

# Temp DB - Resolve tempdb related errors and exceptions

Last updated by | Vitor Tomaz | Feb 24, 2023 at 3:27 AM PST

## Contents

- [Resolve tempdb related errors and exceptions](#)
  - [Emergency mitigation - initiate a database failover to clear...](#)
  - [Short-term mitigation - scale database or pool to a larger ...](#)
  - [Features and operations that require tempdb space](#)
  - [Monitoring and analyzing tempdb space usage](#)
    - [Size of tempdb, free and used space, user objects vs. inter...](#)
    - [Space consumed in each session and each running task pe...](#)
    - [Sessions with open transactions in tempdb](#)
  - [User objects are allocating the tempdb space](#)
  - [Internal objects are allocating the tempdb space](#)
    - [Warning: "Operator used tempdb to spill data during exec...](#)
    - [Hash Match Join, Hash Aggregate operators](#)
    - [Table Spool, Index Spool](#)
  - [Running out of transaction log space in tempdb](#)
  - [BULK INSERT command using tempdb](#)
  - [CREATE or ALTER INDEX in Hyperscale database](#)
  - [Resources](#)

This article is a copy of the Selfhelp - Common Solutions article which the customer will see when requesting help through the Azure portal.

## Resolve tempdb related errors and exceptions

The tempdb system database is a global resource available to users who are connected to Azure SQL Database or any instance of SQL Server. It holds temporary user objects that are explicitly created by a user or application, and internal objects that are created by the SQL Server database engine itself. The most common tempdb issue is running out of space, either regarding tempdb's overall size quota or the transaction log.

The available tempdb space in Azure SQL Database depends on two factors: the service tier (pricing tier) that the database is configured with, and the type of workload that is executed against the database. These are also the main factors to control if you are running out of tempdb space.

Scan the following sections for information and steps that will help resolve your issue.

## Emergency mitigation - initiate a database failover to clear tempdb



Consider this step under the following conditions:

- you are running out of space in tempdb
- the issue is occurring for the first time
- your production is blocked
- you don't have sufficient time for troubleshooting



If these conditions do *not* apply, then do *not* initiate a failover. Rather continue troubleshooting with the next section "Short-term mitigation - scale database or pool to a larger service tier" instead.

After initiating a failover, the database will be restarted in a similar way as in a planned maintenance event. This clears the current tempdb and a new, empty tempdb will be created, thus freeing the space from the earlier issue. It will also cause a brief unavailability of the database, like in a planned maintenance event. Note that for elastic pools, all databases in the elastic pool will be affected by the restart.

To initiate a failover for a **single Azure SQL database**, you have the following options:

- See the [Invoke-AzSqlDatabaseFailover](#)  PowerShell command.
- See the REST API command in the article [Databases - Failover](#) . You can run the failover command directly from the article page by selecting the **Try it** button on the first syntax option. If successful, it will trigger an asynchronous operation to failover the database in the background.

If the database is in an **Elastic Pool**, the commands above will failover only the given database without affecting the other databases in the same elastic pool, and it will not resolve the tempdb issue. To initiate a failover for an elastic pool, use the following options instead:

- See the [Invoke-AzSqlElasticPoolFailover](#)  PowerShell command.
- See the REST API command in the article [Elastic Pools - Failover](#) . You can run the failover command directly from the article page by selecting the **Try it** button on the first syntax option. If successful, it will trigger an asynchronous operation to failover the elastic pool in the background.

## Short-term mitigation - scale database or pool to a larger service tier

If you are running out of space in tempdb, if your production is blocked, and if you don't have sufficient time for troubleshooting, the quickest mitigation step is to scale the affected database or elastic pool up to a larger service tier. This can help avoiding the issue and will give you time for analyzing what is consuming the space in tempdb.

Scaling up the database will help if you encounter one of the following error messages:

The database 'tempdb' has reached its size quota. Partition or delete data, drop indexes, or consult the documentation for possible resolutions. (Microsoft SQL Server, Error: 40544)

Msg 40197, Level: 20, State: 1,

The service has encountered an error processing your request. Please try again. Error code 1104.

Error 1104: TEMPDB ran out of space during spilling. Create space by dropping objects and/or rewrite the query to consume fewer rows. If the issue still persists, consider upgrading to a higher service level objective.





Msg 9002, Level 17, State 4, Line 1 The transaction log for database 'tempdb' is full due to 'ACTIVE\_TRANSACTION' and the holdup lsn is ...

The scaling has two aspects towards mitigation:

- It provides more space in tempdb due to the higher service tier.
- It clears the current tempdb by restarting the SQL service. A new, empty tempdb will be created upon restart, thus clearing the space from the earlier issue.

The maximum space for tempdb depends on the selected service tier of the database. When changing the service tier, you can either scale to a higher level within the same purchasing model or consider moving to another purchasing model. For databases in the DTU model, the range is between 13.9 GB and 384 GB (Basic to Standard S12), and is 166.7 GB in Premium. In the vCore model, the range is between 32 GB and 4096 GB, depending on the hardware platform and if using General Purpose vs. Business Critical.

The exact limits for each service tier are documented in the following set of articles:

- DTU model: [Single databases](#)  and [Elastic pools](#) 
- vCore model: [Single databases](#)  and [Elastic pools](#) 

## Features and operations that require tempdb space

The tempdb space is consumed by temporary user objects that are explicitly created by a user or application, and by internal SQL Server objects that are created by the database engine itself. See the following table for an overview:


Temporary user objects	Internal objects
Global or local temporary tables and indexes	Query plan operators: Sort, Hash Match Join, Hash Aggregate, Table Spool, Index Spool
Temporary stored procedures	Create or rebuild index with SORT_IN_TEMPDB = ON
Table variables	Intermediate sort results for large GROUP BY, ORDER BY, or UNION queries
Tables returned in table-valued functions	Intermediate results for large object (LOB) operations
Cursors (e.g. <code>DECLARE c CURSOR FOR SELECT col1 FROM dbo.test</code> )	LOB variables like VARCHAR(max) or XML data types

**Regarding global temporary tables and stored procedures:** These are shared for all user sessions in the same Azure SQL database, but user sessions from other Azure SQL databases cannot access them. This makes troubleshooting difficult for elastic pools where several databases are sharing the same tempdb, and any of the pool's databases could be contributing to the tempdb issue.

**Regarding table variables:** These can be a great feature as long as they hold only a few rows, e.g. about 100 rows or less. They become a bad choice if they contain a lot more than that, like thousands or more rows. There are also no column statistics on table variables, which may lead to wrong query optimizer decisions and bad performance in addition to the tempdb I/O and space contention. If you are using table variables, make sure to keep them small.

**Regarding cursors:** These are a potential performance bottleneck and contribute to tempdb contention if they return larger resultsets. If your applications are still using cursors, then work on replacing them with set-based operations as much as possible.

**Regarding query plan operations** like Sort, Hash Match Join, Hash Aggregate, Table Spool, Index Spool, or certain Merge Join operations: These often occur because of missing indexes, high index fragmentation, or outdated statistics. They are the side effect of non-optimal query execution plans and need to be treated as a performance issue. See the corresponding section further below for additional guidance.

**Regarding create or rebuild index with SORT\_IN\_TEMPDB = ON:** If the index operation using SORT\_IN\_TEMPDB runs out of tempdb space, you need either increase the tempdb size by scaling to a larger service tier, or set SORT\_IN\_TEMPDB = OFF. For a detailed discussion and further guidance, see [Disk Space Requirements](#) .

## Monitoring and analyzing tempdb space usage

To troubleshoot tempdb space issues, it is important to get an overview about how much space is available and what resources are consuming the space. The SQL queries in this section will help with gathering this information. Run the following queries in the Azure SQL database for which you have seen or are suspecting tempdb-related errors or issues.

### Size of tempdb, free and used space, user objects vs. internal objects


These queries will show you the allocated space on the tempdb file level, and how much space is used by user objects and SQL-internal objects.

```
SELECT [Source] = 'database_files',
       [TEMPDB_max_size_MB] = SUM(max_size) * 8 / 1027.0,
       [TEMPDB_current_size_MB] = SUM(size) * 8 / 1027.0,
       [FileCount] = COUNT(FILE_ID)
FROM tempdb.sys.database_files
WHERE type = 0 --ROWS

SELECT [Source] = 'dm_db_file_space_usage',
       [free_space_MB] = SUM(U.unallocated_extent_page_count) * 8 / 1024.0,
       [used_space_MB] = SUM(U.internal_object_reserved_page_count + U.user_object_reserved_page_count + U.version_store_reserved_page_count) * 8 / 1024.0,
       [internal_object_space_MB] = SUM(U.internal_object_reserved_page_count) * 8 / 1024.0,
       [user_object_space_MB] = SUM(U.user_object_reserved_page_count) * 8 / 1024.0,
       [version_store_space_MB] = SUM(U.version_store_reserved_page_count) * 8 / 1024.0
FROM tempdb.sys.dm_db_file_space_usage U
```

### Space consumed in each session and each running task per session

These queries will show you the space consumed by each session, and by each task per session in case the session runs several tasks in parallel. See the size results related to user objects and internal objects; these will indicate if the cause is rather on the user and application side or with the query execution.

The tasks query will also provide you with details regarding the active SQL query text and its actual execution plan, the corresponding client application, and some additional details that will help identifying the source of the query. If you run the tasks query in [SQL Server Management Studio \(SSMS\)](#) , you can select the actual query execution plan directly from the resultset to investigate its details in SSMS.

**Note** that to see the actual execution plan, two database-scoped SET options need to be enabled; these options only apply to queries that have started and completed after the SET options had been enabled. The column "[query\_plan\_current]" will show you the in-flight, transient execution details while the underlying query continues to run; the column "[query\_plan\_previous]" will show you the actual details of an execution that has already completed earlier.

```

/** space consumed in each session */
SELECT [Source] = 'dm_db_session_space_usage',
       [session_id] = SU.session_id,
       [login_name] = MAX(S.login_name),
       [database_id] = MAX(S.database_id),
       [database_name] = MAX(D.name),
       [program_name] = MAX(S.program_name),
       [host_name] = MAX(S.host_name),
       [internal_objects_alloc_page_count_MB] = SUM(internal_objects_alloc_page_count) * 8 / 1024.0,
       [user_objects_alloc_page_count_MB] = SUM(user_objects_alloc_page_count) * 8 / 1024.0
FROM tempdb.sys.dm_db_session_space_usage SU
INNER JOIN sys.dm_exec_sessions S ON SU.session_id = S.session_id
LEFT JOIN sys.databases D ON S.database_id = D.database_id
WHERE internal_objects_alloc_page_count + user_objects_alloc_page_count > 0
GROUP BY SU.session_id
ORDER BY [user_objects_alloc_page_count_MB] desc, SU.session_id;

/** sessions and tasks - details about running tasks in each session */

-- enable lightweight query profiling and statistics collection to get the actual execution plan
-- NOTE that enabling this feature has a 1.5~2 % CPU overhead; LIGHTWEIGHT_QUERY_PROFILING is ON by default
ALTER DATABASE SCOPED CONFIGURATION SET LIGHTWEIGHT_QUERY_PROFILING = ON;
ALTER DATABASE SCOPED CONFIGURATION SET LAST_QUERY_PLAN_STATS = ON;

SELECT [Source] = 'dm_db_task_space_usage',
       [session_id] = SU.session_id, [request_id] = SU.request_id,
       [internal_objects_alloc_page_count_MB] = SU.internal_objects_alloc_page_count * 8 / 1024.0,
       [user_objects_alloc_page_count_MB] = SU.user_objects_alloc_page_count * 8 / 1024.0,
       [database_id] = S.database_id,
       [database_name] = D.name,
       [query_text] = SUBSTRING(T.text, R.statement_start_offset/2 + 1, (CASE WHEN R.statement_end_offset = -1 TH
       [query_plan_current] = P1.query_plan,
       [query_plan_previous] = P2.query_plan,
       [query_plan_handle] = P1.plan_handle,
       [open_transactions] = S.open_transaction_count,
       [login_name] = S.login_name,
       [program_name] = S.program_name,
       [host_name] = S.host_name,
       [start_time] = R.start_time,
       [status] = R.status
FROM sys.dm_db_task_space_usage SU
INNER JOIN sys.dm_exec_requests R ON (SU.session_id = R.session_id AND SU.request_id = R.request_id)
INNER JOIN sys.dm_exec_sessions S ON R.session_id = S.session_id
LEFT JOIN sys.databases D ON S.database_id = D.database_id
CROSS APPLY sys.dm_exec_sql_text(R.sql_handle) T
OUTER APPLY sys.dm_exec_query_statistics_xml(SU.session_id) AS P1
OUTER APPLY sys.dm_exec_query_plan_stats(P1.plan_handle) AS P2
WHERE SU.internal_objects_alloc_page_count + SU.user_objects_alloc_page_count > 0
ORDER BY [user_objects_alloc_page_count_MB] desc, session_id, R.request_id;

```

## Sessions with open transactions in tempdb

This query will return all sessions that have open transactions in tempdb, with their transaction details.

```
-- Sessions with open transactions in tempdb
SELECT [Source] = 'database_transactions',
       [session_id] = ST.session_id,
       [transaction_id] = ST.transaction_id,
       [login_name] = S.login_name,
       [database_id] = S.database_id,
       [program_name] = S.program_name,
       [host_name] = S.host_name,
       [database_id] = DT.database_id,
       [database_name] = CASE
           WHEN D.name IS NULL AND DT.database_id = 2 THEN 'TEMPDB'
           ELSE D.name
       END,
       [log_reuse_wait_desc] = D.log_reuse_wait_desc,
       [database_transaction_log_used_Kb] = CONVERT(numeric(18,2), DT.database_transaction_log_bytes_used / 1024),
       [database_transaction_begin_time] = DT.database_transaction_begin_time,
       [transaction_type_desc] = CASE DT.database_transaction_type
           WHEN 1 THEN 'Read/write transaction'
           WHEN 2 THEN 'Read-only transaction'
           WHEN 3 THEN 'System transaction'
           WHEN 4 THEN 'Distributed transaction'
       END,
       [transaction_state_desc] = CASE DT.database_transaction_state
           WHEN 1 THEN 'The transaction has not been initialized.'
           WHEN 2 THEN 'The transaction is active'
           WHEN 3 THEN 'The transaction has been initialized but has not generated any log records.'
           WHEN 4 THEN 'The transaction has generated log records.'
           WHEN 5 THEN 'The transaction has been prepared.'
           WHEN 10 THEN 'The transaction has been committed.'
           WHEN 11 THEN 'The transaction has been rolled back.'
           WHEN 12 THEN 'The transaction is being committed. (The log record is being generated, but has not been
       END,
       [active_transaction_type_desc] = CASE AT.transaction_type
           WHEN 1 THEN 'Read/write transaction'
           WHEN 2 THEN 'Read-only transaction'
           WHEN 3 THEN 'System transaction'
           WHEN 4 THEN 'Distributed transaction'
       END,
       [active_transaction_state_desc] = CASE AT.transaction_state
           WHEN 0 THEN 'The transaction has not been completely initialized yet.'
           WHEN 1 THEN 'The transaction has been initialized but has not started.'
           WHEN 2 THEN 'The transaction is active'
           WHEN 3 THEN 'The transaction has ended. This is used for read-only transactions.'
           WHEN 4 THEN 'The commit process has been initiated on the distributed transaction.'
           WHEN 5 THEN 'The transaction is in a prepared state and waiting resolution.'
           WHEN 6 THEN 'The transaction has been committed.'
           WHEN 7 THEN 'The transaction is being rolled back.'
           WHEN 8 THEN 'The transaction has been rolled back.'
       END
FROM sys.dm_tran_database_transactions DT
INNER JOIN sys.dm_tran_session_transactions ST ON DT.transaction_id = ST.transaction_id
INNER JOIN sys.dm_tran_active_transactions AT ON DT.transaction_id = AT.transaction_id
INNER JOIN sys.dm_exec_sessions S ON ST.session_id = S.session_id
LEFT JOIN sys.databases D ON DT.database_id = D.database_id
WHERE DT.database_id = 2 -- tempdb
ORDER BY ST.session_id, DT.database_id;
```


## User objects are allocating the tempdb space

Identify the affected sessions, their SQL statements and application name through the system queries in the previous section. Then investigate the applications and SQL statements if they are using any of the user objects listed in section "Features and operations that require tempdb space" above.

The goal is to reduce the number of user objects and/or the data they hold so that they fit within the allocated tempdb space. If this cannot be done, then the second-best mitigation is to scale the database to a larger service tier. See the "Short-term mitigation" section above for further guidance on this.

## Internal objects are allocating the tempdb space

If the tempdb is used by internal objects, then the key is to identify the SQL queries that are related to these internal objects. Use the sessions and tasks query from the "Monitoring and analyzing" section above. It will show the session details and the SQL texts together with their individual tempdb space allocations.



Once the corresponding SQL queries are isolated, you need to troubleshoot their **actual execution plan**. The sessions and tasks query from above is providing two different actual execution plans: "[query\_plan\_current]" shows the transient, incomplete statistics of the currently-executing query; "[query\_plan\_previous]" shows the execution statistics of a previous successful execution. Continue your investigation with the plan from "[query\_plan\_previous]", if available. If you have run the query from a query window in [SQL Server Management Studio \(SSMS\)](#) , you can select the execution plan directly from the `query_plan_previous` column for further analysis.

The goal is to identify the query plan operators "Sort", "Hash Match Join", "Hash Aggregate", "Table spool", and "Index spool", as well as any warnings related to them. Warnings will be indicated as a "!" inside a yellow triangle on these operators. Hover the mouse over the operators to see further details. Then investigate if the query can be changed to avoid these operators or remove the warnings.

### Warning: "Operator used tempdb to spill data during execution with spill level xx"

This occurs if the query engine has estimated the number of rows too low and hasn't allocated sufficient memory. To execute the query, it needs to spill the overflow intermediate results to tempdb. Check the arrow that is pointing into the Sort operator: if the "Actual Number of Rows" is larger than the "Estimated Number of Rows", then this is the cause for the tempdb usage.


The mitigation is to get a correct value for the estimated number of rows and the required memory:

- If indexes and statistics already exist, then run `UPDATE STATISTICS ... WITH FULLSCAN` on the related tables.
- Review the query and its tables to make sure that the relevant columns have indexes and/or statistics.
- If possible, create [covering indexes](#)  to avoid additional join and sort operations in the query plan.
- Make sure that the database compatibility level is set to 150 (SQL Server 2019) or higher, and that the [row mode memory grant feedback](#)  feature is enabled. This feature allows for adjusting memory grants on repeated executions of a query:



```
ALTER DATABASE [database name] SET COMPATIBILITY_LEVEL = 160
```

```
ALTER DATABASE SCOPED CONFIGURATION SET ROW_MODE_MEMORY_GRANT_FEEDBACK = ON
```

### Hash Match Join, Hash Aggregate operators

Hash match joins can efficiently process large, unsorted, nonindexed inputs. They are useful for intermediate results in complex queries, because these are often not suitably sorted for the next operation in the query plan. Also the size estimates of intermediate result may be inaccurate for complex queries, therefore hash match joins can be the most efficient way to process specific query types. See [Hash joins](#)  for further details. Hash aggregates work in a similar way, but for data aggregation tasks, e.g. in a `GROUP BY` query. Hash aggregates are more sensitive to estimating the required memory incorrectly if the statistics are incorrect.

Smaller hash operator queries can be managed in memory, but larger results will require worktables in tempdb and thus contribute to tempdb space issues. If you want to avoid hash operations, consider the following items:

- Run `UPDATE STATISTICS ... WITH FULLSCAN` on the related tables so that the decision to use a hash operator is correct and not wrongly preferred over another operator.
- Try to minimize the memory footprint by reducing the amount of data that needs to be processed:
  - Reduce the number of selected columns as much as possible, ideally so that the columns can be included in a covering index.
  - Add selective conditions to the Where clause (instead of filtering the results later)
- Make sure that all columns on the Join and Where clauses are properly indexed. This also helps retrieving sorted data, which allows for using other operators than hash operators.
- Review the join conditions and if their complexity can be reduced, especially if a condition is not a straightforward equality match and involves calculations, conversions, concatenations, or similar. Consider adding a calculated column to the table to reduce the join complexity.
- Add a specific join or group hint to the query so that the hash operation will be replaced by a loop or merge operation. This might negatively impact the overall performance though and requires thorough testing. See [Join Hints Examples](#)  and [Group Hints](#)  for further guidance.

It might turn out that the hash operation is the most efficient way to execute the query. In this case, the mitigation is to scale the database to a larger service tier. See the "Short-term mitigation" section above for further guidance on it.

### Table Spool, Index Spool

Spools are internal tables in tempdb, and the query optimizer uses them to avoid re-retrieving data from a table or index repeatedly. Spools are often related to the lack of proper indexes, or to the same data being joined several times into the query, e.g. in nested, recursive queries or in Update queries to maintain the initial data.

The mitigation is not straightforward and requires further analysis of the affected query. The goal is either to reduce the data that goes into the spool operation, or to avoid that data needs to be read repeatedly. This requires a review of the Join operations of the query, including sub-queries, recursions, and Where clauses. It also requires a review of the indexes that are servicing these Join operations.

It might turn out that the spool operation is the most efficient way to execute the query. In this case, the mitigation is to scale the database to a larger service tier. See the "Short-term mitigation" section above for further guidance on it.

### Running out of transaction log space in tempdb

The most common error when running out of transaction log space in tempdb is the following:

```
Msg 9002, Level 17, State 4, Line 1 The transaction log for database 'tempdb' is full due to 'ACTIVE_TRANSACTION' and the holdup lsn is ...
```

"ACTIVE\_TRANSACTION" indicates that the database is running out of virtual log files because of an open transaction. Open transactions keep virtual log files active, as their log records might be required to rollback the transaction. The best way to mitigate this error is to avoid long-running transactions and to design transactions as short-lived as possible.



To find out more about open transactions in tempdb, run the following query in the Azure SQL database for which you have seen or are suspecting the tempdb-related issues.



```
-- Sessions with open transactions in tempdb
SELECT [Source] = 'database_transactions',
       [session_id] = ST.session_id,
       [transaction_id] = ST.transaction_id,
       [login_name] = S.login_name,
       [database_id] = S.database_id,
       [program_name] = S.program_name,
       [host_name] = S.host_name,
       [database_id] = DT.database_id,
       [database_name] = CASE
           WHEN D.name IS NULL AND DT.database_id = 2 THEN 'TEMPDB'
           ELSE D.name
       END,
       [log_reuse_wait_desc] = D.log_reuse_wait_desc,
       [database_transaction_log_used_Kb] = CONVERT(numeric(18,2), DT.database_transaction_log_bytes_used / 1024),
       [database_transaction_begin_time] = DT.database_transaction_begin_time,
       [transaction_type_desc] = CASE DT.database_transaction_type
           WHEN 1 THEN 'Read/write transaction'
           WHEN 2 THEN 'Read-only transaction'
           WHEN 3 THEN 'System transaction'
           WHEN 4 THEN 'Distributed transaction'
       END,
       [transaction_state_desc] = CASE DT.database_transaction_state
           WHEN 1 THEN 'The transaction has not been initialized.'
           WHEN 2 THEN 'The transaction is active'
           WHEN 3 THEN 'The transaction has been initialized but has not generated any log records.'
           WHEN 4 THEN 'The transaction has generated log records.'
           WHEN 5 THEN 'The transaction has been prepared.'
           WHEN 10 THEN 'The transaction has been committed.'
           WHEN 11 THEN 'The transaction has been rolled back.'
           WHEN 12 THEN 'The transaction is being committed. (The log record is being generated, but has not been
       END,
       [active_transaction_type_desc] = CASE AT.transaction_type
           WHEN 1 THEN 'Read/write transaction'
           WHEN 2 THEN 'Read-only transaction'
           WHEN 3 THEN 'System transaction'
           WHEN 4 THEN 'Distributed transaction'
       END,
       [active_transaction_state_desc] = CASE AT.transaction_state
           WHEN 0 THEN 'The transaction has not been completely initialized yet.'
           WHEN 1 THEN 'The transaction has been initialized but has not started.'
           WHEN 2 THEN 'The transaction is active'
           WHEN 3 THEN 'The transaction has ended. This is used for read-only transactions.'
           WHEN 4 THEN 'The commit process has been initiated on the distributed transaction.'
           WHEN 5 THEN 'The transaction is in a prepared state and waiting resolution.'
           WHEN 6 THEN 'The transaction has been committed.'
           WHEN 7 THEN 'The transaction is being rolled back.'
           WHEN 8 THEN 'The transaction has been rolled back.'
       END
FROM sys.dm_tran_database_transactions DT
INNER JOIN sys.dm_tran_session_transactions ST ON DT.transaction_id = ST.transaction_id
INNER JOIN sys.dm_tran_active_transactions AT ON DT.transaction_id = AT.transaction_id
INNER JOIN sys.dm_exec_sessions S ON ST.session_id = S.session_id
LEFT JOIN sys.databases D ON DT.database_id = D.database_id
WHERE DT.database_id = 2 -- tempdb
ORDER BY ST.session_id, DT.database_id;
```

The next troubleshooting step is to relate the session information to the workload behind these transactions. Check if and how the workload can be changed, for example by configuring smaller batch sizes, or by redesigning the process flow to use shorter transactions.

It might turn out though that the workload cannot be sufficiently changed. In this case, the mitigation is to scale the database to a larger service tier. See the "Short-term mitigation" section above for further guidance on it.



## BULK INSERT command using tempdb

The `BULK INSERT` operation might require tempdb space if the the target table has indexes and the inserted data needs to be sorted. If the Bulk Insert runs out of tempdb space, there are two possible mitigation steps:

- Use the [BULK INSERT ... BATCHSIZE](#)  option to configure a smaller batch size. `BATCHSIZE` specifies the number of rows per batch, and each batch is copied to the server as one transaction. By default, all data in the specified data file is inserted in one batch and one transaction, which might exceed the transaction space for large, unsorted source files in either tempdb or the target database.
- Use the [BULK INSERT ... ORDER](#)  option if the data in the source file is sorted according to the clustered index on the target table. If the data file is sorted in a different order, or if there's no clustered index on the table, the `ORDER` clause is ignored. By default, the bulk insert operation assumes the data file is unordered.







Besides potential tempdb restrictions, consider temporarily increasing the performance level of the database prior to the Bulk Insert operation, especially for a large volume of data. A larger service tier improves the I/O performance of the insert and scaling up reduces the overall duration and the risk of timeouts. Scale down again after the Bulk operation was successful.

## CREATE or ALTER INDEX in Hyperscale database

In Hyperscale databases, the option `SORT_IN_TEMPDB` for [CREATE INDEX](#)  and [ALTER INDEX](#)  is always set to "ON", no matter what you specify when executing either command. The default is "OFF" for all other types of Azure SQL Database. This makes Hyperscale prone to running out of tempdb space when indexes are (re-)built for large tables, as the intermediate sort results are always stored in tempdb.

There is one exception to this rule though. `SORT_IN_TEMPDB` is always "ON", unless the `RESUMABLE` option is used. So making the operation resumable, as in `CREATE INDEX ... ON ... WITH (ONLINE = ON, RESUMABLE = ON)`, is implicitly disabling `SORT_IN_TEMPDB` on Hyperscale.

## Resources

- [Azure SQL DB and TEMPDB usage tracking](#) 
- [Monitoring tempdb use](#) 
- [Tempdb Database](#) 
- [Query Profiling Infrastructure](#) 
- [Lightweight query execution statistics profiling infrastructure v3](#) 
- [Using Extended Events to capture an Actual Execution Plan](#) 

How good have you found this content?

