

SQL Server 2005 – Table Partitioning

Chad Boyd

chad.boyd@gmail.com

<http://blogs.msdn.com/chadboyd>

<http://www.mssqltips.com>

Types of Partitioning in Database Designs

- **Application-managed partitioning**
 - Data is divided among multiple tables or across servers
 - Application decides where to direct specific queries at execution time
- **Partition Views**
 - 'UNION' Views link tables within or across databases and servers (DPVs)
- **Partitioned Tables and Indexes**
 - New in SQL Server 2005
 - Applies to objects in a *single* database and instance
 - Focus of this talk!

Partitioned Tables and Indexes

- Split tables and indexes into multiple storage objects based on the value of a data column
 - Based on range of a single column's value
- Still treated as single object by the relational engine
- Handled as multiple objects by the storage engine
 - Up to 1000 partitions per object supported

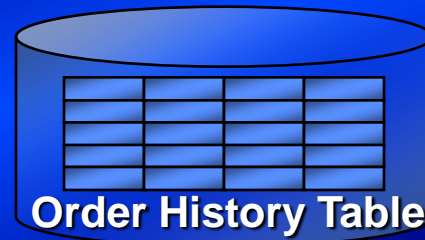
Partitioning and Storage

<i>Order History</i>
Order ID
Customer ID
Order Date
Amount
...

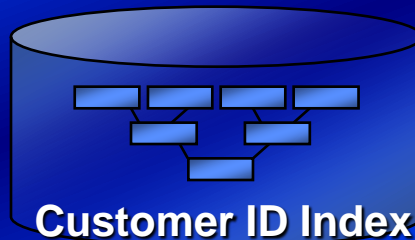
Example:
Table ORDER HISTORY with a
Nonclustered Index on CUSTOMER ID

Nonpartitioned:

Filegroup DATA



Filegroup IDX

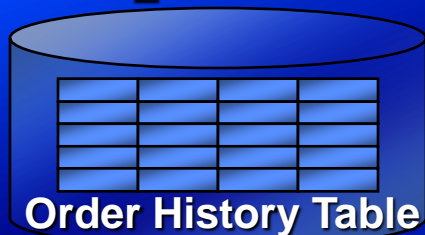


Partitioning and Storage

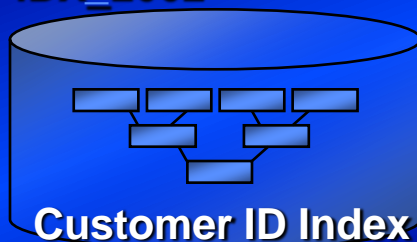
Partitioned by ORDER DATE:

<i>Order History</i>
Order ID
Customer ID
Order Date
Amount
...

Filegroup
DATA_2002

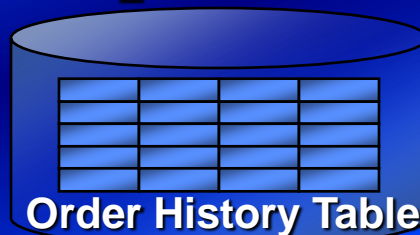


Filegroup
IDX_2002



Order Date <
'2003-01-01'

Filegroup
DATA_2003

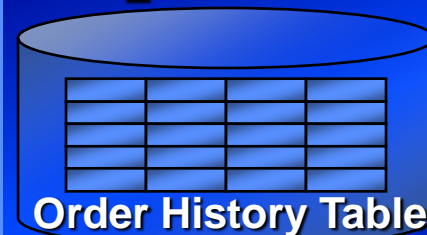


Filegroup
IDX_2003

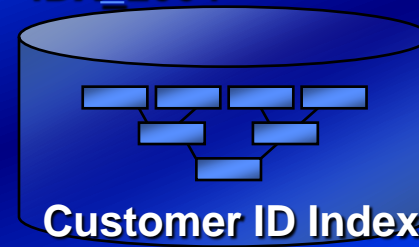


Order Date >=
'2003-01-01' and
Order Date <
'2004-01-01'

Filegroup
DATA_2004



Filegroup
IDX_2004



Order Date >=
'2004-01-01'

Benefits of Partitioned Tables

■ Manageability

- *Fast Data Deletion and Data Load*
- *Piecemeal backup / restore of historical data*
- *Partition-wise index management*
- *Minimize index fragmentation for historically-partitioned tables*
- *Support alternative storage for historical data*

■ Performance querying Large Tables

- *Join efficiency*
- *Smaller index tree or table scan when querying a single partition*
- *Simpler query plans compared to Partition Views*

Improvement Over Partition Views

- Partitioned Table: a *single object* in query plans
 - Single set of statistics
 - Smaller plans, faster compilation than Partition Views
- Auto-parameterization supported
- Insert / Bulk Insert / BCP fully supported
- Numerous fine-grained partitions work well
- Queries may access partitions in parallel
 - Partition is the *unit* of parallelism

But...

- Cannot span multiple DBs or instances
 - Potentially use PV or DPVs atop Partitioned Tables

Partition Building Blocks

Objects:

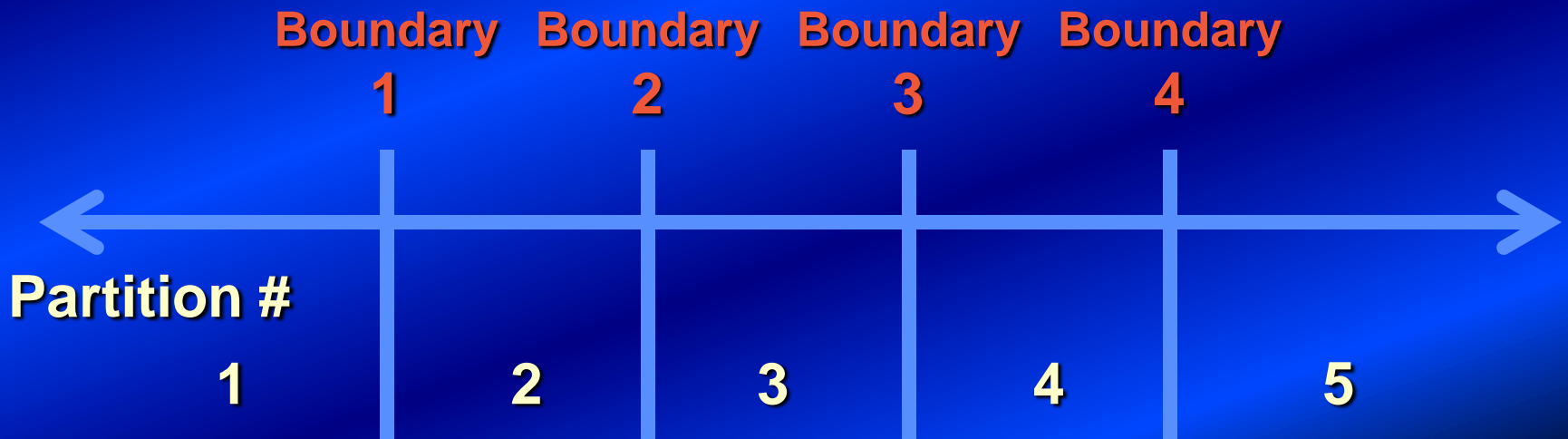
- Partition Function
- Partition Scheme

Operations:

- Split Partition
- Merge Partition
- Switch Partition

Partition Function

- Maps ranges of a data type to integer values
- Defined by specifying boundary points
- N boundary points define $N+1$ partitions



Partition Function DDL

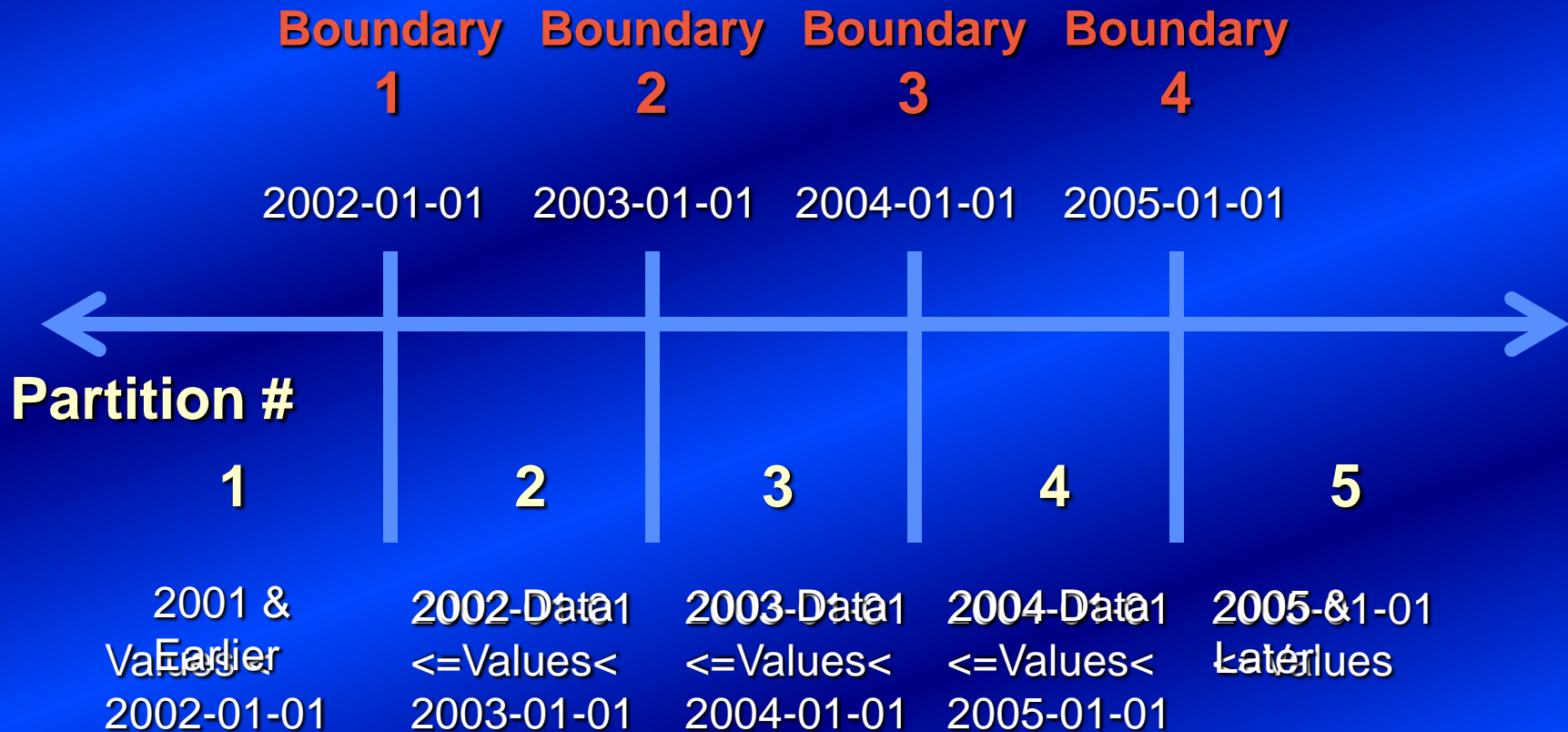
```
CREATE PARTITION FUNCTION annual_range  
    (DATETIME)
```

```
as RANGE RIGHT
```

```
for values
```

```
(          -- Partition 1 -- 2001 and earlier  
'2002-01-01', -- Partition 2 -- 2002  
'2003-01-01', -- Partition 3 -- 2003  
'2004-01-01', -- Partition 4 -- 2004  
'2005-01-01'   -- Partition 5 -- 2005 and later  
)
```


Range Right Datetime Partition Example



Partition Function Notes

- Understand the difference between LEFT and RIGHT 😊
- Use `$partition.<function_name>` to query partition function values
- Best practice:
 - Use RANGE RIGHT for 'continuous' data values – more intuitive
 - Boundary becomes starting point for each range

Partition Scheme

- Associates a storage location (Filegroup) with each partition defined by a partition function
- No *requirement* to use different filegroups for different partitions
 - Useful for Manageability
 - Filegroup-based backup or storage location
- Best Practice: Spread *all* of your Filegroups in a Partition Scheme across as many disk spindles as possible.
 - Rarely want to dedicate separate drives to separate partitions

Partition Scheme DDL

```
CREATE PARTITION SCHEME annual_scheme_1
as PARTITION annual_range to
(annual_min,                                -- filegroup for pre-2002
annual_2002,                                -- filegroup for 2002
annual_2003,                                -- filegroup for 2003
annual_2004,                                -- filegroup for 2004
annual_2005)                                -- filegroup for 2005 and later
```

```
Create PARTITION SCHEME annual_scheme_2
as PARTITION annual_range
ALL to ([PRIMARY])
```

Range Right Datetime

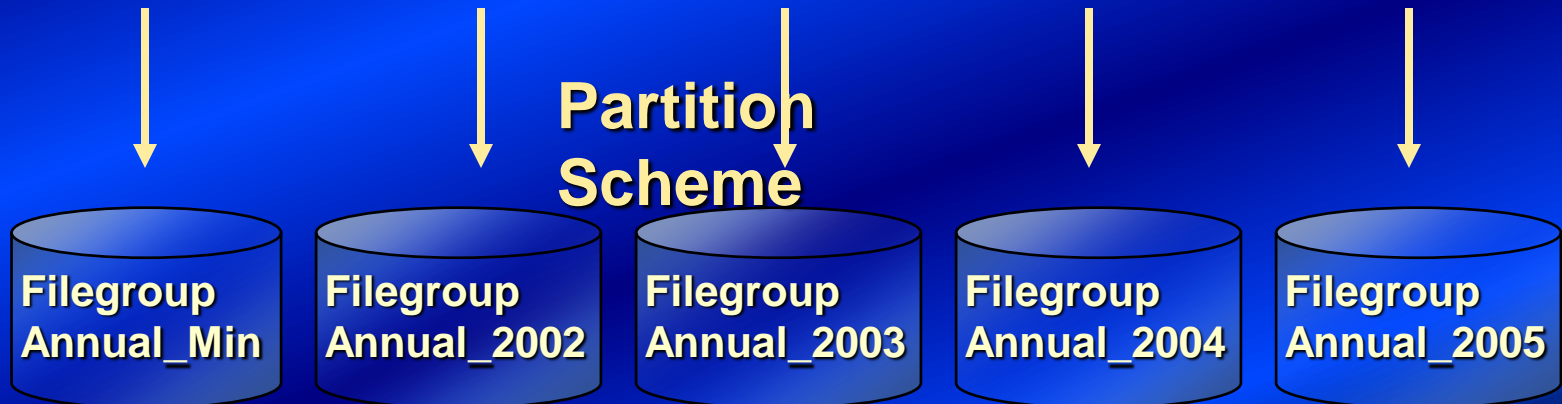
Partition Function

Boundary 1 **Boundary 2** **Boundary 3** **Boundary 4**

2002-01-01 2003-01-01 2004-01-01 2005-01-01



Partition Scheme



Partitioned Tables & Indexes

- A single column must be selected as the Partitioning Key
- Partitioned Tables and Indexes are created on *Partition Schemes* instead of *Filegroups*
- All query operations on tables or indexes are transparent to partitioning
- Different tables and indexes *may* share common partition functions and schemes



Table and Index Creation

```
CREATE TABLE Order_History (  
    Order_ID    bigint,  
    Order_Date  datetime,  
    Customer_ID bigint  
    ...  
) ON Annual_Scheme_1(Order_Date)
```

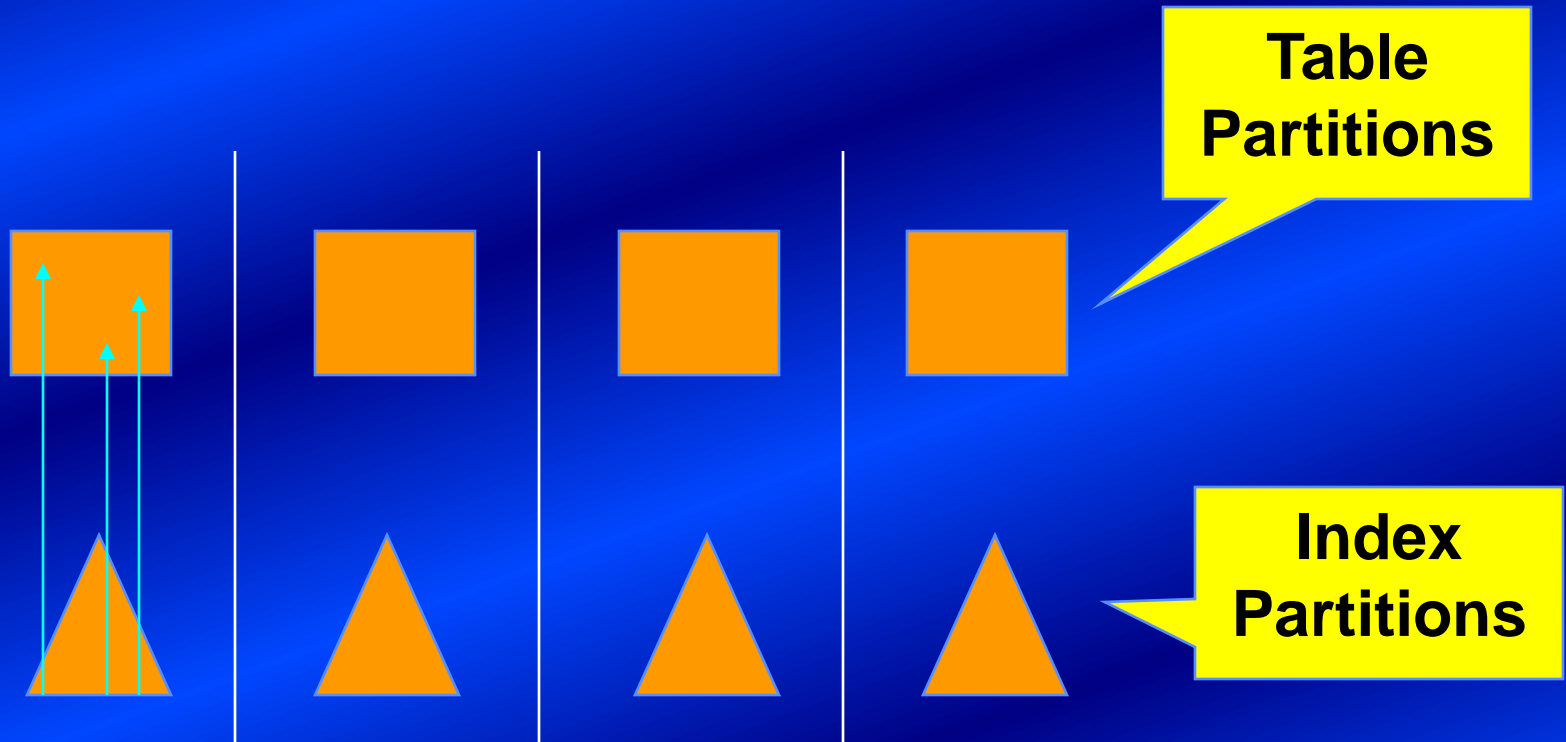
```
CREATE INDEX Order_Cust_Idx  
    ON Order_History(Order_ID)  
ON Annual_Scheme_1(Order_Date)
```

Index Partitioning

- Partitioning Key of an Index *need not be part of the Index Key*
 - SQL 2005 indexes can include columns outside of the btree, at the leaf level only
 - Essential for partitioning, and also great for covering index scenarios
- If an index uses a similar partition function and same partitioning key as the base table, then the index is “aligned”
 - One-to-one correspondence between data in table and index partition
 - All index entries in one partition map to data in a single partition of the base table

Aligned Index

One-to-one partition correspondence



Having all aligned indexes is a best practice

Index Alignment

■ The Good

- Aligned Indexes are the default
- Partitioning key is *automatically* added to the index
- Defaults to same partition scheme as the base table

■ The Bad

- A Non-Aligned Index reduces the manageability benefits of partitioning a large table
- Prevent *fast Switching* of data into / out of table

■ The Ugly

- For a UNIQUE or PK Index to be partitioned, the Partitioning Key *must* be part of the Unique Key
- Otherwise the index will be non-partitioned – and therefore Non-Aligned

Key vs. Alignment Tradeoff

- Example: A large table of Orders
 - Natural Partitioning Key = Order_Date
 - Unique Business Key = Order_ID
- Design Alternatives:
 1. Partition table on Order_Date, *Non-Unique*, aligned index on Order_ID
 2. Partition table on Order_Date
a Unique, *non-aligned* index on Order_ID
 3. Design the Order ID to always increase in time,
Partition on Order_ID, align table and indexes
- Best Solution depends on your workload!

Querying -- Best Practices

- Restrict the Partitioning Key in the WHERE clause to reduce # of partitions touched
 - Critical for performance of both large queries and high-volume small queries

Select * from OrderHistory o where ...
o.date_key between '2002-01-01' and '2002-06-30'
- Beware: Restrictions applied via a *join* will *not* eliminate partitions
 - Following query will touch *all* Region partitions:
Select * from Sales s INNER JOIN Region d
~~on d.region_id = s.region_id where ...~~
~~*d.Name = 'Asia'*~~ *s.region_id = 7*
 - Resolve by placing restriction on region_id instead

Other Query Enhancements

- Many other operations can be performed “per-partition”
 - Grouping, filtering, projection
 - Inserts, updates, deletes
 - Create index, bulk insert
- SQL Server tries to find large groups of operations that can be performed per-partition to improve query performance

DW Partitioning Best Practices

- Typically partition Fact Tables by Period
 - Easily unload history and load new periods
 - Make typical queries more efficient
- Partition on a meaningful datetime or integer column that is usually *directly restricted* in queries, e.g.
 - WHERE Date_key between '2004-01-01' and '2004-01-31'
 - WHERE Period_key = 200402 (month = Feb 2004)
 - WHERE Year_key = 2004
- Don't Partition based on a *surrogate* period key
 - ...unless the query application can explicitly restrict the key
 - Users typically don't know those keys and can't constrain them directly in ad-hoc queries

OLTP Partitioning Best Practices

- When partitioning large transaction tables for manageability
 - Partition directly on a Primary Key that increases monotonically over time
 - Allows fast lookup on the key, simple maintenance
 - Avoids non-aligned indexes
- Some applications benefit from partitioning on Organization or Geography
 - Especially if these are naturally part of a Primary Key
 - Potential query performance benefits

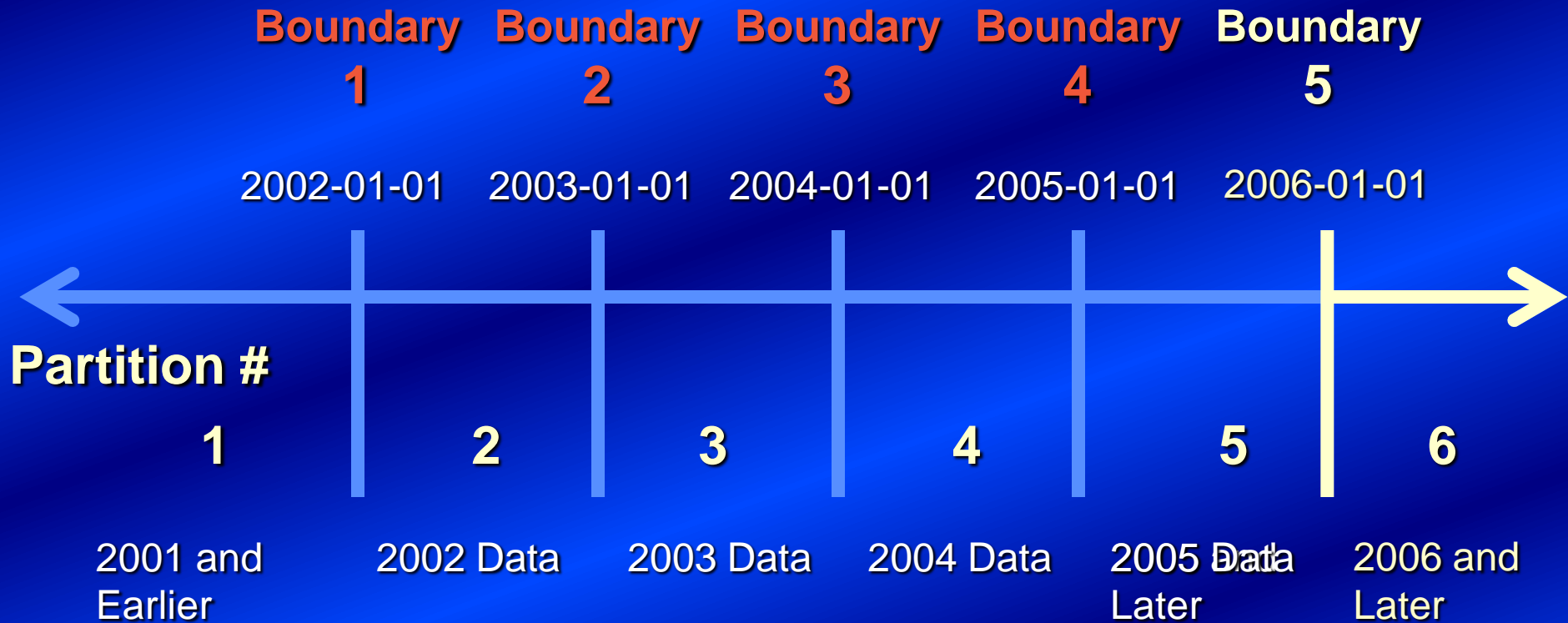
Add & Drop Partitions?

- A typical requirement is to insert or remove entire partitions of data in bulk
- Achieve this with a sequence of basic operations on partitions:
 - Split
 - Merge
 - Switch

Split

- **ALTER PARTITION FUNCTION ... SPLIT RANGE ...**
- Adds a boundary point to a Partition Function
- Affects all objects using that Partition Function
- One partition is split into two
 - The *new* partition is the one containing the *new* boundary point:
 - To the right of boundary if RANGE RIGHT
 - To the left of boundary if RANGE LEFT
- Data that falls into the *new* range is *moved* from the split partition

Split Example: Range Right



```
ALTER PARTITION FUNCTION annual_range()  
SPLIT RANGE ('2006-01-01')
```

2006 and later
data moved

Split -- Considerations

- Instantaneous if partition is *empty*
- This is *IO-intensive* if partition is populated
 - Any data on *new* side of the boundary is physically deleted from old partition and inserted into the *new* partition
 - Fully logged operation
 - Exclusive table lock held for duration of Split
- Schema change, so plans are invalidated
- Always assign 'Next Used' filegroup in Partition Scheme before using Split
 - **ALTER PARTITION SCHEME NEXT USED ...**
 - Specifies which filegroup holds new partition's data

Merge

- **ALTER PARTITION FUNCTION ... MERGE RANGE**
- Removes a boundary point from a Partition Function
- Affects all objects using that Partition Function
- The partitions on each side of the boundary are merged into one
 - The partition that held the boundary value is *removed*
 - To the right of boundary if RANGE RIGHT
 - To the left of boundary if RANGE LEFT
- Data that was in the removed partition is moved to the remaining partition

Merge Example: Range Right

Boundary 1 **Boundary 2** **Boundary 3** **Boundary 4** **Boundary 5**

2002-01-01 2003-01-01 2004-01-01 2005-01-01 2006-01-01

1

2

2

3

4

5

2001 and
Earlier

2002 Data
Earlier

2003 Data

2004 Data

2005 Data

2006 and
Later

2002 Data
Moved

```
ALTER PARTITION FUNCTION annual_range()  
MERGE RANGE ('2002-01-01')
```

Merge -- Considerations

- Instantaneous if merged range is *empty*
- *IO intensive* if the partition is already populated
 - Any rows in the removed partition are physically moved into the remaining partition.
 - Fully logged operation
 - Exclusive table lock held for duration of Split
- Schema change, so plans are invalidated

Switch

- Instantly swaps the content of one partition or table (source) with another table's empty partition or an empty table (target)
- Metadata-only operation, no data moves
- Restrictions:
 - Target table or partition must be *empty*
 - Source and target must be in *same filegroup*
 - Source must have all indexes required by the target, aligned and in matching filegroups
 - If Target is a partition, Source must have check constraints (if table) or a partition range that fits within the target range
- All associated indexes are automatically Switched along with the table / partition

Switch – Table to Partition

Table A:

2002-01-01 2003-01-01 2004-01-01 2005-01-01 2006-01-01

[EMPTY]



Partition 1
2001 & Earlier

2
2002 Data

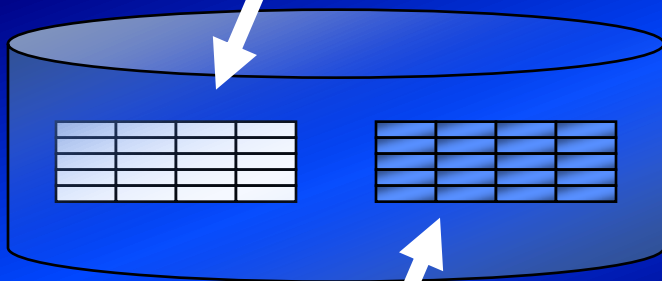
3
2003 Data

4
2004 Data

5
2005 Data

6
2006 & Later

Filegroup
DATA_2002



**Alter Table B
SWITCH TO A
PARTITION 2**

Table B:



[EMPTY]

CHECK CONSTRAINT:

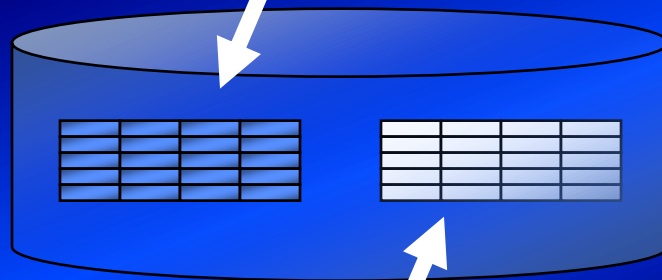
B.Date_Key >= '2002-01-01' and
B.Date_Key < '2003-01-01'

Switch – Partition to Table

Table A:

2002-01-01 2003-01-01 2004-01-01 2005-01-01 2006-01-01

[EMPTY]



**Alter Table A
SWITCH
PARTITION 2
to B**

Table B:



[EMPTY]

Switch – Partition to Partition

Table A:

2002-01-01 2003-01-01 2004-01-01 2005-01-01 2006-01-01

[EMPTY]

Partition 1
#

2001 &
Earlier

2

2002 Data

3

2003 Data

4

2004 Data

5

2005 Data

6

2006 &
Later

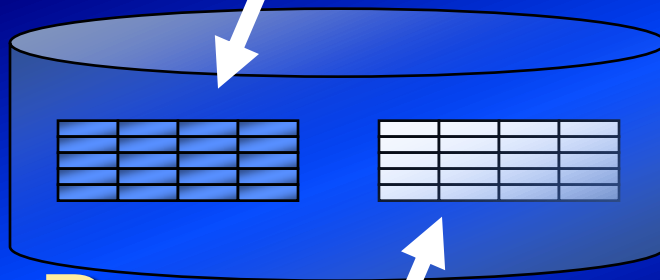


Table B:

1

Partition #
2001 &
Earlier

2

2002 Data
[EMPTY]

3

2003 &
Later

**Alter Table A
SWITCH
PARTITION 2
to B
PARTITION 2**

Switch Tips

- If Switching between a nonpartitioned table and a partition, be sure that the nonpartitioned table's indexes *include the partitioning key*
 - May need to explicitly specify the column using the INCLUDE option of CREATE INDEX (Beta 3)
- Use the \$partition function in scripts that automate the Switch command
 - Avoids absolute partition number references
 - Helpful since partition numbers may change over time

Scenarios

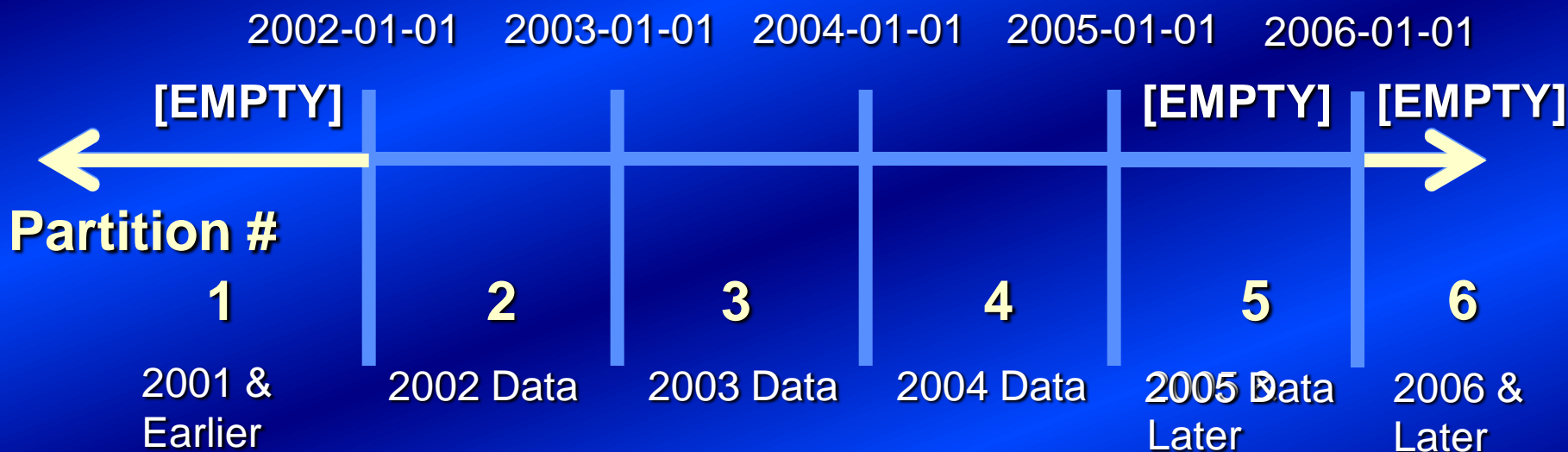
- **Sliding Window Load and Delete**
- **Efficient Backup and Restore**
- **Partition-wise Index Maintenance**
- **Snapshot (for availability)**

Sliding Window Scenario

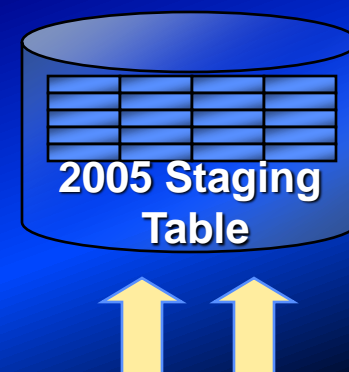
Assumptions

- Large database
- Each hour/day/week/month...add new partition and remove the oldest one.
- New partition –
 - May need to batch load, scrub, and transform before incorporating into the whole table
 - *Or* start partition as empty and populate gradually using transactions.
- Old partition – may need backup, archive, restore.

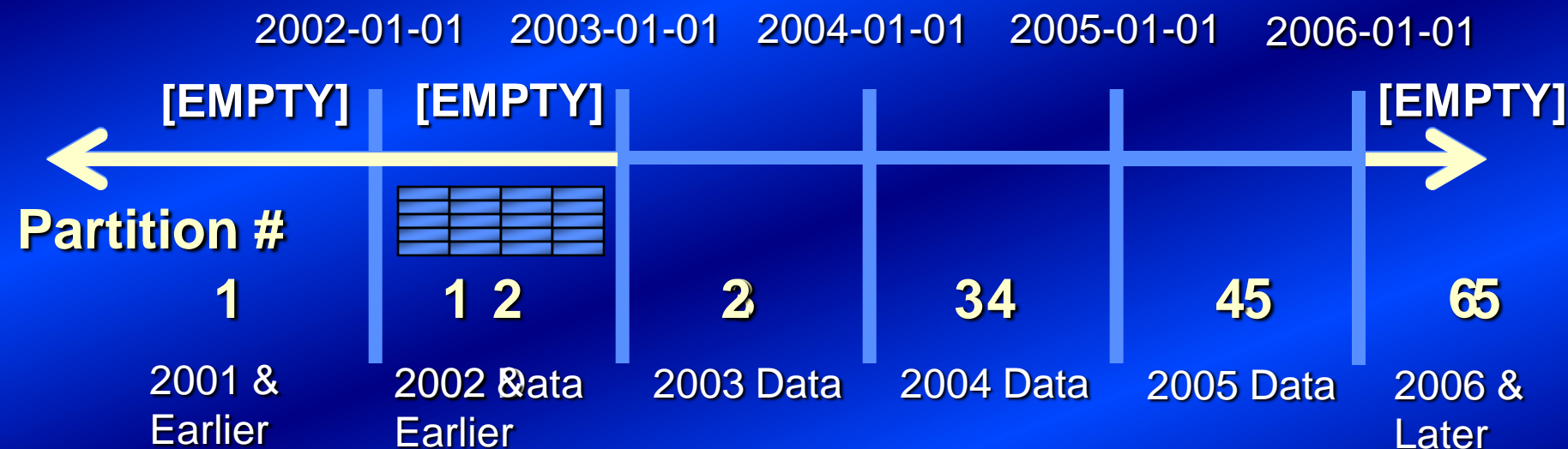
Loading Most Recent Data



- Create staging table in same filegroup as target partition (2005)
- Split most recent partition, adding boundary point for following period
- Bulk load and index staging table
- Switch data into next-to-last partition



Unloading Oldest Data



- Create unload table in same filegroup as partition to remove (2002)
- Switch data out of second partition
- Merge first partition, removing the boundary point for the unload period



Sliding Window – Best Practices

- Using Range Right Partitioning:
- Always maintain *empty* partitions for the earliest and latest date ranges
 - Ensures that Split and Merge is instantaneous
- Adding New Data:
 - First Split latest date range, then Switch data into the partition to the left of the boundary
- Removing Old Data:
 - First Switch data out of the oldest populated partition (Partition #2), then Merge the first date range

Backup & Restore

- Partitioning leverage Filegroup Backup / Restore enhancements in SQL Server 2005
 - Read-Only Filegroups can now be restored *without* applying transaction logs
- Reduces data volume for regular backups when historical data is not changing

Online Restore

- **SQL Server 2000**

- Database is not available during restore

- **Yukon**

- Database remains online

- Only data being restored is offline

- Options

- File / Filegroup Restore

- Damaged Page Tracking and Page-Level Restore

Piecemeal Restore

- Online restore of filegroups by priority
- Database is online if Primary filegroup is online
- Online throughout restore

Backups

Primary



Log



Filegroup A



Filegroup B



Database

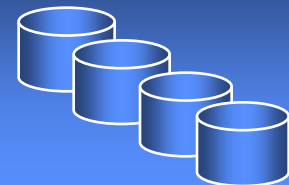
Primary
Filegroup



Filegroup A



Filegroup B



Piecemeal Backup & Restore

- Mark filegroups for unchanging, historical partitions as READ ONLY
- Perform one-time backup of these READ ONLY filegroups
- Regularly backup only the Primary filegroup and other filegroups containing active, changing data
 - Potentially a small percentage of total data
- To Restore, restore Primary and active filegroups and recover log. Then restore the read-only filegroups separately

Partitioned Index Maintenance

SQL Server 2000	SQL Server 2005
DBCC DBREINDEX	ALTER INDEX ...REBUILD
DBCC INDEXDEFRAG	ALTER INDEX ... REORGANIZE

- To reindex (REBUILD INDEX) a single partition: **ALTER INDEX ...
REBUILD PARTITION = <partition number>**
- To defragment (REORGANIZE) a single partition: **ALTER INDEX ...
REORGANIZE PARTITION = <partition
number>**

Online Index Operations

- Online Index Operations allow concurrent modification of the underlying table or index
 - Updates, Inserts, Deletes
- Online Index Maintenance
 - Create, Rebuild, Drop
 - Reorganize (including BLOBs)
 - Index-based constraints (PK, Unique)
- DDL is simple
- Online / Offline are both supported
- Updates incur some additional cost during an online index operation

Index Maintenance Notes

- Historical partitions may never need rebuilding or reorganizing indexes
- For large, partitioned transaction tables
 - Different partitions may be reindexed on different schedules (See ALTER INDEX)
 - Potential to shorten maintenance cycle
- Note: *Online* index rebuilds cannot be performed on individual partitions

Conclusion

- **Plan ahead:**
 - **Consider Partition Keys and relationship to any required unique indexes**
 - **Ensure Partition Keys are restricted in important queries' WHERE clauses**
 - **Design for bulk load and bulk delete strategy**
 - **Design for backup and restore strategy**

General Best Practices

- Use **ALIGNED** indexes
- Split and Merge only *empty* partitions for best maintenance performance
- Spread all data across as many disks as possible
- Test all of your partitioning operations & scripts on empty tables and indexes first, including
 - Piecemeal backup and restore
 - Maintenance plans
 - Split / merge / switch operations
 - New filegroups and ALTER PARTITION SCHEME