

An aerial photograph of a vast, dense forest covering a mountain slope. The trees are in various shades of green, suggesting a mix of species and possibly some autumnal changes. The terrain is rugged, with visible ridges and valleys. The text is overlaid on the upper half of the image.

# Don't Bite Off More Than You Can Chew – Take It in Chunks

Erland Sommarskog  
SQL Server MVP



# Erland Sommarskog

Independent consultant based in Stockholm

SQL Server MVP since 2001

[esquel@sommarskog.se](mailto:esquel@sommarskog.se)

<http://www.sommarskog.se>

Slides and scripts are available on

<http://www.sommarskog.se/present>



# Agenda

- When to use chunking?
  - Don't bite off more than you can chew.
  - Be nice to others.
- Implementing chunking.
  - Performance and pitfalls.
- Using chunking for error handling.
  - When all-or-nothing is not what you want.

# Don't Bite Off More than You Can Chew

Examples of operations you may want to perform on an entire table or the better part of it:

- `ALTER TABLE` to change `int` to `bigint`.
- Copying the data to a new version of the schema.
- Updating a new column for all rows.
- Purging 50% of the rows.

Say now that the table is 300 GB in size...



# Working with All Data at Once?

What can happen?

- Things may go just fine. :-)
- The log file may grow violently.
- Tempdb – both data and log file – may grow out of bounds to fit temp tables, spool operators etc.
- The buffer cache may get thrashed, because the amount of data you work with exceeds the available RAM with a factor X
- If you have to cancel the operation, the rollback can take a long time.

# Split up the Work in Chunks

- Divide the work into chunks operating on subsets of the data.
- The chunk size should be small enough to avoid log-file and tempdb explosions.
- Don't make it too small: chunking adds overhead, and bigger chunks are more efficient.
- What is the right size? It depends!
  - Current size of log file and tempdb (and what growth that is permissible).
  - Don't overlook the size in bytes. One million rows with a LOB column with an average size of 1 MB, that's 1 TB!
  - My own standard value is 5 million rows – but your mileage may vary.



# Transaction-Log Management

- When operating on large amounts of data, you will produce a lot of transaction log – chunking or not.
- Easy way out: simple recovery – but not very often feasible in production.
- With full recovery, you may have to increase the backup frequency, maybe as often as every minute.
  - With the normal schedule, the log may still explode.
- Add BACKUP LOG to your chunking script? Only if you write to the same place and name convention as the scheduled log backup.

# Be Nice to Others

- Your operation as such could be carried out in a single step.
- But in a live system you may impact other users.
  - By taking up too much resources (CPU, memory etc).
  - By causing blocking and/or deadlocks.
- Split up the operation in chunks, with a fairly *small* chunk size.
- Your operation will be less efficient – but the overall system health will be better.



# Avoiding Table and Page Locks

- Often when you work in a live system, you want to avoid table and page locks. This implies that the chunk size must be at most 5000 to avoid lock escalation.
- Run a test operation with different chunk sizes in a transaction and inspect your locks in `sys.dm_tran_locks`.
  - You should see X locks on KEY and RID resources *only*.
  - If you see X locks on OBJECT or PAG, your chunk size is too big.
  - IX (intent locks) on OBJECT and PAG are OK. They are only saying “I’m in here somewhere”.

[01\\_locktest.sql](#)



# Challenges with Chunking

- You need to write more code.
  - ⇒ More code that can have bugs.
  - ⇒ More code to test.
- How is the application affected?
- What if the work is interrupted half-way through? How to deal with this?
- Performance. Done wrong, chunking can severely impair your performance.



# Is This a Good Chunking Operation?

```
DECLARE @chunksize int = 5000000
DECLARE @rowc      int = @chunksize
WHILE @rowc = @chunksize BEGIN
    UPDATE TOP (@chunksize) SomeBigTable
    SET      NewColumn = <somevalue>
    WHERE    NewColumn IS NULL
    SET @rowc = @@rowcount
END
```

It is not. Each iteration will have to scan more and more rows to find rows to update.



# Indexing Is Critical

- You must always define your chunks over an indexed column.
- Else you will scan the big table over and over again – guaranteed to be worse than not chunking at all.
- The larger the chunk size, the more imperative the clustered index becomes.
  - The optimizer may frown at doing five million key lookups.
- For smaller chunk sizes a non-clustered index may work.
- Business logic or joins to other tables may constrain you.



# Introducing BigTrans

- 31 million rows, 2.2 GB (courtesy of Adam Machanic).

Colname	Type	NULL	Key	Change to	
TrnId	int	NOT NULL	PK	bigint	
ProdId	int	NOT NULL	FK		
TrnDate	datetime	NULL		date	NOT NULL
Quantity	int	NULL		smallint	NOT NULL
Amount	money	NULL		decimal(20,2)	NOT NULL

- NC index on (ProdId, TrnDate) INCLUDE (Amount, Qty).



# How to Alter BigTrans?

- With ALTER TABLE?
  - PK and NC index must be dropped and recreated at end.
  - Four ALTER TABLE ALTER COLUMN statements, all rebuilding the table.
  - Total time 984 s and log grew by 20 GB. (Starting on 700 MB.)
- Created NewTrans with CI, but without FK and NC index.
  - Copied all rows in a single INSERT statement.
  - Created foreign key and non-clustered index.
  - This took 68 sec, 7 GB in T-log growth.
- Can we improve this further with chunking?

# A Generic Way to Drive Chunks

```
CREATE OR ALTER PROCEDURE insert_top_plain @chunksize int AS  
    DECLARE @minID int, @maxID int  
    SELECT @minID = MIN(TrnID) FROM BigTrans
```



```

CREATE OR ALTER PROCEDURE insert_top_plain @chunksize int AS
DECLARE @minID int, @maxID int
SELECT @minID = MIN(TrnID) FROM BigTrans
WHILE @minID IS NOT NULL BEGIN
    SELECT @maxID = MAX(TrnID) FROM
        (SELECT TOP(@chunksize) TrnID
         FROM BigTrans
         WHERE TrnID >= @minID
         ORDER BY TrnID ASC) AS B
    INSERT NewTrans(TrnID, ProdID, TrnDate, Qty, Amount)
        SELECT TrnID, ProdID, TrnDate, Qty, Amount
        FROM BigTrans
        WHERE TrnID BETWEEN @minID AND @maxID
    SELECT @minID = MIN(TrnID) FROM BigTrans
    WHERE TrnID > @maxID
END -- Recreate NC index and FK.

```

# A Generic Way to Drive Chunks

```
CREATE OR ALTER PROCEDURE insert_top_plain @chunksize int AS
DECLARE @minID int, @maxID int
SELECT @minID = MIN(TrnID) FROM BigTrans
WHILE @minID IS NOT NULL BEGIN
    SELECT @maxID = MAX(TrnID) FROM
        (SELECT TOP(@chunksize) TrnID FROM BigTrans
         WHERE TrnID >= @minID
         ORDER BY TrnID ASC) AS B
    INSERT NewTrans(TrnID, ProdID, TrnDate, Qty, Amount)
        SELECT TrnID, ProdID, TrnDate, Qty, Amount FROM BigTrans
        WHERE TrnID BETWEEN @minID AND @maxID
    SELECT @minID = MIN(TrnID) FROM BigTrans WHERE TrnID > @maxID
END; -- Create NC index and foreign key.
```



# Some Performance Data

- All times include FK and NC index.
- Smaller chunk sizes yield longer execution time.
- Chunk size  $\leq$  1 million, no log growth.
- Best chunk size slightly faster than all-at-once.
- Keep in mind – table is only 2 GB!
- Data is from my laptop. Your testing may yield different results.

Chunk size	Exec time (s)	Log Grwth (MB)
1 000	299	67
5 000	119	0
30 000	64	0
200 000	62	0
1 000 000	61	0
6 000 000	62	939
All	68	7046

# Optimization Considerations

- The optimizer does not know the values of @minID and @maxID and will therefore make a blind assumption about hit rate.
- This can adversely affect performance, particularly if you are joining to other tables of any size.
- Simple way out: `OPTION(RECOMPILE)`.
  - Considerable overhead for small chunk sizes.
- Alternative: make the chunk boundaries into parameters by pushing the operation to an inner scope.



# Wrap the Operation in Dynamic SQL

```
EXEC sp_executesql
    N'INSERT NewTrans(TrnID, ProdID, TrnDate, Qty, Amount)
      SELECT TrnID, ProdID, TrnDate, Qty, Amount
      FROM    dbo.BigTrans
      WHERE   TrnID BETWEEN @minID AND @maxID',
    N'@minID int, @maxID int', @minID, @maxID
```

@minID and @maxID are now parameters and the optimizer can sniff their values for a better cached plan.

# Effects of Recompile and Dynamic SQL

- Drastic improvement with dynamic SQL!
- RECOMPILE also gives some improvement, but compilation overhead costly.
- This is not overhead, but an optimization that backfires.
- Data is from tests in early 2019 on SQL 2016. New tests on SQL 2019 are not equally startling, but pattern is the same.

Chunk size	Plain	Recompile	Dyn. SQL
1 000	299	179	99
5 000	119	136	77
30 000	64	81	64
200 000	62	78	63
1 000 000	61	76	62
6 000 000	62	76	63



# Looking at a Different Operation

Initialise a new column TotalAmount added to BigOrders.

```
UPDATE BigOrders
SET     TotalAmount = OD.Amount*(1-O.Discount/100) + O.Freight
FROM    BigOrders O
JOIN    (SELECT OrderID, SUM(Quantity * UnitPrice) AS Amount
        FROM    BigDetails
        GROUP BY OrderID) AS OD ON O.OrderID = OD.OrderID
```

BigOrders: 5.1 mln rows, 1 GB.

BigDetails: 15 mln rows, 3.8 GB.

# Dynamic SQL and Statistics

- Dynamic SQL is slower for small chunk sizes. Why?
- BigOrders has sampled stats, and lowest ID in histogram is 13 063.
- Estimate is a single row, and plan is not optimal for 1000/5000 rows.
- With the range inside the histogram, performance is better.

Chunk size	Plain	Recomp.	Dyn. SQL
1 000	52	68	131
5 000	46	42	125
25 000	38	34	31
125 000	38	37	38
1 000 000	36	37	37



# Dynamic SQL and Statistics

- Dynamic SQL is slower for small chunk sizes. Why?
- BigOrders has sampled stats, and lowest ID in histogram is 13 063.
- Estimate is a single row, and plan is not optimal for 1000/5000 rows.
- With the range inside the histogram, performance is better.
- `UPDATE STATISTICS BigOrders PK_BigOrders WITH FULLSCAN` resolved the issue, but may always not be feasible.

Chunk size	Plain	Recomp.	Dyn. SQL
1 000	52	68	52
5 000	46	42	42
25 000	38	34	35
125 000	38	37	41
1 000 000	36	37	41

# Dynamic SQL/Recompile as Safeguards

- Same operation, now with the clustered index on the BigOrders table on CustomerID, and the PK (OrderID) is non-clustered.
- Recompile/Dynamic SQL is a safeguard against disasters.
- In passing: chunking over NC index slightly better than CI due to alignment with BigDetails.

	Over CustomerID			Over OrderID		
Chunk size	Plain	Re-comp.	Dyn. SQL	Plain	Re-comp.	Dyn. SQL
1 000	4341	127	96	1706	140	107
5 000	655	97	93	376	109	79
25 000	223	81	77	102	70	56
125 000	85	84	88	49	52	58
1 000 000	48	49	46	38	41	37



# Multi-Column Keys

- How to do chunking over BigDetails, PK/CI on (OrderID, ProductID), up to 600 products per order?
- Ignore ProductID and run a TOP loop over OrderID only?
- Chunks may have up to 600 rows extra – not a big deal.
- You want to stick to this pattern as far as possible!

# Multi-Column Keys, cont'd

- What if there can be 10 000 products per order, and you must not exceed a chunk size of 1 000?
- You need something else, obviously.
- Extend the logic with TOP to use two or more keys?
- Code becomes very complex already at two keys and therefore unattractive.

[04\\_multicolchunking.sql](#)



# Defining Chunks in a Temp Table

```
CREATE TABLE #chunks (OrderID    int    NOT NULL,  
                        ProductID int    NOT NULL,  
                        ChunkNo    int    NOT NULL,  
                        INDEX clu UNIQUE CLUSTERED (ChunkNo, OrderID, ProductID))  
  
INSERT #chunks SELECT OrderID, ProductID,  
    row_number() OVER(ORDER BY OrderID, ProductID) / @chunksize  
FROM    BigDetails
```

```

CREATE TABLE #chunks (OrderID    int    NOT NULL,
                        ProductID int    NOT NULL,
                        ChunkNo   int    NOT NULL,
                        INDEX clu UNIQUE CLUSTERED (ChunkNo, OrderID, ProductID))
INSERT #chunks SELECT OrderID, ProductID,
    row_number() OVER(ORDER BY OrderID, ProductID) / @chunksize
FROM BigDetails
DECLARE chunkcur CURSOR STATIC LOCAL FOR
    SELECT DISTINCT ChunkNo FROM #chunks
OPEN chunkcur
WHILE 1 = 1 BEGIN
    FETCH chunkcur INTO @chunkno IF @@fetch_status <> 0 BREAK
    UPDATE BigDetails SET UnitPrice = UnitPrice * 1.2
    WHERE EXISTS (SELECT * FROM #chunks c
        WHERE c.ChunkNo      = @chunkno
          AND c.OrderID      = BigDetails.OrderID
          AND c.ProductID    = BigDetails.ProductID)
END

```



# Define Chunks in a Temp Table

```
CREATE TABLE #chunks (OrderID    int    NOT NULL,  
                        ProductID int    NOT NULL,  
                        ChunkNo   int    NOT NULL,  
                        INDEX clu UNIQUE CLUSTERED (ChunkNo, OrderID, ProductID))  
INSERT #chunks SELECT OrderID, ProductID,  
    row_number() OVER(ORDER BY OrderID, ProductID) / @chunksize  
FROM    BigDetails
```

- Each chunk has a contiguous set of keys for efficient join.
- Ran 1.3 to 5.6 times longer than the “triple” TOP loop in my tests.
  - Caveat: Profile of the table matters a lot for the result.
- We’re sacrificing efficiency for simplicity.

# The “Big” and the “Small” Temp Table

- Previous slide was the “big” temp table.
- Has as many rows as the actual table, but is narrower.
- Requires a full scan to fill – can block writers.
- Not always a good solution, but still a viable technique – it’s simple!
- Alternative: the “small” temp table, which you fill up one chunk at a time.
  - A little more complex, but leaner.



# The “Small” Temp Table, pt. 1

```
DECLARE @CurOrderID int, @LastProdID int, @rowcnt int = @chunksize
CREATE TABLE #chunk (OrderID int NOT NULL, ProductID int NOT NULL,
                      PRIMARY KEY (OrderID, ProductID))
SELECT @CurOrderID = MIN(OrderID) - 1 FROM BigDetails
WHILE @rowcnt = @chunksize BEGIN
    TRUNCATE TABLE #chunk
    INSERT #chunk(OrderID, ProductID)
        SELECT TOP (@chunksize) OrderID, ProductID FROM BigDetails
        WHERE OrderID = @CurOrderID AND ProductID > @LastProdID OR
              OrderID > @CurOrderID
        ORDER BY OrderID, ProductID
    SELECT @rowcnt = @@rowcount
```

# The “Small” Temp Table, pt. 2

```
UPDATE BigDetails
SET     UnitPrice = UnitPrice * 1.2
FROM    BigDetails BD
WHERE   EXISTS (SELECT * FROM #chunk t
                WHERE t.OrderID    = BD.OrderID
                AND   t.ProductID = BD.ProductID)
SELECT TOP 1 @CurOrderID = OrderID, @LastProdID = ProductID
FROM      #chunk
ORDER BY  OrderID DESC, ProductID DESC
END
```

- In my tests, 1.3 to 3.4 times slower than the “triple” TOP loop.
- 0.2 to 1.3 of the time for the “big” temp table.



# Considerations on Application Impact

- If you don't do chunking, all is one transaction → application will see all or nothing of the operation.
- But if you do chunking, how will the application behave if it sees data that is processed only half-way?
- Even more pronounced if operation is interrupted, accidentally or purposely.
- Maybe you can shrug your shoulders.
- ...or you may have to make substantial changes in the application.

# Resuming Chunking

[05\\_restart\\_strategies.sql](#)

- Just ignore and start over.
  - Works well with purges.
  - Also with absolute updates, but you will redo work.
- Use logic of your operation to find out where you were.
  - For instance, finding the last row you inserted.
- Add a help table to track where you are (not in tempdb!).
- Restore a backup.
  - When nothing else is safe, for instance with a relative update, and you did not plan ahead.



# Chunking Inside a Transaction?

- Sounds crazy, contradicts the purpose in most cases.
- There is a reason for chunking I have not mentioned yet: a query plan where the performance does not grow linearly.
  - Eg. processing 1 000 rows takes 50 ms, 10 000 rows takes two seconds.
  - Or a plan that grows linearly to some point, but then spills to disk.
- Best would be address the query, but it may not be feasible.
- I have encountered this situation.
  - Removing the transaction would have had big application impact.

# The Risk for Bugs

Introducing chunking means increased complexity, and coding casually, you may:

- Skip a row between chunks or process the same row in two chunks.
- Miss the last few rows.
- Test on small data set with different chunk sizes.
  - Last chunk full minus one, exactly full, just a single row.
- Review your code.
- If possible, add assertions to check.
  - Eg. when copying rows to a new table, check row count.



# Testing for Performance

- Testing all possible variations is usually not viable – takes too much time.
- You will need to make a choice from gut feeling – but test that gut feeling.
- Keep in mind: you are not looking for the very best performance – you want to stay clear of the disasters.
- Execution time is not all – also monitor log-space and tempdb consumption.

# Some Index Considerations

- If possible, disable NC indexes not needed for the chunking operation and rebuild to re-enable later – this can boost performance a lot.
- Adding indexes to support the chunking operation?
  - Yes, it can pay off. You create it once and seek it many times.
- Changing the clustered index?
  - Trade-off. Can speed up the chunking part a lot, but recreating the index (twice!) is expensive in full recovery and can fill up the log.
  - SQL 2019 introduces resumable index creation – which is chunking behind the scenes.



# Using Chunks to Handle Errors

- Sometimes all-or-nothing is not what we want.
- Example: read 70 000 orders from a file. If one order cannot be inserted because of incorrect data, we still want the others.
- Process one order at a time?
- No! Divide the orders into chunks.
- On error in a chunk, re-divide into smaller chunks and retry.
- Eventually a few orders will be processed one-by-one.

[06\\_errhande-loop.sql](#)  
[07-errhandle-chunk.sql](#)

# Picking the Initial Chunk Size

- If you set the initial chunk size too high, about every chunk will error out and roll back – not efficient.
- Rule of thumb: if you expect one row out of  $N$  to error out, set the initial chunk size to  $N/10$ . (Assuming no other limitations.)
  - Determining  $N$  may require some good gut feeling.
- Do errors come in bursts or are they scattered? This can affect your choice.
- Divide into smaller chunks by 100 to 1000 at a time.



# Summary I

- Chunking is nothing you use every day – it's a trick you have in your toolbox for large operations.
  - To keep transaction log and tempdb in check.
  - To avoid conflicts with the rest of the system.
- Make sure that you have transaction-log backups running!
- Indexing is critical – your chunking must always follow an index, preferably the clustered index. Repeating scans is very costly.

# Summary II

- The TOP method is a generic way to drive the chunks.
  - Works with any indexable data type.
  - Works even with non-unique indexes to some extent.
- For multi-column keys, work with the second level if you really have to!
  - Defining chunks in a temp table is less efficient than extending the TOP method – but oh so much simpler.
- Make interval variables known to the optimizer with OPTION (RECOMPILE) or sniffable through dynamic SQL
  - Dynamic SQL may benefit from FULLSCAN statistics.



# Summary III

- Don't forget to consider how applications may be affected.
- Design your loops to be able to restart where they were interrupted.
  - If needed, create a help table to track where you were.
- Don't forget to test!
  - For correctness...
  - ...and performance.

# The Last Chunk

Erland Sommarskog  
[esquel@sommarskog.se](mailto:esquel@sommarskog.se)

Slides and scripts on <http://www.sommarskog.se/present>.

BigDB is here: <http://www.sommarskog.se/present/BigDB.bak>.  
(Backup size = 3 GB, DB size = 20 GB. Requires SQL 2016.)