

SQLskills Immersion Event

IE1: Internals and Performance

Module 12: Partitioning

Kimberly L. Tripp
Kimberly@SQLskills.com



Overview

- Horizontal partitioning strategies
 - Partitioned views
 - Partitioned tables
- Implementing the sliding window scenario
- Partitioning design techniques combined
- The case for columnstore indexes



2

© SQLskills, All rights reserved.
<http://www.SQLskills.com>

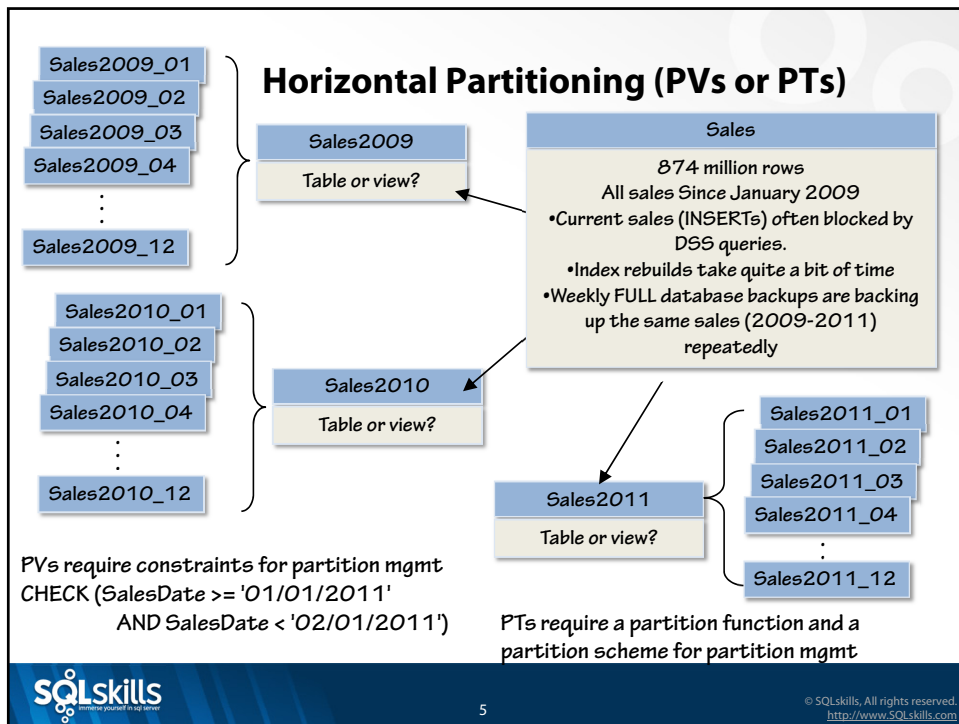


Horizontal Partitioning: Motivation?

- Resource contention limitations
- Varying access patterns
- Maintenance restrictions
- Availability requirements
- To remove resource blocking or minimize maintenance costs
- Important notes:
 - Partitioning does not automatically mean Distributed Partitioned Views (DPVs)
 - Partitioning is more of a scale-up/manageability enhancement; DPVs are a form of scale-out partitioning

Table's Shared Data/Indexes

- A single large table...
 - Presents different types of problems
 - Management
 - Index create/build or rebuilds
 - Backup/Restore and Recovery
 - Escalation
 - May have different access patterns
 - Some data new – OLTP (inserts/updates for new and current sales)
 - Some data old for historical lookups – small singleton for OLTP lookups or larger for analysis
- Does NOT have to be one table!



Functional Partitioning – Why?

- **More granular control**
 - Varying access patterns
 - Varying index defrag patterns
 - Very fast sliding window control
- **More backup/restore choices**
 - File/filegroup backups
 - File/filegroup restores from anything larger
 - Full backups support PAGE, FILE, FILEGROUP or FULL restores
 - Filegroup backups support PAGE, FILE or FILEGROUP restores
 - File backups support PAGE or FILE restores
 - Even support for file/filegroup in simple recovery model (backups separated between RW and RO backups and RO backups can be performed at any frequency!)
- **Significantly better availability**

SQLskills
© SQLskills. All rights reserved.
<http://www.SQLskills.com>

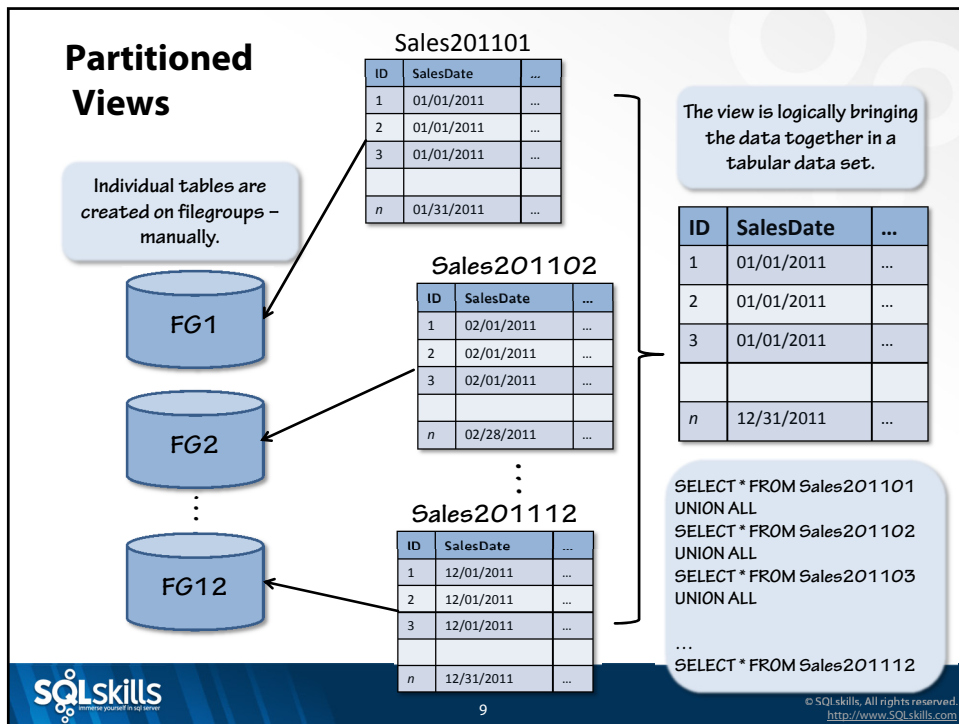
Functional Partitioning – What?

- Structures designed for sliding window scenario
- Tables created and data flows on regular/consistent basis (weekly, monthly, yearly, etc.)
- Data may be archived/removed to keep only “current” timeframe (year, two years, etc.)
- Implemented with either partitioned views or partitioned tables



Functional Partitioning – How?

- **Partitioned tables (Enterprise Edition)**
 - Can convert an existing table as an ONLINE operation IF the table doesn't have any LOB columns in 2005/2008/R2 (fixed in 2012)
 - Might run into problems around “unique” index requirements for PTs in that the partitioning column must be a member of the key – for all unique indexes
 - Cannot do fast switching in 2005 if Indexed Views
 - Cannot do fast switching if iFTS desired
- **Partitioned views (Any edition)**
 - Might be able to replace an existing table with a view (even for DML) if you meet the correct criteria
 - Can programmatically direct modifications
 - Might be able to use INSTEAD OF triggers
 - Conversion may require downtime or time where certain data is inaccessible



Partitioned Views

Still Motivation for Creating in SQL Server 2005+

- **Queryable partitioned views in SQL Server 7.0**
 - Query Optimizer removes irrelevant tables from query plan (partition elimination)
- **Updateable partitioned views in SQL Server 2000**
 - Inserts/updates/deletes may need to be directed to correct base table (if you can't meet UPV requirements that partitioning key has to be the first column of the primary key)
 - Data modification statements can be directed to appropriate base tables IF the partition key is part of the primary key and it's enforced by a CHECK constraint
- Requires separately named/created tables
- Manual placement on different filegroups
- Checked/verified constraints – and tested for logic problems (what will happen on February 29th?)
- UNION ALL views

SQLskills
© SQLskills. All rights reserved.
<http://www.SQLskills.com>

Creating Partitioned Views

- Create files/filegroups
- Create individual tables on filegroups
- Create base table indexes as per performance requirements on ALL partitions
- Create base table constraints to restrict range on ALL partitions, **MUST** be checked and verified ("trusted" constraints)
- Create views
 - Must use UNION ALL (not UNION)

Check Constraints (Filtering)

- Defines column's domain of legal values
- All data checked on INSERT/UPDATE
 - Only range constraints supported

```
CHECK (SalesDate >= '20040101'
      AND SalesDate < '20040201')
```
 - Limited subset of range operators supported for partitioning (i.e. filtering) are: BETWEEN, AND, OR, <, <=, >, >=, =
 - Others are supported for data integrity purposes only
- Added during CREATE TABLE
- Add later using ALTER TABLE

But what about the existing data?

Default is to CHECK the existing data...

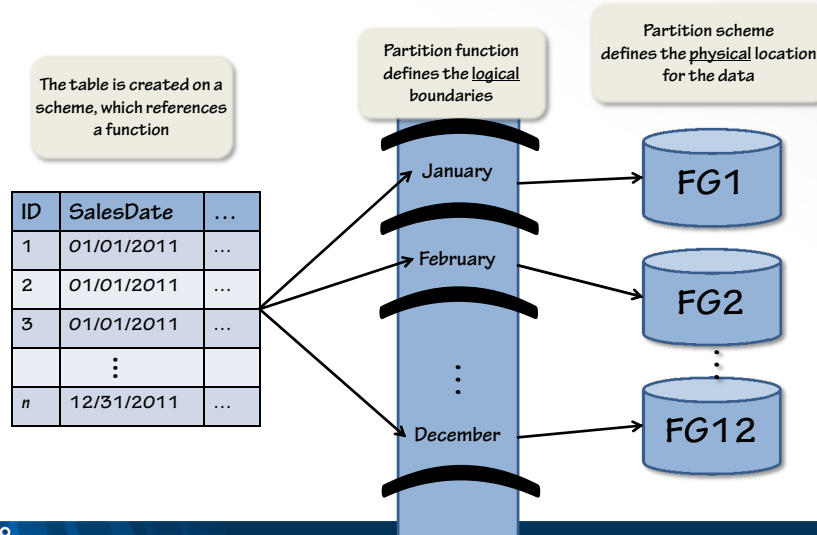
Verify Constraint is “Trusted”

- Check to see if constraint is trusted:
`OBJECTPROPERTY(object_id('constraint'),
'CnstIsNotTrusted')`
- Check to see if any data violates constraints:
`DBCC CHECKCONSTRAINTS('charge')`
`DBCC CHECKCONSTRAINTS('constraint')`
- However, even if DBCC CHECKCONSTRAINTS returns NO errors or warnings about your data... your constraint is still NOT trusted!
- You must DROP/re-CREATE or re-CHECK

Partitioned View Challenges

- Administration of separate tables, index creation on many tables
- Constraints (and business logic) are not guaranteed with partitioned views: must be managed on all tables
- Constraints must be trusted: if you disable constraints make sure you not only “enable” but recheck the data
- Queries can work well but data modifications may not work through the view
 - Base tables can’t have identity (to insert through view)
 - Partitioning column must be leading column of PK
- Harder for the optimizer to optimize as tables could be different (some problems with optimizing certain predicates):
 - <http://blogs.msdn.com/craigfr/archive/2006/11/27/introduction-to-partitioned-tables.aspx> (<http://bit.ly/Qyu6Zp>)
 - <http://blogs.msdn.com/sqlcat/archive/2006/02/17/Partition-Elimination-in-SQL-Server-2005.aspx> (<http://bit.ly/Qyu4Rx>)

Partitioned Tables



Range Partitioned Tables

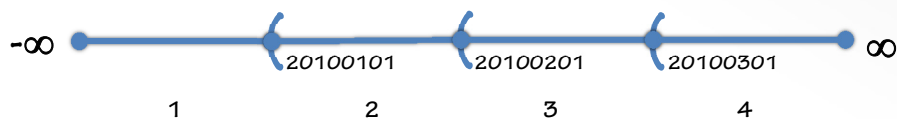
- Scripts from samples, lab and whitepaper (updated) in file: [PartitioningScripts.ssmss1n](#)
- Step 1: Create filegroups
- Step 2: Create files in filegroups
- [Step 3: Create partition function](#)
- [Step 4: Create partition scheme](#)
- Step 5: Create table(s) on partition scheme
- Step 6: Verify data with catalog views (optional)
- Step 7: Add data to tables
 - SQL Server redirects data and queries to appropriate partition

Step 3: Partition Function

- Not related to scalar/table-valued functions
- Must define the data type over which the function will calculate
 - Partition function can partition over ONLY ONE column of a table
- Always includes the entire domain of possible values from negative infinity ($-\infty$) to infinity (∞)
- Table should use constraints to limit the domain of data values (they are not required as they are with partitioned views)

Partition Function

- Range “right” means the value is contained in the partition to the right



- Right-based partitions
 - Use lower boundaries
- n boundaries creates $n+1$ partitions
- One table, one set of indexes, simplified management (to a point)

Partition Function

Left or Right?

- Defines whether or not the first boundary point should fall to the LEFT or the RIGHT within the full domain of values ($-\infty$ to ∞)
- Values can use functions to calculate boundary point but boundary point will always be stored as a literal value based on calculation of the function at time PF created
- Only the boundary point is stored, not the expression used to calculate it
- Can see boundary points stored within the catalog view (sys.partition_range_values)

Partition Function

Left or Right?

- **Right**
 - Defines that the boundary point falls into the SECOND (RIGHT, of the first two) partition
 - All other boundary points follow this pattern
 - A “beginning point” makes more sense when using RIGHT
 - In date-based partitioning, this is an easier value to define
- **LEFT**
 - Defines that the boundary point falls into the FIRST (LEFT, of the first two) partition
 - All other boundary points follow this pattern
 - An “ending point” makes more sense when using LEFT
 - In date-based partitioning, this can be a very frustrating value with which to work

Choosing Upper Boundary (LEFT)

Added Complexity with Datetime Data



- Upper boundary is inclusive ($\leq$) so must be exact
- Datetime value of 23:59:59.999 is not “available” due to the precision with datetime data. If entered it will round to 00:00:00.000 of the next day
 - 23:59:59.999 rounds up to 00:00:00.000
 - 23:59:59.998 rounds down to 23:59:59.997
 - 23:59:59.997 can be stored and is the last explicit value supported
- The partition function should use either of these two values for the upper boundary:
 - date 23:59:59.997
 - DATEADD(ms, -3, date)

Create the Partition Function

Using LEFT



- Orders and Order Details will include the OrderDate column (yes, duplicated data needed in the Order Details table)
- Orders range from October 2002 through September 2004

```
CREATE PARTITION FUNCTION TwoYearDateRangePFN(datetime)
AS
RANGE LEFT FOR VALUES
( '20021031 23:59:59.997', -- Oct 2002
  '20021130 23:59:59.997', -- Nov 2002
  '20021231 23:59:59.997', -- Dec 2002
  '20030131 23:59:59.997', -- Jan 2003
  '20030228 23:59:59.997', -- Feb 2003
  ...
  '20040930 23:59:59.997') -- Sep 2004
```

Create the Partition Function

Using LEFT with DATEADD Function

 hidden slide
w/extra details

- Can use a function in place of a literal value
- Function is evaluated at creation and only the boundary point is stored
- See sys.partition_range_values for values

```
CREATE PARTITION FUNCTION TwoYearDateRangePFN(datetime)
AS
RANGE LEFT FOR VALUES
( dateadd (ms, -3, '20021101') -- Oct 2002
, dateadd (ms, -3, '20021201') -- Nov 2002
, dateadd (ms, -3, '20030101') -- Dec 2002
, dateadd (ms, -3, '20030201') -- Jan 2003
, dateadd (ms, -3, '20030301') -- Feb 2003
...
, dateadd (ms, -3, '20031201') -- Sep 2003
```



23

© SQLskills. All rights reserved.
<http://www.SQLskills.com>

Create the Partition Function

Using RIGHT

- Orders and Order Details will include the OrderDate column (*yes, duplicated data needed in the Order Details table*)
- Orders range from October 2002 through September 2004

```
CREATE PARTITION FUNCTION TwoYearDateRangePFN(datetime)
AS
RANGE RIGHT FOR VALUES
( '20021001', -- Oct 2002
'20021101', -- Nov 2002
'20021201', -- Dec 2002
'20030101', -- Jan 2003
'20030201', -- Feb 2003
...
'20040901') -- Sep 2004
```



24

© SQLskills. All rights reserved.
<http://www.SQLskills.com>

Partitioning: Date Column

- Avoids a programmability problem with datetime and partition switching in the sliding window scenario
- You no longer needs to write your partition boundaries with '23:59:59.997' to match the last time value in a day
- More details about the SQL Server 2005 limitation in the MSDN Whitepaper: Partitioned Tables and Indexes in SQL Server 2005
 - <http://msdn2.microsoft.com/en-us/library/ms345146.aspx>
- Examples using date in the SQL Server 2008 whitepaper:
 - <http://msdn.microsoft.com/en-us/library/dd578580.aspx>

Step 4: Partition Scheme

- Maps the partitions (number defined by the function) to the filegroups (and therefore files) with the database
- Because both the extreme left and extreme right boundary cases are included, a partition function with n boundary conditions actually creates n+1 partitions
- This first (in RIGHT) partition will remain empty as data slides/rolls forward

Step 4: Partition Scheme

- **You can have empty partitions**
 - In RIGHT-based partition functions, the first partition is empty
 - Pro – Your automation routines are the same from the first onward
 - Con – None, except a few extra partitions to define
 - If LEFT-based partition functions, the last partition is empty (same pros/cons)
- **If using an empty partition:**
 - Use a special "tiny" filegroup
 - Create a file/filegroup that's restricted to 2MB without autogrowth
 - Fast fail if you "MERGE" the wrong boundary point
 - Data should never reside in it
 - Constraints should be used to restrict the table's data.
- **Explicitly name a filegroup for the "active" partitions and then use "tiny" for the empty partition (currently and always to remain empty)**

Step 4: Partition Scheme

- **If we hold 2 years worth of Order/Order Details data in 24 partitions we will create a partition scheme to handle 25 partitions where the FIRST one will begin/remain empty:**

```
CREATE PARTITION SCHEME [TwoYearDateRangePScheme]
AS PARTITION TwoYearDateRangePFN TO
( [Tiny],
  [FG1], [FG2], [FG3], [FG4], [FG5],
  [FG6], [FG7], [FG8], [FG9], [FG10],
  [FG11], [FG12], [FG13], [FG14], [FG15],
  [FG16], [FG17], [FG18], [FG19], [FG20],
  [FG21], [FG22], [FG23], [FG24])
GO
```

Step 5: Create Table

```
CREATE TABLE Sales
(
    SalesDate    datetime2 NOT NULL
                CONSTRAINT DF_Sales_SalesDate
                DEFAULT SYSDATETIME()
    ...
) ON PartitionScheme(SalesDate)
```

- Column has to match data type of partition function
- Table is created on partition scheme
- Can be done with an online operation for an existing table by rebuilding the clustered index ON the partition scheme
 - Nonclustered indexes are un-aligned until manually rebuilt on the same (or aligned) scheme
 - Table is kept online but fast-switching is not allowed until nonclustered are also rebuilt and aligned

Step 6: Create Indexes

```
CREATE UNIQUE CLUSTERED INDEX SalesPK
ON Sales (SalesID)
WITH (DROP_EXISTING = ON, ONLINE = ON)
ON [Sales4Partitions_PS](SalesID)
```

- Can “move” a table to one or more filegroups as an online operation by rebuilding the clustered index on the new scheme WITH ONLINE = ON
 - Note: Online only when there are no LOB columns in the table (fixed in 2012)
- Secondary nonclustered indexes are NOT moved/partitioned
- New indexes are automatically aligned by default but existing indexes (prior to partitioning) must be rebuilt/re-created on the new scheme to be aligned properly
- If clustered index was created through a primary key constraint, you rebuild by using the constraint name and same definition – on new scheme (example above is a PK)
- Column has to match data type of partition function
- Unique indexes must contain the partitioning column as part of the key

Verify Partition Data/Location

- **Programmatically determine which partition...**
`SELECT $partition.PFN('date-to-modify')`
- **See all data across all partitions...**
`SELECT $partition.PFN(o.OrderDate) AS [Partition#]
 , min(o.OrderDate) AS [Min Order Date]
 , max(o.OrderDate) AS [Max Order Date]
 , count(*) AS [Rows In Partition]
FROM dbo.Orders AS o
GROUP BY $partition.PFN(o.OrderDate)
ORDER BY [Partition#]`
- **See all data across all partitions...**
`SELECT * FROM sys.dm_db_index_physical_stats(...)`

The Sliding Window Scenario

- **New data must come in**
- **Old data must be removed**
- **The table should only change it's overall data set... with simple metadata changes**
- **To create metadata ONLY changes**
 - Create a new table on the same filegroup where the partition resides
 - Structure it IDENTICALLY to that of the partitioned table
 - SWITCH the partition to live in the new table and the table to become the partition... this either brings data IN or OUT of the table

The Sliding Window Summary (1)

- If you're removing data, prepare the "staging out" table
- If you're bringing data in
 - Load/transform/cleanse – in a staging area
 - Move the data by building the clustered index on the destination filegroup
- Alter the partition scheme to SET NEXT USED (based on the filegroup where you created the clustered index in step 2)
- Split the function to add the new boundary point (this will put the boundary point on the filegroup specified in the scheme's next used setting). IMPORTANT NOTE: be sure that NO data needs to move (if using LEFT, then the splitting partitioning should be empty)

Note: none of the first 4 steps impact the actual table!

The Sliding Window Summary (2)

- Switch "IN" the new data
 - Switch "OUT" the old data
- Note: steps 5 and 6 can be in either order – and are FAST (re: metadata only operations)*
- Backup/remove the old data (drop the table)
 - Merge the boundary point (from the emptied [switched out] partition)
- NOTE: If the "merge" process is slow then it's probable that you had data in the partition being merged. Be careful never to split or merge where there's already data; this data will need to be relocated (and this process is slow).

The Sliding Window Key Points

1. The staging tables must match in every way
 1. Same column types
 2. Same indexes (and ALL indexes)
 3. SQL Server 2008, all indexed views and their indexes
2. The staging tables must be on the same filegroup at the time of the switch (can build the data in a staging area first and then move to destination)
3. Be sure that NO data needs to move... test!
4. Only special consideration for the table that you're switching IN: it must have a "trusted" constraint to guarantee the data matches the partition into which it's switching.

Partitioned Table Challenges

Special Considerations for PTs

- If RW portion is only current month then that's the only place where fragmentation will occur
- If current month is September then only rebuild September:

```
ALTER INDEX ChargesPTPK ON ChargesPT  
REBUILD PARTITION = 5  
WITH (ONLINE = ON)
```

Msg 155, Level 15, State 1, Line 3

'ONLINE' is not a recognized ALTER INDEX REBUILD PARTITION option.

Online Operations, Indexes and Partitioning

- Partitioning and online operations are Enterprise-only (Enterprise, Enterprise Eval, Developer)
- Table-level index rebuilds can be done online if there are no LOB columns in the table (fixed in 2012)
- Rebuilding/partitioning the clustered index does NOT rebuild any of the nonclustered indexes – these must be rebuilt on the new scheme as well
 - Until the nonclustered indexes are rebuilt the partitioned table will not support fast-switching; everything else will work!
 - Unique indexes must have the partitioning column as a member of the unique key (to support fast-switching). If you don't need fast-switching then you don't need to do this!

Partitioning: PVs vs. PTs

Partitioned views

- Any edition
- Lots of tables to administer
 - Must create/drop indexes on all base tables
 - Can have different indexes
 - Harder for the optimizer to optimize with so many indexes
 - Must verify business logic so that there are no gaps or overlapping values
 - Each table has [potentially] better statistics as the tables are smaller
- Can rebuild any of the tables ONLINE (if using EE)
- Can support multiple constraints on one or more columns

Partitioned tables

- Enterprise Edition only
- Only 1 table to administer
 - Only 1 table to create/drop indexes
 - All partitions have same indexes (which is easier for the optimizer to optimize)
 - Can create different indexes with filtered indexes
 - No possibility of errors (or gaps or overlapping values)
 - Table-level statistics can be less accurate for very large tables when there's a lot of skew to the data
- Partition-level rebuilds are offline but can rebuild the ENTIRE table online (not desirable)
- Can only support partitioning over a single column

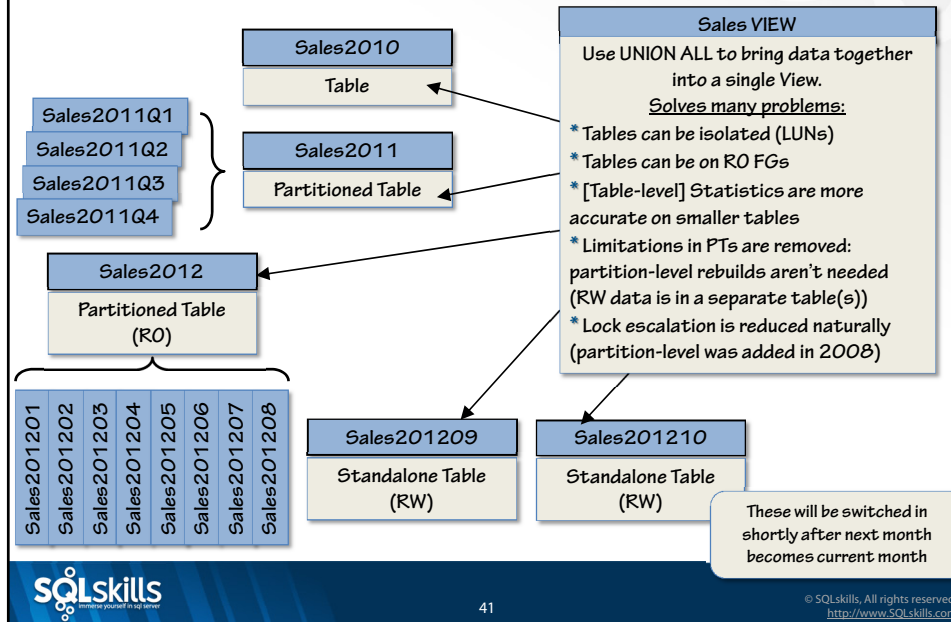
Other Features and Partitioning

- **iFTS**
 - PTs: cannot do fast-switching if PT has iFTS index(es)
- **Indexed views**
 - 2005 PTs: cannot do fast-switching if PT has IVs – must drop, switch, recreate (nightmare! fixed in 2008)
- **Identity columns**
 - PVs: cannot use UPVs for INSERT if they have an identity column on base tables (must use application directed INSERTs)
 - PTs: can have an identity
- **Multiple partitioning columns**
 - PTs: no, only one
 - PVs: yes, as many as makes sense!

Partition-Aligned Index Views

- **SQL Server 2005 does not support partition switching while index views are defined**
 - Drop the index(es) to switch partitions ☹
 - Then re-create the index(es) once switched
 - SLOW to do (and those queries using the indexed views are slow during this period too!)
- **SQL Server 2008 has partition-aligned indexed views**
 - Allows the switch to happen with a meta-data operation (including the indexed view partition data)
 - Much better support for Data Warehouse maintenance

Combining Technologies



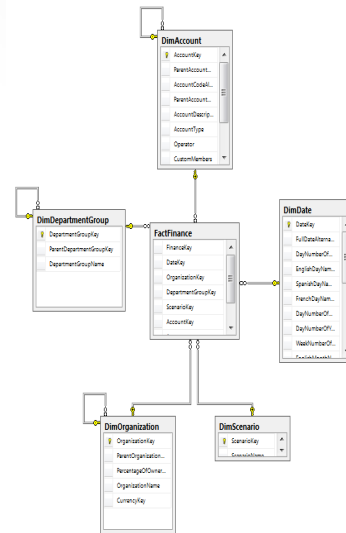
41

Partitioning for Performance & Online Index Rebuilds

- Separate read-only into a partitioned table or even multiple partitioned tables (eg. by year, re: possibly better stats depending on data distribution)
- Keep read-write as standalone table or additional partitioned table
- For queries, use a partitioned view to UNION ALL of the tables together
- For DML:
 - Use an updateable partitioned view – when possible
 - Or, use application directed inserts/updates/deletes (programmatically determining date) when an UPV is not possible
 - Or, consider using a synonym for the “current month” to simplify direct base table access
- Other benefits:
 - Partitioned tables are easier for the optimizer to optimize (same indexes)
 - Partitioned tables are easier to manage wrt creating/dropping indexes
 - Partitioned views allow differences in the data to be treated as such:
 - Can index RO one way (with more indexes), RW another (with fewer indexes)!
 - Statistics are separated between drastically different data (volatile v. static, current v. historical)
 - SQL Server 2005 lock escalation problems are reduced (SQL Server 2008 has partition-level lock escalation so this is less of a concern)

Improved Data Warehouse Query Performance The Case for Columnstore

- Columnstore indexes provide an easy way to significantly improve data warehouse and decision support query performance against very large data sets
- Performance improvements for “typical” data warehouse queries from 10x to 100x
- Ideal candidates include queries against star schemas that use filtering, aggregations and grouping against very large fact tables



How Are Performance Gains Achieved?

- Two complimentary technologies:
 - Storage
 - Data is stored in a compressed columnar data format (stored by column) instead of row store format (stored by row).
 - Columnar storage allows for less data to be accessed when only a sub-set of columns are referenced
 - Data density/selectivity determines how compression friendly a column is – example “State” / “City” / “Gender”
 - Translates to improved buffer pool memory usage
 - New “batch mode” execution
 - Data can then be processed in batches (1,000 row blocks) versus row-by-row
 - Depending on filtering and other factors, a query may also benefit by “segment elimination” - bypassing million row chunks (segments) of data, reducing I/O

Column vs. Row Store

- Row Store (Heap / B-Tree)
- Leaf-level pages consist of multiple “rows” as defined by the table or index
- Support compression but might not be as compressible

data page 1000

Product ID	OrderDate	Cost
310	20010701	2171.29
311	20010701	1912.15
312	20010702	2171.29
313	20010702	413.14

data page 1001

Product ID	OrderDate	Cost
314	20010701	333.42
315	20010701	1295.00
316	20010702	4233.14
317	20010702	641.22

Column vs. Row Store

- Column Store
- Only that column is stored on the same page – potentially HIGHLY compressible!
- Data is segmented into groups of 1000 rows for better batch processing
- SQL Server can do “segment elimination” (just like partition elimination) and further reduce the batch(es) processed!

ProductID	OrderDate	Cost
310	20010701	2171.29
311	...	1912.15
312	20010702	2171.29
313	...	413.14
314	...	333.42
315	20010703	1295.00
316	...	4233.14
317	...	641.22
318	...	24.95
319	...	64.32
320	20010704	1111.25
321	...	

data page 2000 data page 2001 data page 2002

Batch Mode

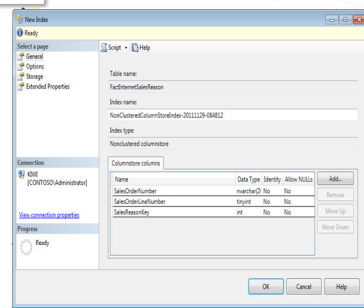
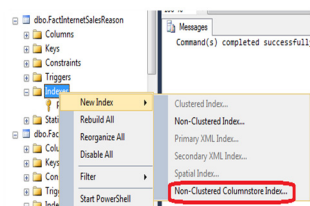
- **Allows processing of 1,000 row blocks as an alternative to single row-by-row operations**
 - Enables additional algorithms that can reduce CPU overhead significantly
 - Batch mode “segment” is a partition broken into million row chunks with associated statistics used for Storage Engine filtering
- **Batch mode can work to further improve query performance of a columnstore index, but this mode isn’t always chosen:**
 - Some operations aren’t enabled for batch mode:
 - E.g. outer joins to columnstore index table / joining strings / NOT IN / IN / EXISTS / scalar aggregates
 - Row mode might be used if there is SQL Server memory pressure or parallelism is unavailable
 - Confirm batch vs. row mode by looking at the graphical execution plan

Creating a Columnstore index

- **T-SQL**

```
CREATE NONCLUSTERED COLUMNSTORE INDEX [IX_CS_FactProductInventory]
ON dbo.FactProductInventory
(
    ProductKey,
    DateKey,
    UnitCost,
    UnitsIn,
    UnitsOut,
    UnitsBalance
);
```

- **SSMS**



Good Candidates for Columnstore Indexing

- **Table candidates:**
 - Very large fact tables (for example – billions of rows)
 - Larger dimension tables (millions of rows) with compression friendly column data
 - If unsure, it is easy to create a columnstore index and test the impact on your query workload
- **Query candidates (against table with a columnstore index):**
 - Scan vs. seek (columnstore indexes don't support seek operations)
 - Aggregated results far smaller than table size
 - Joins to smaller dimension tables
 - Filtering on fact / dimension tables – star schema pattern
 - Sub-set of columns (being selective in columns vs. returning ALL columns)
 - Single-column joins between columnstore indexed table and other tables
 - (NOTE: joins to smaller dimensions NOT large tables)

Columnstore Summary

- **SQL Server 2012 offers significantly faster query performance for data warehouse and decision support scenarios**
 - 10x to 100x performance improvement depending on the schema and query
 - I/O reduction and memory savings through columnstore compressed storage
 - CPU reduction with batch versus row processing, further I/O reduction if segmentation elimination occurs
 - Easy to deploy and requires less management than some legacy ROLAP or OLAP methods
 - No need to create intermediate tables, aggregates, pre-processing and cubes
 - Interoperability with partitioning

Review

- Horizontal partitioning strategies
 - Partitioned views
 - Partitioned tables
- Implementing the sliding window scenario
- Partitioning design techniques combined
- The case for columnstore indexes

Questions!