SQL Server: Advanced Corruption Recovery Techniques

Module 2: DBCC CHECKDB Internals and Performance

Paul S. Randal
http://www.SQLskills.com/blogs/paul/
Paul@SQLskills.com

pluralsight
hardcore developer training

Introduction

- **Gaining more advanced knowledge of working with SQL Server corruption also involves knowing a bit more about how DBCC CHECKDB works and how to enhance its performance**

- **In this module we'll cover:**
  - Various internals of DBCC CHECKDB
  - Trace flags to aid DBCC CHECKDB performance
  - Parallelism and memory settings for DBCC CHECKDB

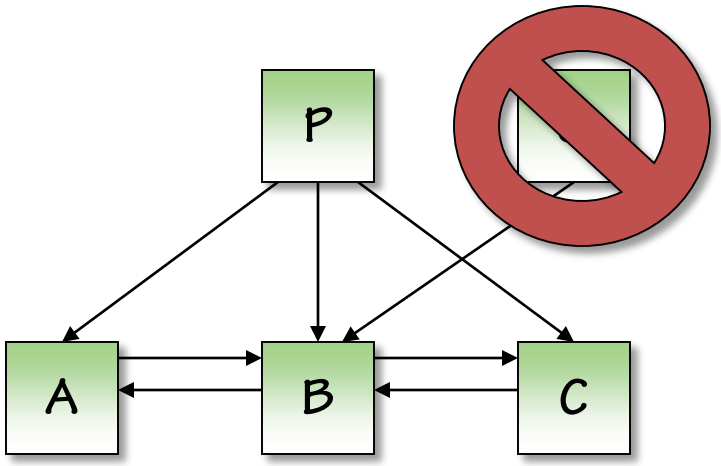
What Exactly Does DBCC CHECKDB Do?

- **Primitive checks of critical system tables**
- **Allocation checks**
- **Logical checks of critical system tables**
- **Logical checks of all other tables**
- **Metadata checks**
- **Service Broker data validation**
- **Indexed view, XML index, spatial index checks**
  - Only when the WITH EXTENDED_LOGICAL_CHECKS option is used, from SQL Server 2008 onward
- **If a repair option is specified, repairs are done at each stage if necessary and possible**
- **Uses the Query Processor extensively**

Fact Processing (1)

- **DBCC CHECKDB reads all allocated pages in the database in a very efficient manner**
- **It does not use the obvious link-following recursive algorithm as that would tend to $O(n^2)$**
- **It reads the pages in allocation order, driving its own readahead, and generates "facts" about what it has seen, which is O(n*log(n))**
  - E.g. facts about pages, b-tree linkages, off-row text linkages
  - Driven by an internal structure called the MultiObjectScanner
- **Each fact has a multi-part key**
  - Including object ID, index ID, allocation unit ID, page ID
- **Facts are passed to the Query Processor for efficient sorting and hashing, then passed back to DBCC for aggregation**
  - Aggregation means that facts cancel each other out
  - When extra or missing facts are detected, that's a corruption

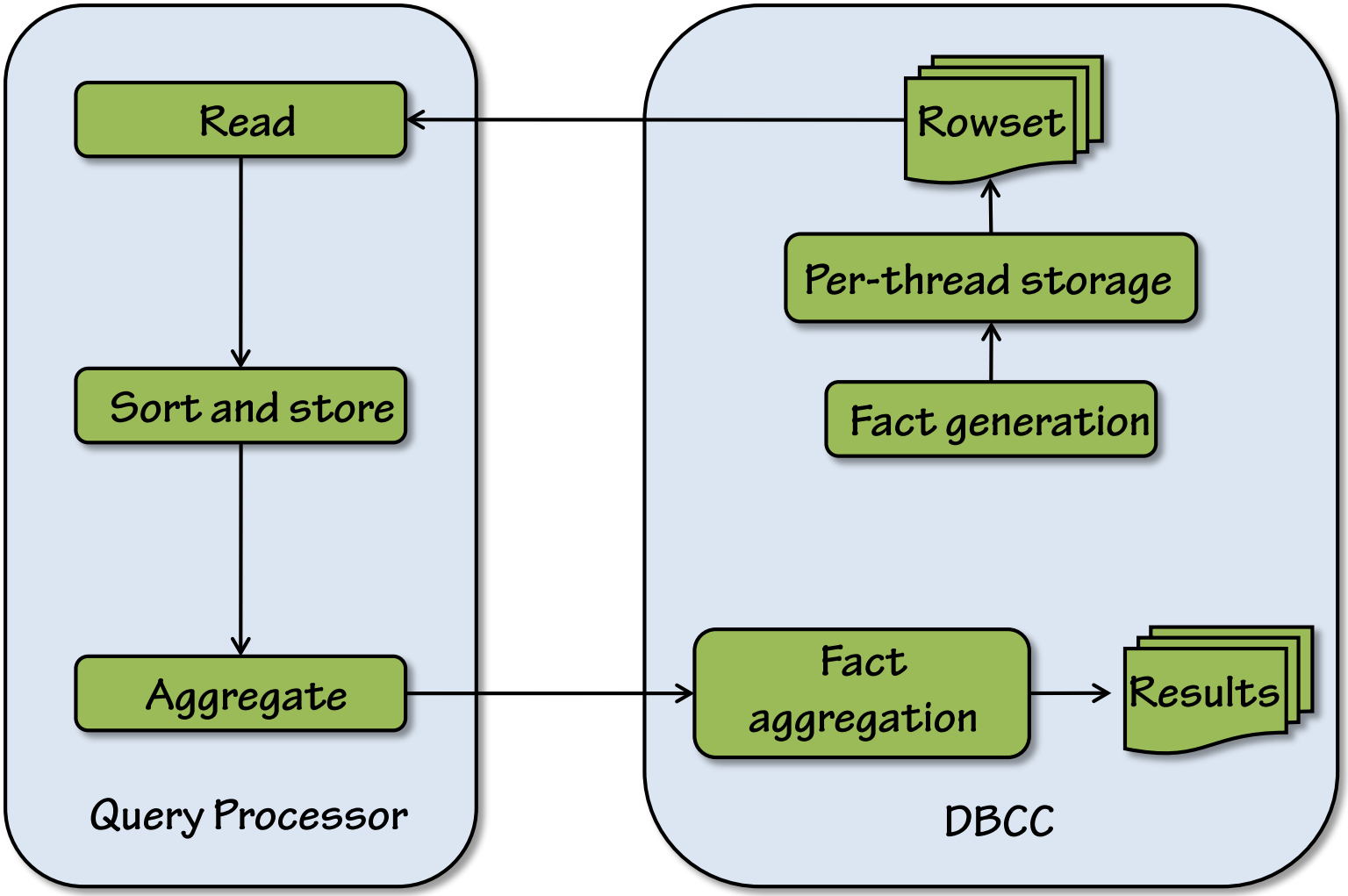
Fact Processing (2)



B-Actual
B-Sibling-A
B-Sibling-C
A-Actual
A-Sibling-B
P-Actual
P-Parent-A
P-Parent-B
P-Parent-C
Q-Actual
Q-Parent-B

B-Actual
A-Sibling-B
C-Sibling-B
P-Parent-B
Q-Parent-B ERROR!!

Fact Processing (3)



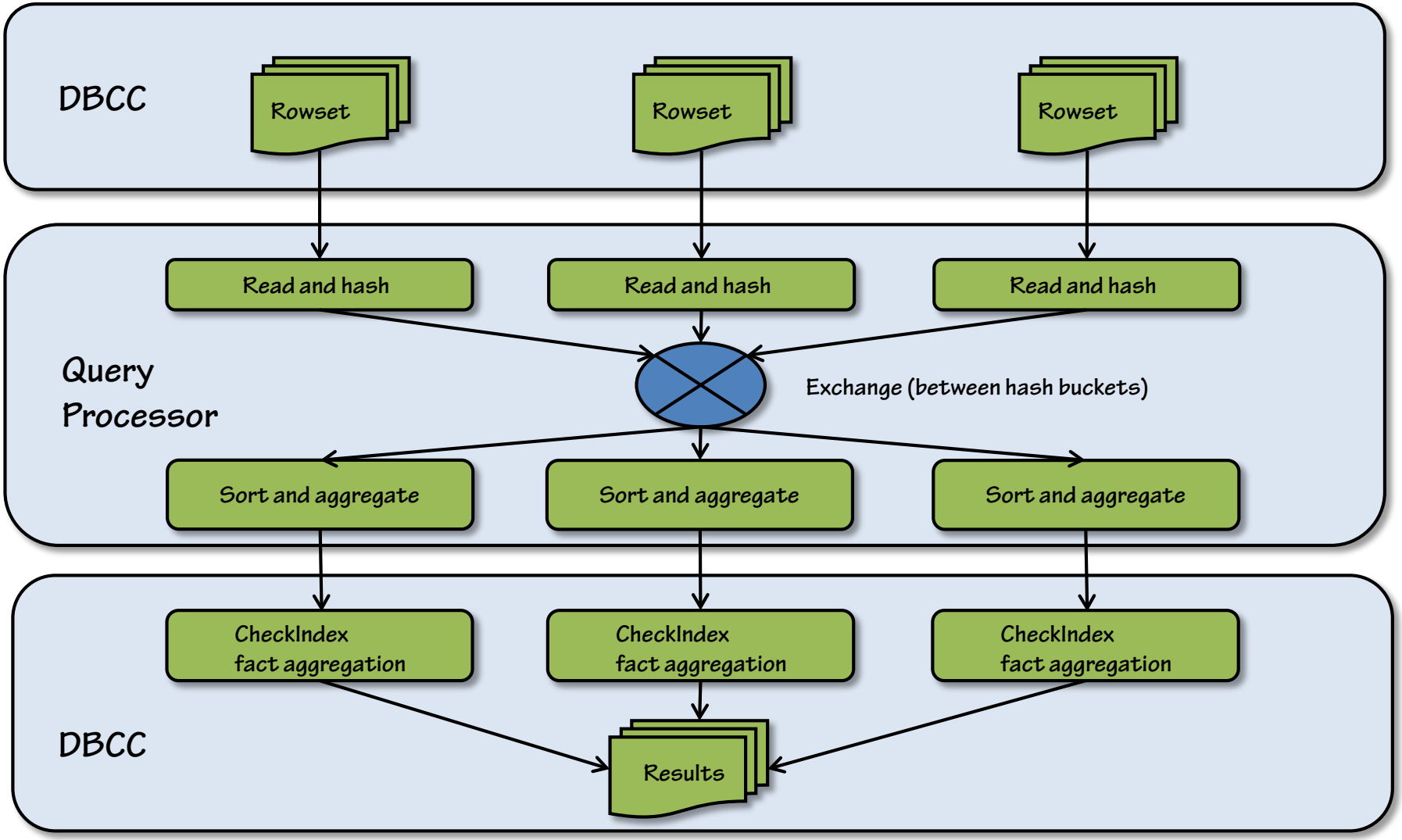
Batch Processing

- **The fact-storage worktable usually requires more memory than is available so it spills into *tempdb***
- **The batch size is limited to prevent excessive *tempdb* space usage**
  - A batch's "size" is the number of tables and associated indexes
- **The minimum batch size is a single table and its nonclustered indexes as everything about a table must be checked at the same time**
- **Tables are added to the batch until one of the following occurs:**
  - The total number of indexes in the batch becomes more than 512
  - The estimate for all necessary facts for the batch becomes more than 32 MB
  - If a repair option is specified, the batch size is always limited to a single table
- **Read and write performance of *tempdb* can greatly affect the run time**

Parallelism (1)

- **DBCC CHECKDB will run in parallel on Enterprise Edition**
  - Also applies to DBCC CHECKTABLE and DBCC CHECKFILEGROUP
  - DBCC CHECKALLOC and DBCC CHECKCATALOG are always single-threaded
- **Each batch executes the DBCC CHECKDB internal "query" and the Query Processor decides how far to parallelize**
  - Up to the limit imposed by the sp_configure MAXDOP setting or the Resource Governor workload group MAX_DOP setting
  - There is no option to set DBCC CHECKDB's degree of parallelism
- **Each thread is given a page to generate facts for, and then calls into the MultiObjectScanner to get the next page to process**
- **Threads then participate in fact aggregation once all pages have been read and processed for facts**
- **Parallelism can be disabled using documented trace flag 2528**
  - Greatly increases run time, but greatly decreases the resources required

Parallelism (2)



Nonclustered Index Checks (1)

- **One of the most interesting algorithms inside DBCC CHECKDB is the one driving nonclustered index checks**
- **Each table row must have one matching row in each nonclustered index, and vice-versa**
  - Taking into account filtered indexes from SQL Server 2008 onward
- **Easiest way to check this is:**
  - For each data record, look up each matching nonclustered index record
  - For each nonclustered index record, look up the matching data record
- **This is not how it's done, as this would be extremely inefficient**

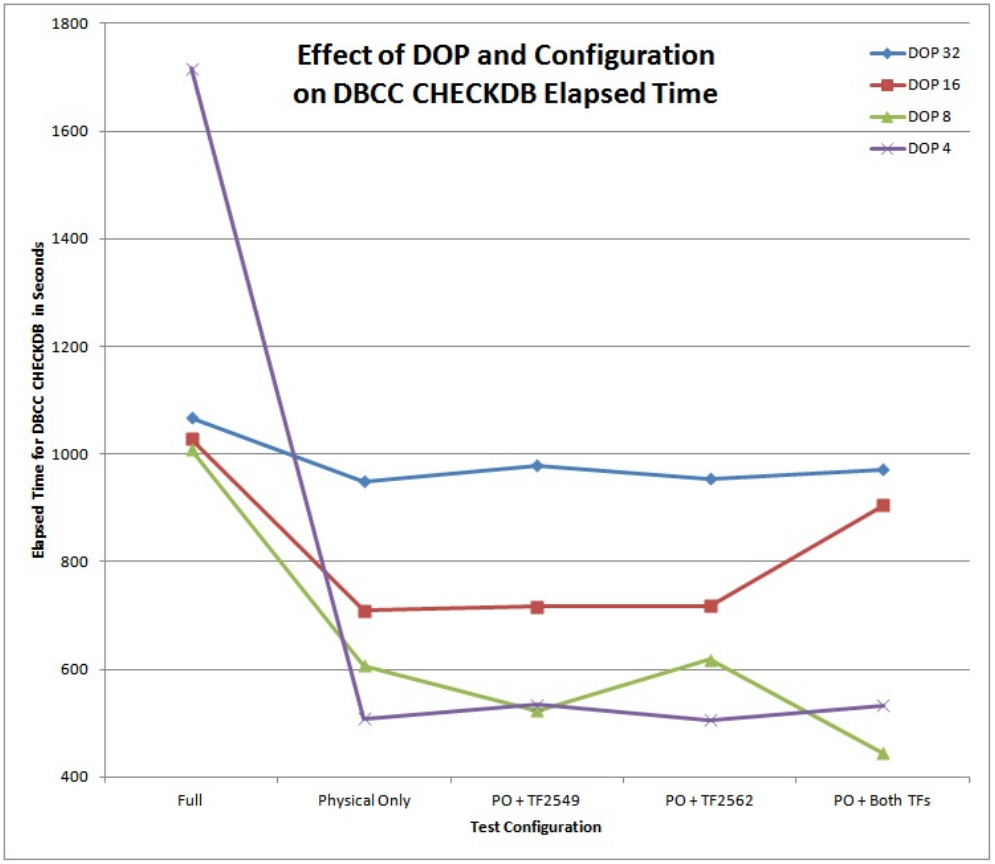
Nonclustered Index Checks (2)

- **The algorithm is basically as follows:**
  - Each index partition has two checksums
    - One generated from its own records, and one from other records
  - For each data record found:
    - For each index, generate the index record that should exist
    - Hash the index record and add the hash value to the correct index partition's "other" checksum
  - For each index record found:
    - Hash the index record and add the hash value to the index partition's "own" checksum
  - At the end of the batch, each index partition's "own" and "other" checksums should match
  - If not, a "deep dive" process is done for that index partition
    - Two left-anti-semi joins are performed by the Query Processor to find the missing record(s), using query syntax only available to DBCC

Making DBCC CHECKDB Go Faster (1)

- **Recent work has been done to improve the performance of DBCC CHECKDB by:**
  - Allowing it to check all tables as a single batch
  - Slightly changing the I/O pattern it generates
  - Reducing contention on the DBCC_MULTIOBJECT_SCANNER latch
- **KB article describing these is at http://support.microsoft.com/kb/2634571**
- **Ported to SQL Server 2008, 2008 R2, 2012**
  - Uses documented trace flags 2549 and 2562
  - Check the KB article to see which trace flags are required in which versions
- **Be aware these changes may push the I/O subsystem much harder, and you may not see any changes at all**
- **Bob Ward's blog post is at http://bit.ly/yAFY7W**

Making DBCC CHECKDB Go Faster (2)



- Source: my blog post at http://bit.ly/10rgH70
  (those are zeroes, not Os)

- Consider reducing DOP to lower run time when using the option WITH PHYSICAL_ONLY
- On systems with high numbers of physical cores (32+), consider reducing DOP to 4 or 8 to lower run time even without using PHYSICAL_ONLY
- Resource Governor can help with this

Making DBCC CHECKDB Go Faster (3)

- **Consider reducing the memory available to DBCC CHECKDB**
- **The Query Processor by default will grant DBCC CHECKDB a very large query execution memory grant, potentially causing buffer pool shrinkage**
  - Requested grant is many terabytes because of unknown cardinality
  - On a system with 100 GB of memory, grant may be as high as 10 GB!
- **Performance testing by Jonathan Kehayias has shown that limiting the amount of memory for DBCC CHECKDB to 1 GB can lead to 5-10% performance increase, and very limited buffer pool pressure**
- **Use Resource Governor to do this**
- **Jonathan's blog post at http://bit.ly/YBb7iH**

Making DBCC CHECKDB Go Faster (4)

- **Indexes on computed columns drastically slow down DBCC CHECKDB**
  - Also applies to DBCC CHECKTABLE and DBCC CHECKFILEGROUP
- **I discovered this problem during benchmarking tests for the degree-of-parallelism data on one of the previous slides**
- **Any time a table has an index on a computed column, all threads bottleneck on the DBCC_OBJECT_METADATA latch**
  - This controls access to an Expression Evaluator construct in the Query Processor that evaluates computed column expressions
  - The latch can only be acquired in EX mode, hence the bottleneck
- **This can lead to 20x performance decrease**
- **Solution is to disable such indexes while DBCC CHECKDB is running**
- **My blog post on this is at http://bit.ly/YAzY6h**

Summary

- **Knowing how DBCC CHECKDB works can give some insight into its performance characteristics**
- **There are several tweaks you can make around memory, DOP, indexes, options and trace flags to reduce DBCC CHECKDB's run time**

- **In the next module, we'll discuss:**
  - Useful undocumented DBCC commands