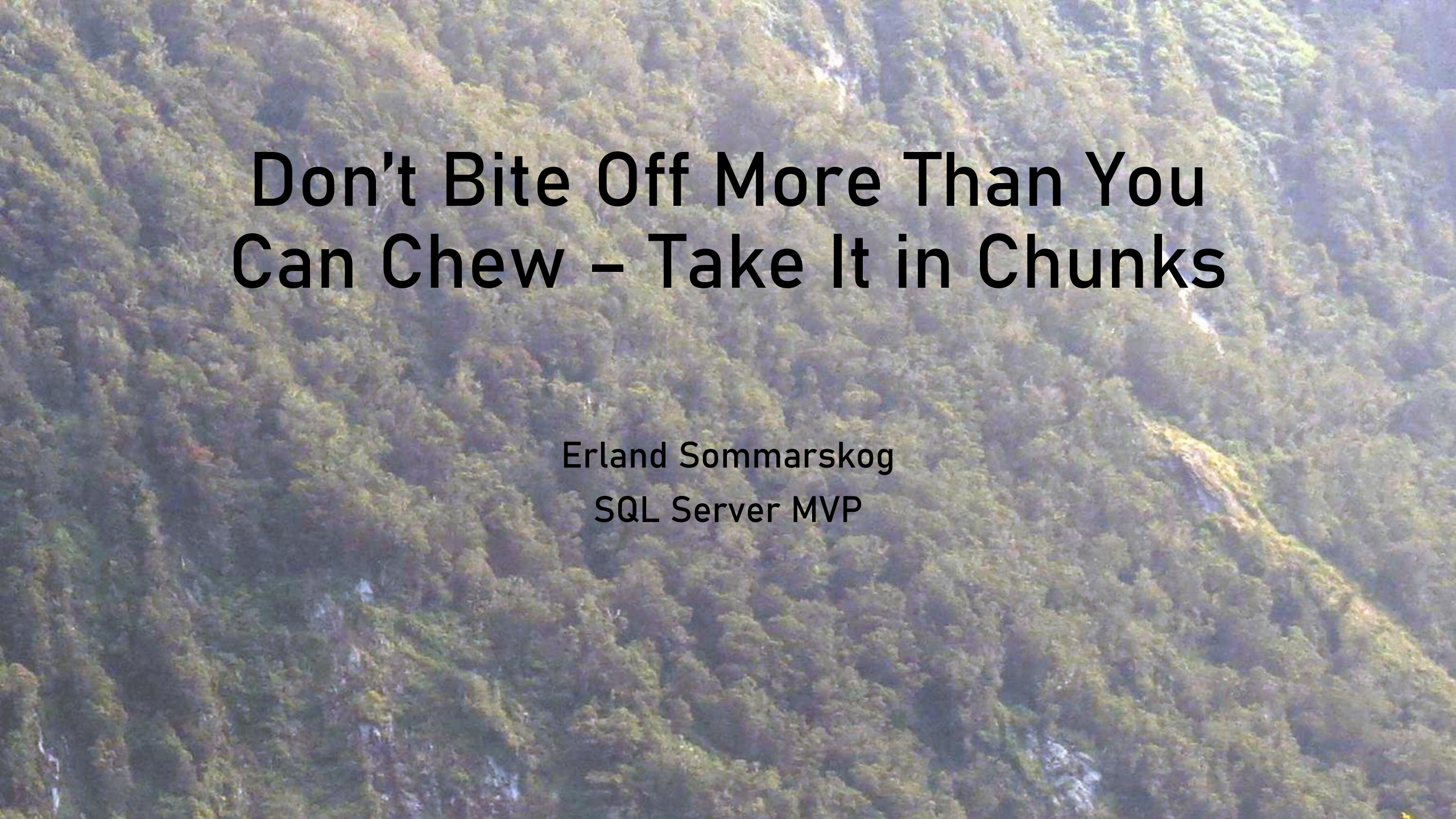


An aerial photograph of a vast, dense forest covering a mountain slope. The trees are mostly green, with some patches of brown and yellow, suggesting autumn. The terrain is rugged, with visible ridges and valleys. The text is overlaid on the upper half of the image.

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

<http://www.sommarskog.se>

esquel@sommarskog.se

Slides and scripts are available on

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

and the SQL Saturday web site

Organizers



Andriy Pogorelov



Paul Stetsenko



Natalia Pogorelova



Sponsors



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).
 - Row size – what matters is the number of bytes, not the number of rows.
One million rows with a LOB column with an average size of 1 MB => 1 TB!
- Make sure that transaction-log backups are running!

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`. `01_locktest.sql`
 - 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”.

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?

```
CREATE PROCEDURE PurgeOldData @purgedate date,  
                               @chunksize int AS  
  
DECLARE @rowc int = @chunksize  
WHILE @rowc = @chunksize  
BEGIN  
    DELETE TOP(@chunksize) BigTrans  
    WHERE TrnDate < @purgedate  
    SELECT @rowc = @@rowcount  
END
```

Yes?

No?

It depends!

Indexing Is Critical

- In this example, TrnDate is not indexed.
- Without chunking, it is not a big deal – the table is scanned exactly once.
- With chunking, the table is scanned for each chunk. Very costly!
- Chunking must always be over indexed column(s).
- For big chunk sizes, it more or less has to be the clustered index.
- For smaller chunk sizes, a non-clustered index *may* work.

Introducing BigTrans

- BigTrans: 31 million rows, 2.2 GB (courtesy of Adam Machanic).
- Five columns: TrnID (PK, clust.), ProdID (FK), TrnDate, Amount and Qty.
- NC index on (ProdId, TrnDate) INCLUDE (Amount, Qty).
- Task: change the data type of four columns. [02_altertable.sql](#)
- With ALTER TABLE this took 984 s and log grew by 20 GB. (Original size 700 MB.)
- Copying all rows in one go to NewTrans + creating FK and NC index: 68 sec, 7 GB in T-log growth.

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 ≤ 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 value 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.

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

Multi-Column Keys

- Consider a table with a multi-column key (A, B, C...)
- Ignore the other keys and run a TOP loop over A only.
 - You get some extra rows in each chunk, but as long as the number of rows for a certain value of A is moderate this is a non-issue.
- You want to stick to this pattern as far as possible.
- However, for a chunk size of 1 000 and typically 10 000 rows for a single value of A, you need something else.
- 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) - 1) / @size  
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) - 1) / @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) - 1) / @size  
FROM    BigDetails
```

- Each chunk has a contiguous set of keys for efficient join.
- Ran 2.2–5.6 times longer than a plain TOP loop in my tests.
 - Caveat: Profile of the table matters a lot for the result.
- We're sacrificing efficiency for simplicity.
- Drawback: requires an initial scan of the table.

Alternative: Small Temp Table

Fill up the temp table chunk by chunk in the loop.

```
DECLARE @rowcnt int = @chunksize, @OrderID int, @ProdID int
CREATE TABLE #chunk (OrderID int NOT NULL, ProductID int NOT NULL,
                      PRIMARY KEY (OrderID, ProductID))
SELECT @OrderID = 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 = @OrderID AND ProductID > @ProdID OR
              OrderID > @OrderID
        ORDER BY OrderID, ProductID
    SELECT @rowcnt = @@rowcount
```

The Small Temp Table, cont'd

```
UPDATE BigDetails
```

```
...
```

```
SELECT TOP 1 @OrderID = OrderID, @ProdID = ProductID FROM #chunk  
ORDER BY OrderID DESC, ProductID DESC
```

- Avoids an initial full scan of the table.
 - Good when you want to be nice to others.
- In my tests, generally slower than the big temp table if not for all chunk sizes.
 - Again: performance depends on the profile of the table.
- More samples as well as performance data:

[04_multicolchunking.sql](#)

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 have to make substantial changes in the application.

What if Work is Interrupted?

`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?

- Contradicts the purpose in most cases.
- But there is a reason to 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, for one reason or another.
- And, yes, I have encountered this situation.

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.
- Execution time is not all – also monitor log-space and tempdb consumption.

General Perftest Caveats

- Test on a copy of the production database or equivalent.
- Make sure that you have the same settings:
 - Recovery model.
 - Snapshot in some form. (Includes AG with readable secondary!)
 - Accelerated Data Recovery (new in SQL 2019).
 - Matching or inferior hardware.
 - Available RAM should preferably be the same.
- All dependent on how critical it is that you don't run into surprises in production.

Using Chunks to Handle Errors

- Sometimes all-or-nothing is not what we want.
- Example: read 45 000 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 into chunks. On error, re-divide the chunk into smaller and try again, eventually leading to a chunk size of one.

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

Picking the Initial Chunk Size

- You need to have an idea of how common errors will be.
- 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.)
- Divide into smaller chunks by 100 to 1000 at a time.
- Do errors come in bursts or are they scattered? This can affect your choices.

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, only care about 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

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.
 - A temp table you fill up with chunk numbers initially may better be to do as a permanent table if you restart often.
- Don't forget to test!
 - For correctness
 - ...and performance.

Sponsors



The Last Chunk

Erland Sommarskog

esquel@sommarskog.se

Slides and scripts on <http://www.sommarskog.se/present> .
Also on the SQL Saturday web site.

BigDB is here: <http://www.sommarskog.se/present/BigDB.bak>.
(Backup size = 1.2 GB, DB size = 12 GB. Requires SQL 2016.)