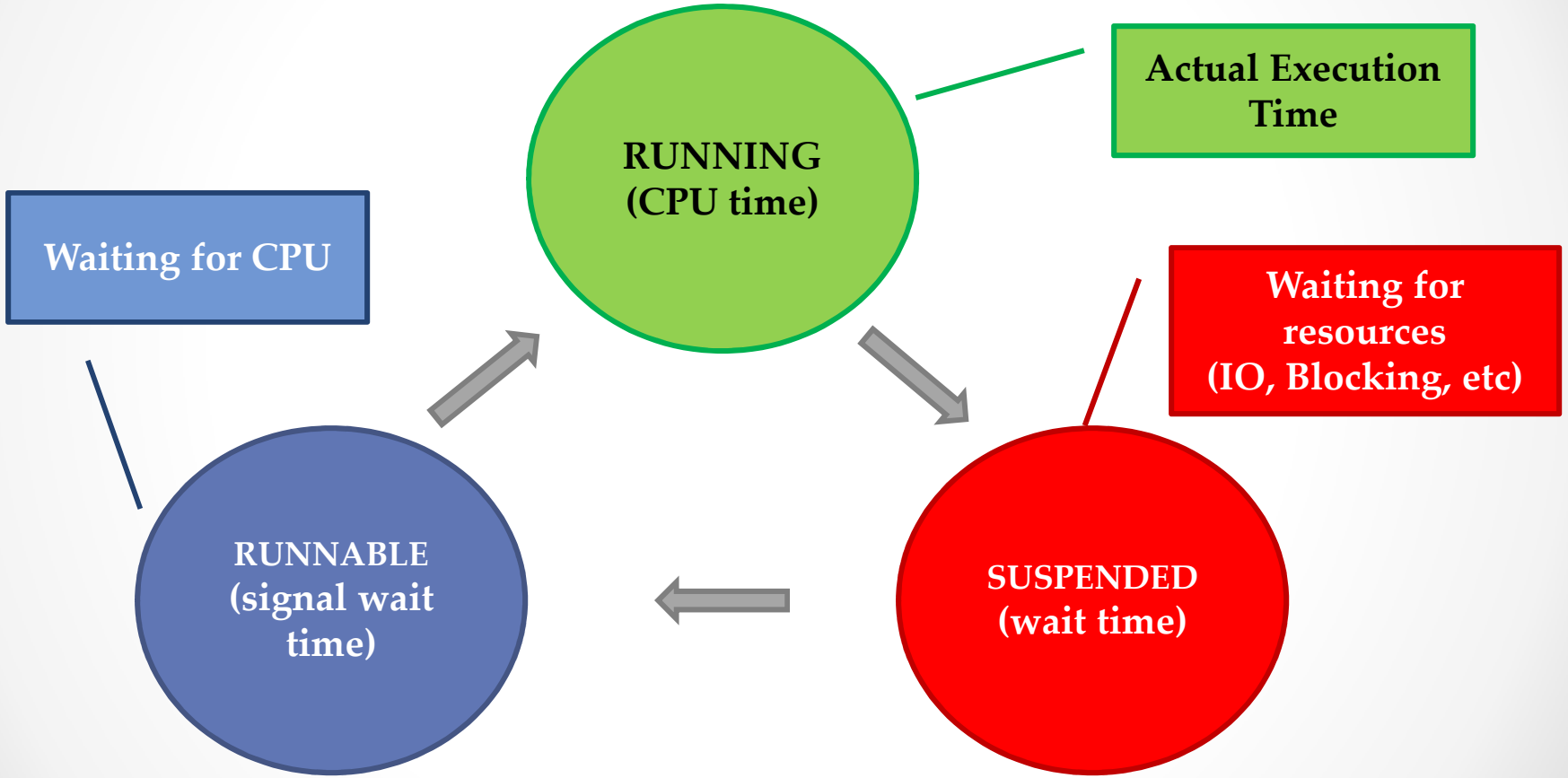
System is working slow. Time to upgrade?

- Hardware is overrated
  - *Up to degree*
- Virtualization and Clouds
  - Do you trust SAN administrator in your company?
  - What about developer who wrote SAN administration service for your cloud hosting?
  - Research and know the limitations!

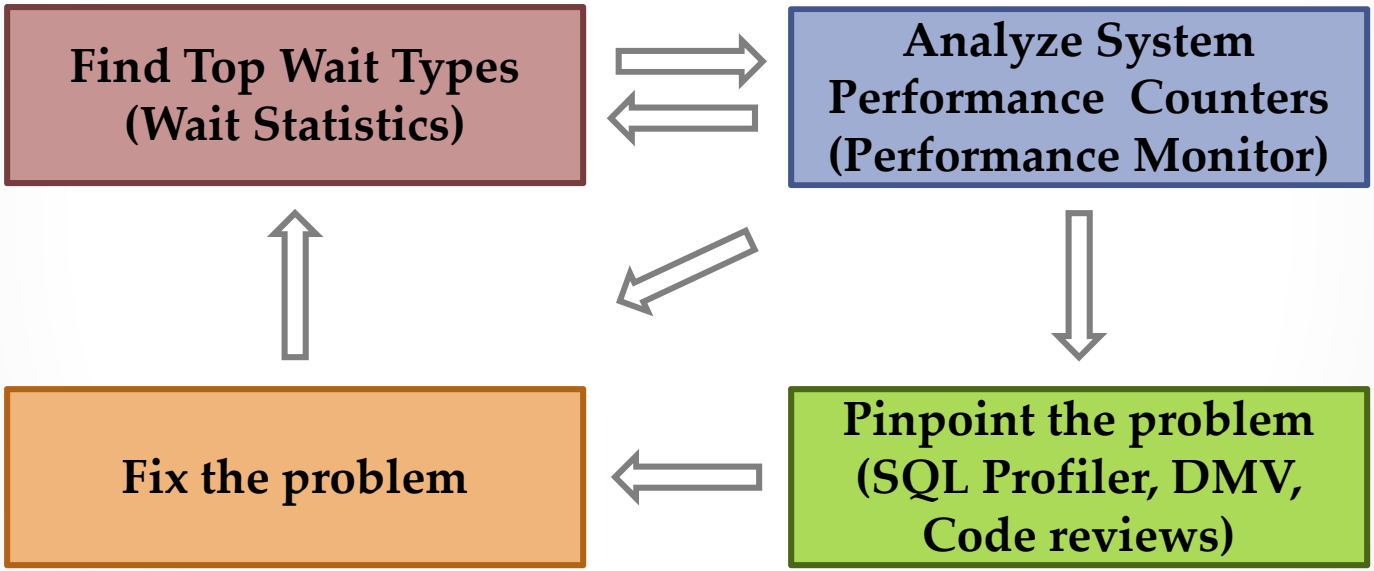
Create Baseline

- Operational standpoint
  - Most part of performance metrics are meaningless by themselves
    - *"I have 35 full scans per second. Is everything OK with my system"?*
    - *"My disk latency is 22 ms. Should I be worried?"*
  - Baseline helps to be proactive
- Helps to demonstrate achievements to the management and/or customer ☺
  - *"We have less blocking"* vs. *"% of locking waits decreased from 40% to 15%".*

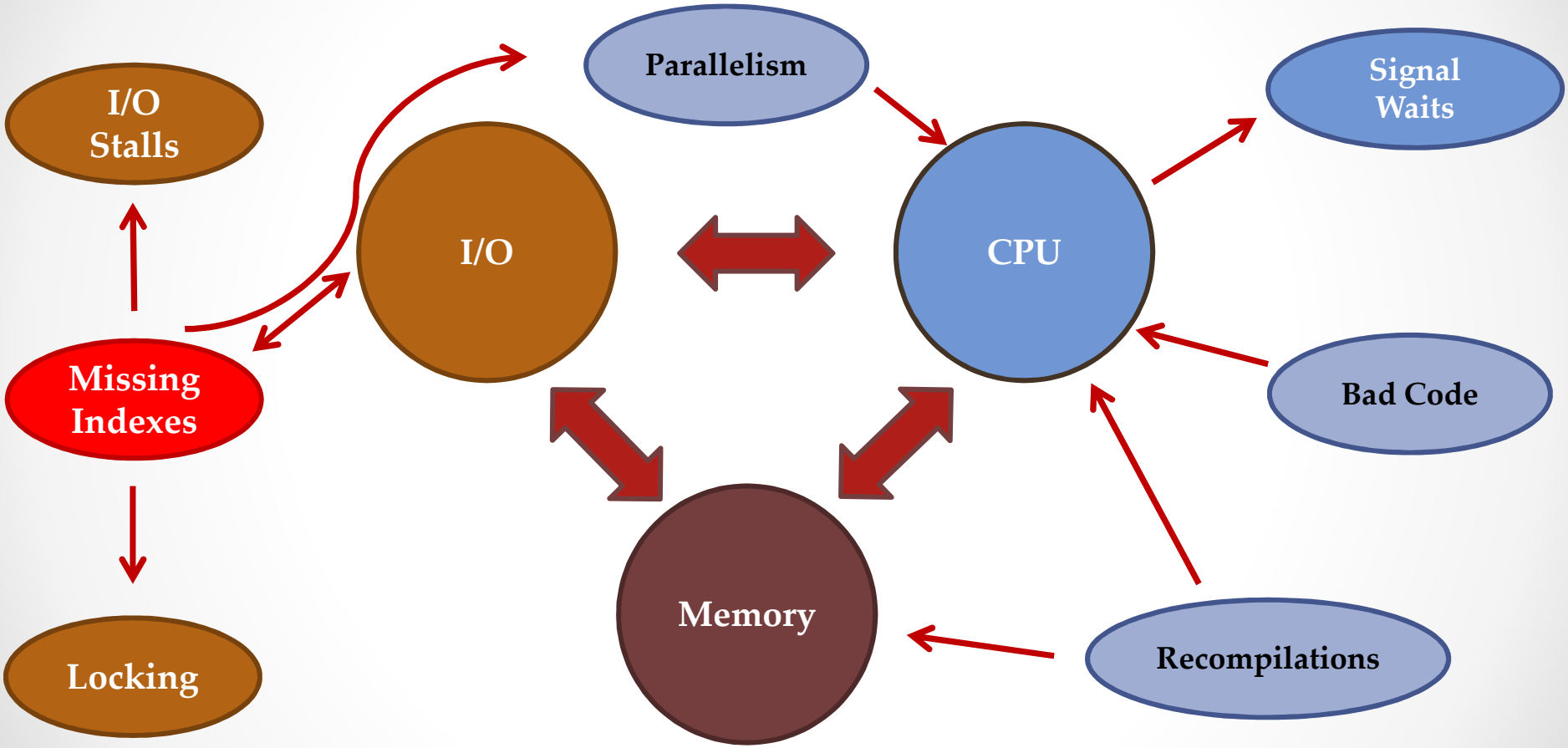
Simplified Query Life Cycle



Never-ending troubleshooting

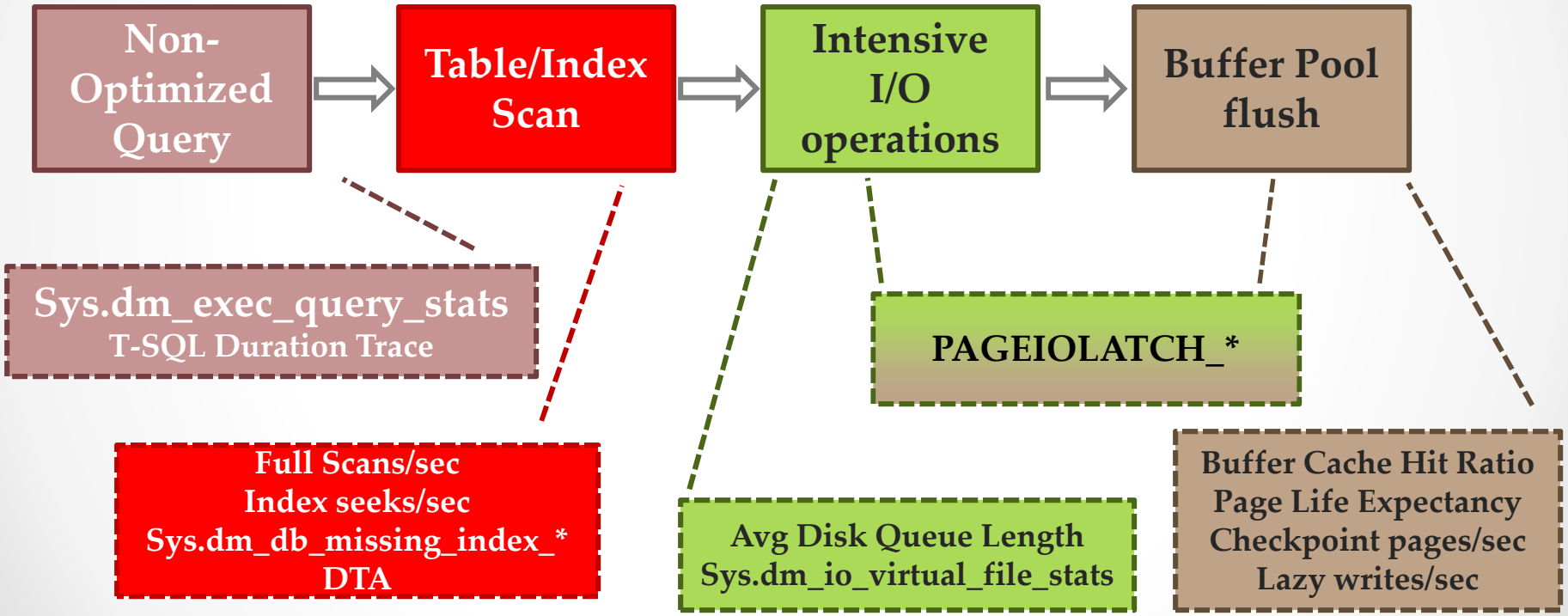


Everything is related



Memory and I/O bottlenecks

- In 95% of the cases caused by non-optimized queries



I/O and Memory issues troubleshooting

Type	Name	Description
Wait Types:	**PAGEIOLATCH_***	Disk to memory transfer
	IO_COMPLETION	I/O operations. Usually non data pages
	ASYNC_IO_COMPLETION	Asynchronous I/O
	WRITELOG, LOGMRG	Log I/O operations
Performance Objects:	**Buffer cache hit ratio**	How often page found in the cache. Should be > 95-97%
	Page life expectancy	How long page stays in the cache. Should be > 300 sec.
	Checkpoint pages/sec Lazy writers/sec	How often pages saved to disk Memory pressure: High values + low page life expectancy
	Full scans/sec	Number of full unrestricted scans per second
	Index searches/sec	Number of index searches
	(Avg) Disk Queue Length	The length of the disk queue

I/O and Memory issues troubleshooting

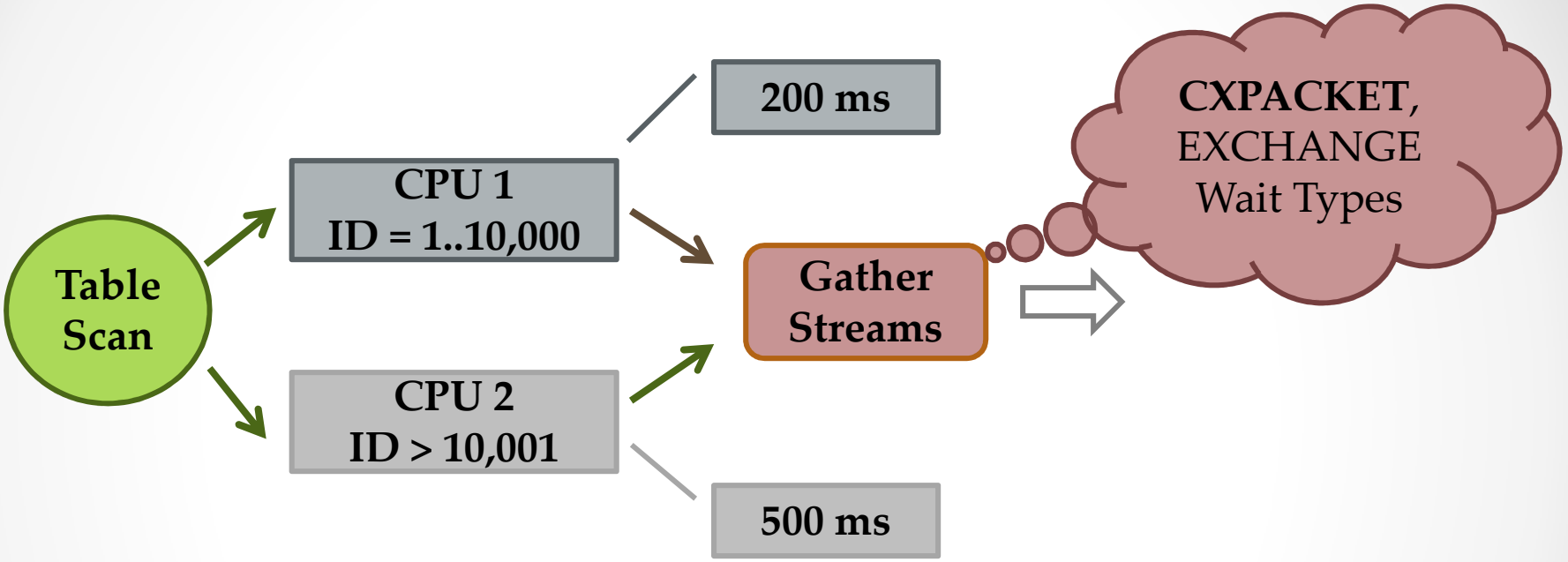
Type	Name	Description
Wait Types:	RESOURCE_SEMAPHORE	Memory grants wait and statistics Waits should be minimal for OLTP Expected for Data Warehouse type systems
Performance Objects:	Memory Grant Pending	
	Memory Grant Outstanding	
DMV:	Sys.dm_db_missing_index_*	Information about missing indexes
	Sys.dm_io_virtual_file_stats	I/O statistics for database files. Io_stall – total time that users waited for I/O

CPU Bottleneck

Type	Name	Description
Wait Types:	**SOS_SCHEDULER_YIELD**	Task is waiting for its quantum to be renewed
	CMEMTHREAD	Memory allocation from the same object. Possibly Ad-hoc sql
DMV:	**Sys.dm_os_wait_stats**	Signal_wait_time_ms > 25% of total waits
Performance Objects:	Batch Requests/sec	Total Batch Requests per second
	SQL Compilations/sec	Initial compilations + recompilations
	SQL Re-Compilations/sec	Recompilations

- Could mask:
  - Excessive Ad-Hoc SQL / Dynamic SQL / recompilations
  - Bad SQL Code
  - Non-optimized queries
- OLTP Systems:
  - Initial Compilations = Sql Compilations/sec – SQL Re-Compilations/sec
  - Plan Reuse = (Batch requests/sec – Initial Compilations) / Batch request/secs > 90%

Parallelism issues



- Parallelism is not required for OLTP Systems
- Parallelism always exists in Data Warehouse Systems
- MaxDOP must be <= # of CPUs per hardware NUMA node

Locking, Blocking and Deadlocks

Type	Name	Description
Wait Types:	**LCK_M_***	Waiting for lock to be obtained
DMV:	**Sys.dm_tran_locks**	Currently active locks
Traces	Blocked Process Report	Tasks have been blocked for more than specified amount of time
	Deadlock graph	Deadlocks
Performance Objects:	Counters from <Instance>\Locks	Locks/Timeouts/Deadlocks statistics

Why Locking?

- Major Lock Types:
  - Shared (S) – acquired by readers
  - Exclusive (X) – acquired by writers
  - Update (U) – acquired by writers while locating rows for update
- Lock Compatibility Matrix:

	S	U	X
S			☹
U		☹	☹
X	☹	☹	☹

- SQL Server always obtains U/X locks regardless of isolation level (even read uncommitted)
- (X) Locks held till end of transactions
- **Beware of non-optimized queries**

Lock Escalation

- SQL Server tries to escalate locks to the table/partitions level
  - Initial Threshold: ~5,000 locks on the object
  - If it fails, it tries again every ~1,250 locks
- Pattern: batch operation triggers lock escalation. All other sessions accessing the object are blocked
- Troubleshooting
  - High wait % of intent locks (LCK_M_I*)
  - SQL Profiler Locks: Escalation event
- Solution
  - Trace flag 1211 (instance level) – not recommended but sometimes required
  - SQL Server 2008: *alter table .. set lock_escalation*
  - Optimistic transaction isolation levels
    - Row version model – writers don't block readers

Optimistic Isolation Levels

- TANSTAAFL! (There Ain't No Such Thing As A Free Lunch!)
  - Bigger row size (14 bytes pointer)
  - Tempdb load
  - Development chalenges
- Read committed snapshot
  - When row is modified, old version stores in TempDb version store
  - Only 1 "old" version stored
  - Can be switched as the database option without the code changes
- Snapshot
  - Data "freezes" on the moment of transaction
  - Multiple "old" versions stored in the version store
  - Different behavior

```
set transaction isolation level snapshot
begin tran
    update dbo.LargeRow
    set CharField = 'Black'
    where CharField = 'White'
commit
```

```
set transaction isolation level snapshot
begin tran
    update dbo.LargeRow
    set CharField = 'White'
    where CharField = 'Black'
commit
```

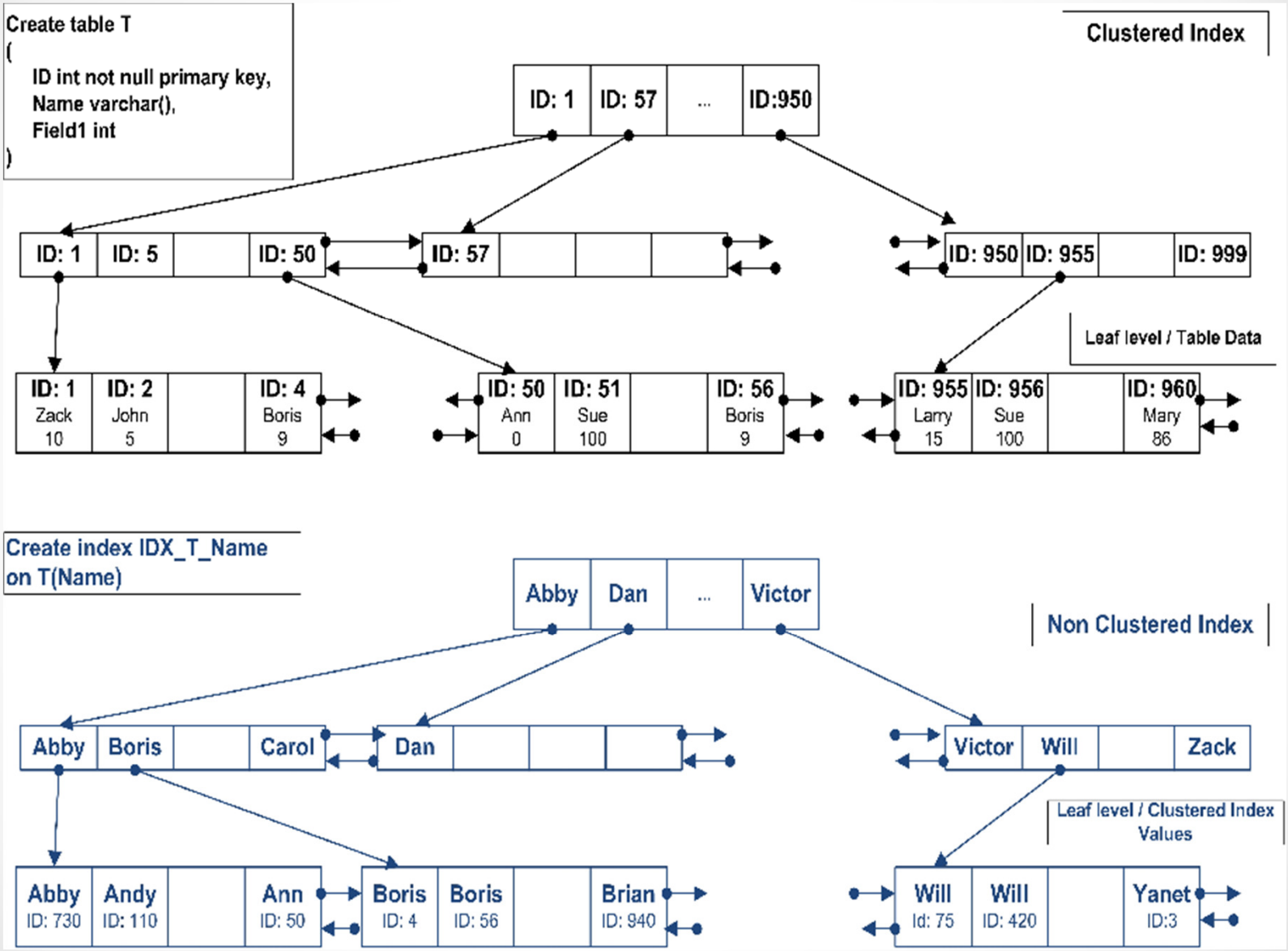
	ID	IntField	CharField
1	1	1	Black
2	2	2	White

	ID	IntField	CharField
1	1	1	White
2	2	2	Black

So.. If main bottleneck is

- I/O
  - Focus on I/O
- I/O and Memory
  - Focus on I/O
- Memory without I/O
  - Google it!
- Parallelism in OLTP system
  - Reduce MaxDOP AND TEST!
- Locking and blocking
  - Detect problematic queries (if needed)
  - Beware of Lock Escalation
  - As the temporary solution – switch to optimistic isolation level (beware of consequences!)
  - Focus on I/O. If I/O *looks* OK – check client code.
- So bottom line – detect and fix I/O issues.

Index structure



Selectivity

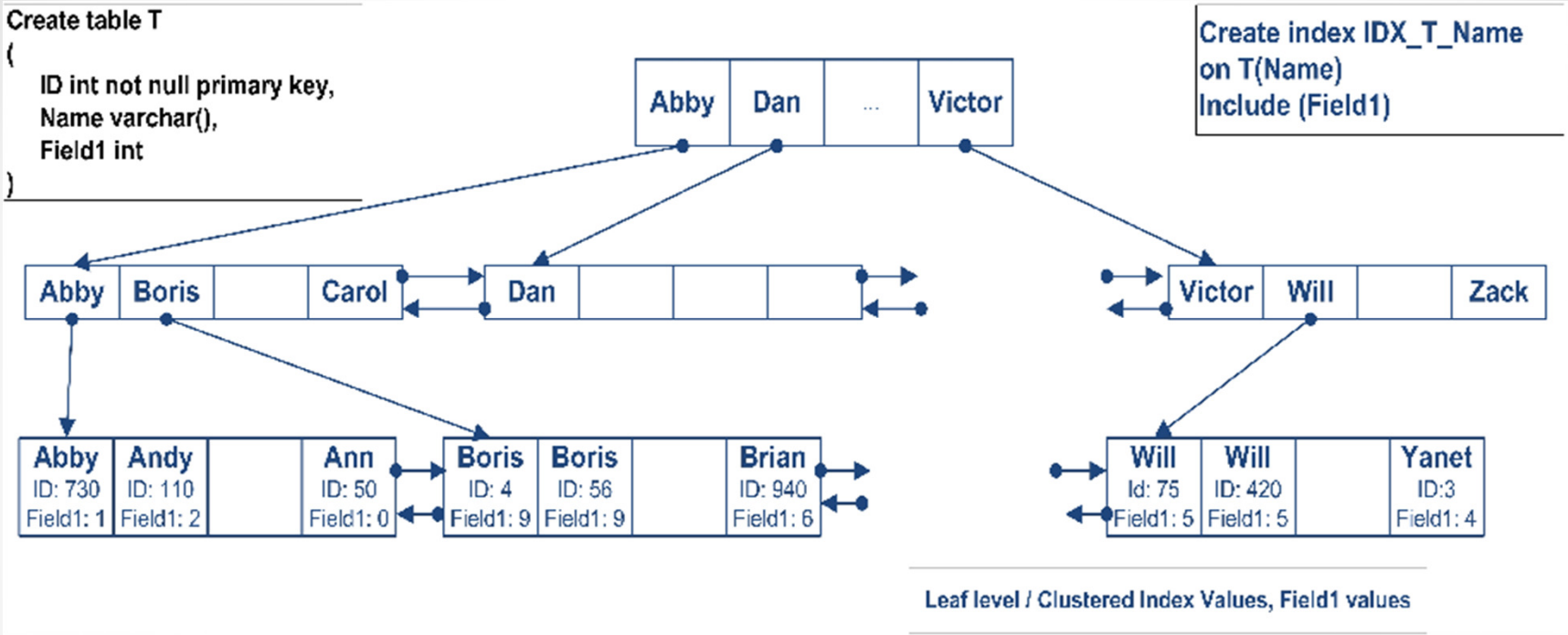
- Very simple example:
- Table has 80,000 rows, 400 bytes each, IN
- CI = 20 rows per leaf page = 4,000 pages
- Clustered index is 3 levels deep
- NCI - Name – 20 bytes = ~320 rows per page = 250 pages. Data distributed evenly
  - Select * from T where name like 'ABC%' – (1/26^3) - ~5 rows = ~20 reads
  - Select * from T where name like 'AB%' – (1/26^2) ~118 rows = ~360 reads
  - Select * from T where name like 'A%' - (1/26) ~3100 rows = ~9250 reads
  - ~1330 rows = ~1.67% = ~4000 reads

Sequential I/O With READ-AHEAD

Random I/O

Threshold varies but really low

Covered indexes



- Select Field1 where Name = 'ABC'
  - Does not require access to the CI data
- Scan always faster than CI scan (smaller row size)
- Bigger leaf row size in compare with regular indexes

Large / Very-large DB considerations

- Avoid Key Lookup when processing a lot of rows
  - Use covered indexes
    - *Range scan* on covered index could be better than *index seek + key lookup*
    - Ideally (OLTP) – very few indexes with set of included columns
      - Scans on the small rows sets
      - Avoid wide leaf rows
- Avoid non-unique CI
- Minimize Data Row size
  - Avoid fixed-width types (char/binary)
    - Use Data Compression if you cannot change DB design

Insert/Update, Random Values

```
create table dbo.Data
(
    ID int not null,
    Name nvarchar(128) not null,
    HashValue varbinary(max) null,

    constraint PK_Data
    primary key clustered(ID)
)
```

Page Splits

```
insert into dbo.Data(ID, Name)
    select top 100000 ID, Name
    from dbo.SomeStageTable

update dbo.Data
set HashValue = HashBytes('SHA1',Name)
```

```
insert into dbo.Data(ID, Name, HashValue)
    select top 100000 ID, Name, HashBytes('SHA1',Name)
    from dbo.SomeStageTable
```

```
insert into dbo.Data(ID, Name, HashValue)
    select top 100000 ID, Name, convert(varbinary(20),REPLICATE('0',20))
    from dbo.SomeStageTable

update dbo.Data
set HashValue = HashBytes('SHA1',Name)
```

Random inserts -> Page Splits, Fragmentation, etc

```
create unique nonclustered index IDX_Data_HashValue
on dbo.Data(HashValue)
```

HashValue ::= <Sequential Part> + <Random Part>

Logical data partitioning

```
create table dbo.HugeTable
(
    RecId bigint not null identity(1,1),
    AccountId int not null
        constraint FK_HugeTable_Accounts
        foreign key
        references dbo.Accounts(AccountId),
    TranDate datetime2(0) not null,

    constraint PK_HugeTable
    primary key clustered(RecId)
)
go

create nonclustered index IDX_HugeTable_TranDate
on dbo.HugeTable(TranDate)
go
```

- Localization of I/O
- Less physical I/O with read-ahead
- Less blocking

```
create table dbo.HugeTable
(
    RecId bigint not null identity(1,1),
    AccountId int not null
        constraint FK_HugeTable_Accounts
        foreign key
        references dbo.Accounts(AccountId),
    TranDate datetime2(0) not null,

    constraint PK_HugeTable
    primary key clustered(CompanyId, RecId)
)
go

create nonclustered index IDX_HugeTable_TranDate
on dbo.HugeTable(CompanyId, TranDate)
go
```

Uneven data distribution

- SQL Server uses statistics to generate "good enough" query plan
  - It keeps histogram only on the first column of composite index.
  - It expects data to be distributed evenly

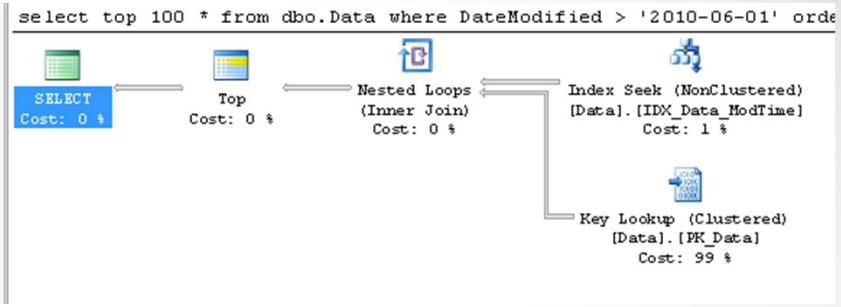
```
create table dbo.MyData
(
    AccountId int not null,
    RecId bigint not null,
    UserId int not null,
    MyDataColumns ...

    constraint PK_MyData
    primary key clustered(AccountId, RecID)
)
go

create unique index IDX_MyData_AccountId_UserId_RecId
on dbo.MyData(AccountId, UserId, RecId)
go
```

What index should it use?

```
select *
from dbo.MyData
where
    AccountId = @AccountId and
    UserId in (select UserId from #userList)
```

Table Partitioning

```
create table dbo.Data
(
    Id int not null identity(1,1),
    DateCreated datetime not null,
    DateModified datetime not null,

    constraint PK_Data
    primary key clustered(ID)
)
go

create unique index IDX_Data_ModTime
on dbo.Data(DateModified, ID)
go
```




```
select top 100 *
from dbo.Data
where DateModified > '2010-06-01'
order by DateModified, ID
go
```

```
create partition function pfData(datetime)
as range left for values
('2010-01-01', '2010-03-01', '2010-05-01',
'2010-07-01','2010-09-01','2010-11-01')
go

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

alter table dbo.Data
add constraint PK_Data
primary key clustered(ID,DateCreated)
on psData(DateCreated)
go

create unique index IDX_Data_ModTime
on dbo.Data(DateModified, ID, DateCreated)
on psData(DateCreated)
```

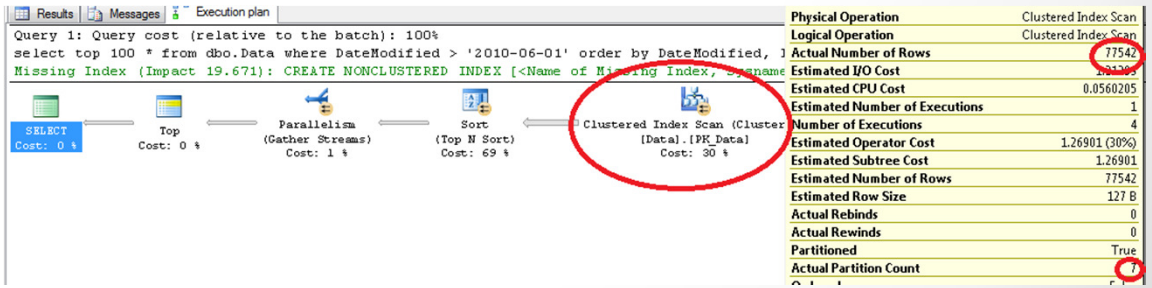


Table Partitioning

```
create table dbo.Data
(
    Id int not null identity(1,1),
    DateCreated datetime not null,
    DateModified datetime not null,

    constraint PK_Data
    primary key clustered(ID)
)
go

create unique index IDX_Data_ModTime
on dbo.Data(DateModified, ID)
go
```

DM:2009-12-01 ID: 100 DC: 2009-11-01	DM: 2009-12-02 ID: 110 DC: 2009-12-01	DM: 2010-01-02 ID: 103 DC: 2009-11-15		DM: 2010-05-04 ID: 275 DC: 2010-05-03	

```
select top 100 *
from dbo.Data
where DateModified > '2010-06-01'
order by DateModified, ID
go
```

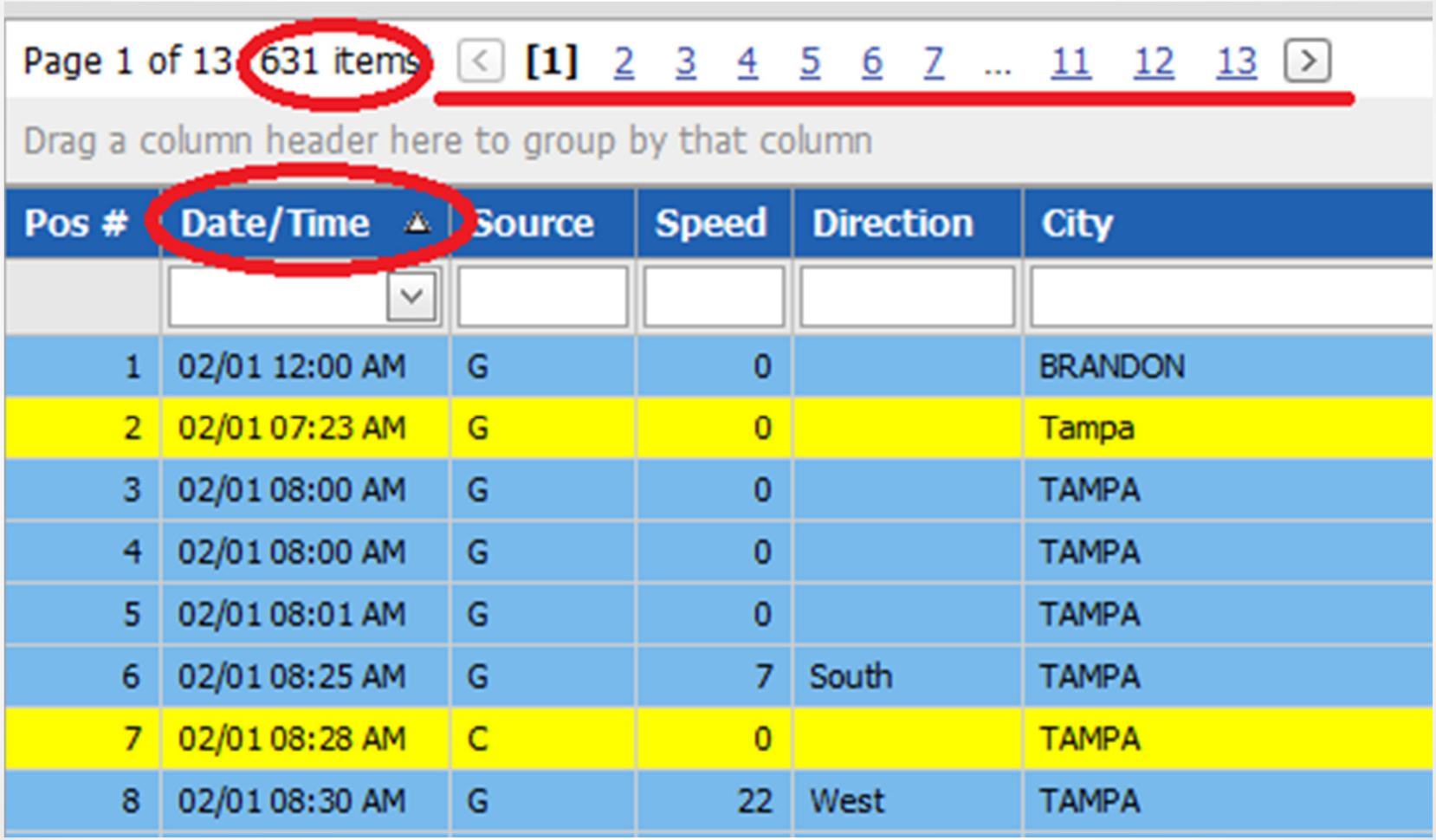
```
create partition function pfData(datetime)
as range left for values
('2010-01-01', '2010-03-01', '2010-05-01',
'2010-07-01','2010-09-01','2010-11-01')
go

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

alter table dbo.Data
add constraint PK_Data
primary key clustered(ID,DateCreated)
on psData(DateCreated)
go

create unique index IDX_Data_ModTime
on dbo.Data(DateModified, ID, DateCreated)
on psData(DateCreated)
```

DM:2009-12-01 ID: 100 DC: 2009-11-01	DM: 2009-12-02 ID: 110 DC: 2009-12-01	DM: 2010-01-02 ID: 103 DC: 2009-11-15	
DM:2010-01-01 ID: 102 DC: 2010-01-01	DM: 2010-02-02 ID: 150 DC: 2010-01-28	DM: 2010-03-15 ID: 170 DC: 2010-02-02	
DM:2010-03-02 ID: 220 DC: 2010-03-01	DM: 2010-03-02 ID: 222 DC: 2010-03-02	DM: 2010-04-05 ID: 300 DC: 2010-04-03	

Other things to check

- Heap tables (sys.indexes)
  - Don't use it unless you need staging tables
- Index usage (sys.dm_db_index_usage_stats)
- Wide clustered index (sys.indexes, sys.index_columns)
- Duplicates in leftmost columns in composite indexes
  - IDX1(A, B), IDX2(A,C) -> IDX3(A,B) INCLUDE(C) or IDX3(A,C) INCLUDE(B)

Paging, paging, paging



Pos #	Date/Time	Source	Speed	Direction	City
1	02/01 12:00 AM	G	0		BRANDON
2	02/01 07:23 AM	G	0		Tampa
3	02/01 08:00 AM	G	0		TAMPA
4	02/01 08:00 AM	G	0		TAMPA
5	02/01 08:01 AM	G	0		TAMPA
6	02/01 08:25 AM	G	7	South	TAMPA
7	02/01 08:28 AM	C	0		TAMPA
8	02/01 08:30 AM	G	22	West	TAMPA

Q & A

- Thank you for attending!
- Session will be available for download
  - http://aboutsqlserver.com/presentations
  - http://sqlsaturday.com

- Email: dmitri@aboutsqlserver.com