

Indexoptimierung bei SQL Server – Leitfaden für IT-Verantwortliche und DBAs

Der Anruf kommt selten montags. Er kommt freitags gegen 16 Uhr, und am anderen Ende sagt jemand den Satz, den du schon hundertmal gehört hast: „Die Anwendung ist langsam geworden.“ Nicht kaputt. Langsam. Bei einem Kunden aus der Fertigung lief seit vier Jahren jede Nacht ein Wartungsplan, der stur alle Indizes der 3-TB-Datenbank neu aufgebaut hat. Der Job brauchte inzwischen sieben Stunden, produzierte 900 GB Transaktionsprotokoll pro Nacht, drückte die Verfügbarkeitsgruppe regelmäßig in eine Redo-Queue von mehreren Minuten – und das Performanceproblem war trotzdem da. Die Ursache war ein einziger fehlender Index auf einer Statusspalte und eine Statistik, die nie mit vernünftiger Stichprobe aktualisiert wurde. Sieben Stunden Nacharbeit gegen ein Problem, das mit vier Minuten Arbeit erledigt war.

Genau darum geht es bei SQL Server Indexoptimierung: nicht darum, möglichst viel zu tun, sondern das Richtige. Ein Index ist ein Vertrag. Du kaufst Lesegeschwindigkeit und zahlst mit Speicherplatz, Schreiblast, Protokollvolumen und Wartungsaufwand. Dieser Leitfaden zeigt dir, wie die Datenstrukturen tatsächlich aussehen, welche Kennzahlen etwas taugen, wann Rebuild und wann Reorganize richtig ist, wie du das Ganze automatisierst, ohne dir die Hochverfügbarkeit zu zerlegen – und wann die beste Maßnahme darin besteht, die Finger stillzuhalten.

Was du aus diesem Leitfaden mitnimmst

Der Clustered Index bestimmt die physische Sortierung der Tabelle und steckt als Zeiger in jedem einzelnen Non-Clustered Index. Ein schlecht gewählter Clustered Key ist deshalb kein Schönheitsfehler, sondern eine Hypothek auf die gesamte Tabelle.

Seitendichte schlägt Fragmentierung. In den meisten Workloads bringt eine höhere Seitendichte mehr als eine niedrigere Fragmentierungszahl – auf Flash-Speicher ist der Unterschied zwischen sequenziellem und wahlfreiem Zugriff ohnehin klein geworden.

Der größte Teil des gefühlten Erfolgs eines Index-Rebuilds kommt von den nebenbei aktualisierten Statistiken. Ein UPDATE STATISTICS kostet Minuten, ein Rebuild Stunden.

Die Schwellenwerte 5 Prozent und 30 Prozent sind brauchbare Startwerte, keine Naturkonstanten. Microsoft empfiehlt ausdrücklich, den Nutzen im eigenen Workload zu messen statt starren Grenzwerten zu folgen.

Wie SQL Server Indizes tatsächlich aufgebaut sind

Der Vergleich mit dem Buchregister ist abgedroschen, trägt aber weiter: Ohne Register musst du das ganze Buch lesen, mit Register schlägst du nach. In SQL Server ist dieses Register ein Baum – genauer ein B+-Baum – aus 8-KB-Seiten. Oben eine Wurzelseite, darunter ein bis drei Zwischenebenen, unten die Blattebene. Für eine Tabelle mit hundert Millionen Zeilen sind das typischerweise vier bis fünf Ebenen. Vier bis fünf Seitenzugriffe, um eine beliebige Zeile zu finden, statt hunderttausender Seiten im vollständigen Scan. Das ist die ganze Magie, und sie ist erstaunlich robust.

Der Clustered Index ist die Tabelle

Beim Clustered Index ist die Blattebene nicht eine Kopie oder ein Verweis, sondern die Tabelle selbst. Die Datenzeilen liegen in der Sortierreihenfolge des Indexschlüssels. Deshalb kann es pro

Tabelle genau einen Clustered Index geben – eine Menge Zeilen lässt sich nun einmal nur nach einem Kriterium physisch anordnen. Wer das Bild braucht: ein Telefonbuch ist nach Nachname geclustert. Du kannst es nicht gleichzeitig nach Straße sortieren, ohne es neu zu drucken.

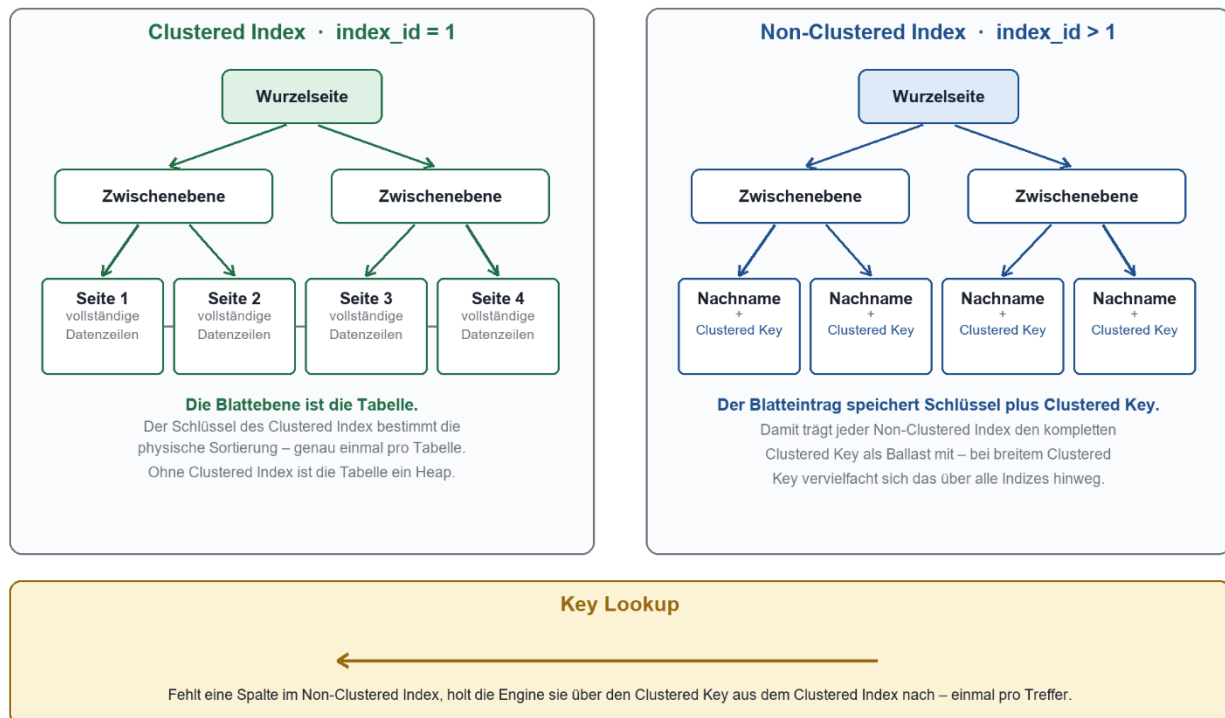
SQL Server legt den Primärschlüssel standardmäßig als Clustered Index an. Das ist eine Voreinstellung, kein Naturgesetz, und sie ist häufig richtig, aber eben nicht immer. Ein guter Clustered Key ist schmal, eindeutig, unveränderlich und möglichst aufsteigend. Schmal, weil er in jedem Non-Clustered Index mitgeschleppt wird. Eindeutig, weil SQL Server sonst intern einen vier Byte großen Uniquifier ergänzt. Unveränderlich, weil ein UPDATE auf den Clustered Key die Zeile physisch verschiebt und sämtliche Non-Clustered Indizes mitzieht. Aufsteigend, weil neue Zeilen dann am Ende angehängt werden, statt sich zwischen bestehende Seiten zu drängeln.

Der Klassiker aus der Praxis ist der zufällige GUID als geclusterter Primärschlüssel. Das Datenmodell kommt aus einem Framework, niemand hat darüber nachgedacht, und die Tabelle wächst fünf Jahre lang. Jeder INSERT landet an einer zufälligen Stelle im Baum, spaltet eine volle Seite in zwei halbvolle und erzeugt dabei vollständig protokollierte Schreibvorgänge. Nach fünf Jahren liegt die Seitendichte bei knapp über 50 Prozent, die Tabelle belegt das Doppelte dessen, was sie müsste, und jede Bereichsabfrage liest doppelt so viele Seiten. Wer GUIDs braucht, nimmt NEWSEQUENTIALID oder clustert auf eine aufsteigende Identity-Spalte und macht den GUID zu einem eindeutigen Non-Clustered Index.

Jeder Non-Clustered Index trägt den Clustered Key mit

Das ist der Punkt, an dem in Design-Diskussionen meist Stille eintritt. Ein Non-Clustered Index speichert in seiner Blattebene den Indexschlüssel plus einen Zeiger auf die Zeile. Dieser Zeiger ist bei einer geclusterten Tabelle nicht etwa eine kompakte Adresse, sondern der vollständige Clustered Key. Steht die Tabelle als Heap da, ist es eine acht Byte große Row-ID. Alles andere ergibt sich daraus: Ein breiter Clustered Key vergrößert nicht einen Index, sondern alle.

Rowstore-Anatomie: Clustered Index und Non-Clustered Index



Skizze 1: Clustered und Non-Clustered Index. Der Blatteintrag des Non-Clustered Index enthält den Clustered Key – über ihn läuft der Key Lookup.

Ein Rechenbeispiel, das ich so bei einem Handelskunden gesehen habe: Der Clustered Key bestand aus vier Spalten und war zusammen rund 40 Byte breit. Auf der Tabelle lagen zwölf Non-Clustered Indizes, die Tabelle hatte 80 Millionen Zeilen. 40 Byte mal zwölf Indizes mal 80 Millionen Zeilen sind rund 38 GB, die nichts anderes tun, als auf Zeilen zu zeigen. Nach der Umstellung auf einen schmalen, aufsteigenden Clustered Key waren es unter 4 GB. Der Rest hat sich in Backupzeit, Wiederherstellungszeit und Buffer-Pool-Platz ausgezahlt, ganz ohne dass eine einzige Abfrage angefasst wurde.

Warnung: Den Clustered Key repariert man nicht nebenbei

Einen Clustered Index zu ändern heißt, ihn neu zu erstellen – über CREATE CLUSTERED INDEX mit DROP_EXISTING oder über Löschen und Neuanlegen. Sobald der Clustered Index angelegt, gelöscht oder mit anderem Schlüssel neu erstellt wird, baut die Engine automatisch sämtliche Non-Clustered Indizes der Tabelle mit neu. Bei einer Terabyte-Tabelle mit fünfzehn Indizes ist das kein Wartungsfenster, sondern ein Projekt mit Testlauf, Platzbedarf für zwei Kopien und Rückfallplan.

Reines Rebuilden eines bestehenden Clustered Index löst diesen Effekt übrigens nicht aus – gleiches gilt, wenn du ihn nur in eine andere Dateigruppe verschiebst oder ein Partitionsschema anwendest. Es ist der Schlüsselwechsel, der teuer ist.

Konsequenz für die Planung: Über den Clustered Key entscheidest du beim Entwurf. Danach entscheidest du nur noch, wie teuer die Korrektur wird.

Heaps: Tabellen ohne Ordnung

Eine Tabelle ohne Clustered Index ist ein Heap. Zeilen liegen dort, wo gerade Platz war. Für reine Ladepuffer im ETL ist das völlig in Ordnung, für alles andere selten. Der unangenehme Effekt heißt Forwarding Record: Wächst eine Zeile durch ein UPDATE über den verbliebenen Platz ihrer Seite hinaus, zieht sie um und hinterlässt einen Verweis. Wer die Zeile sucht, liest die alte Seite und danach die neue. Bei häufigen Updates sammeln sich diese Weiterleitungen an, und sys.dm_db_index_physical_stats zeigt sie als forwarded_record_count. Aufräumen lässt sich das nur mit ALTER TABLE ... REBUILD, und die bessere Lösung ist meistens, endlich einen Clustered Index anzulegen.

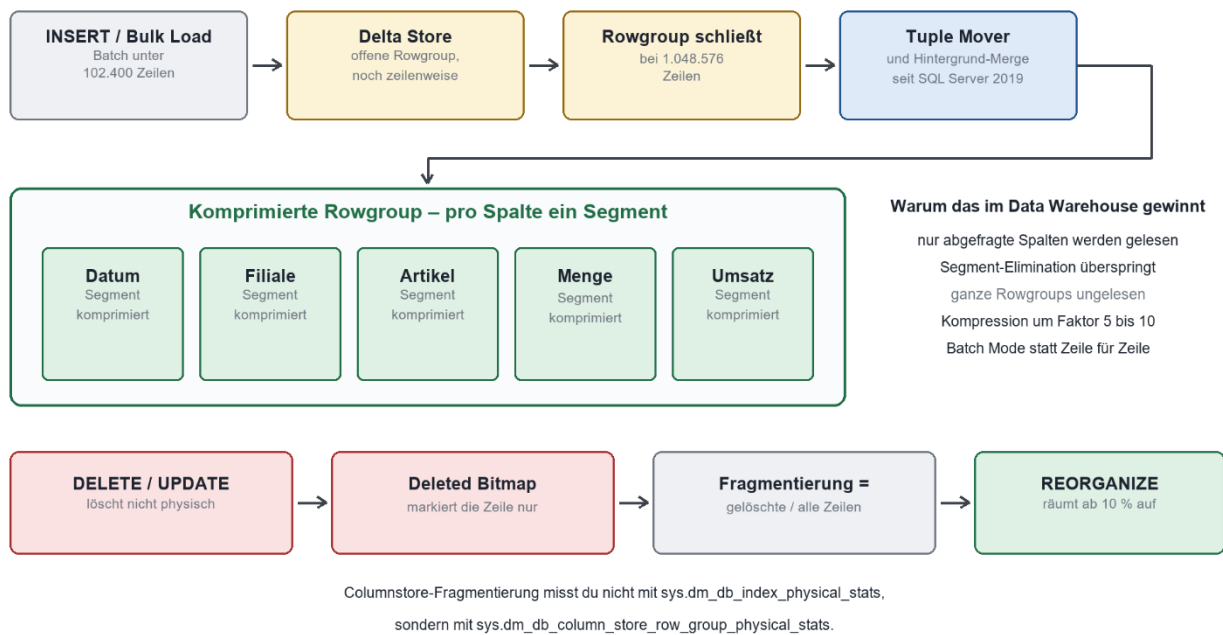
Abdeckende Indizes, INCLUDE und gefilterte Indizes

Ein Key Lookup ist der Preis dafür, dass ein Non-Clustered Index nicht alle benötigten Spalten kennt. Bei zehn Treffern ist das egal, bei zweihunderttausend Treffern kippt der Optimierer irgendwann auf einen vollständigen Scan des Clustered Index um – und dann wundert sich das Team, warum der schöne neue Index „nicht benutzt“ wird. Die INCLUDE-Klausel hängt zusätzliche Spalten an die Blattebene, ohne sie in den Schlüssel aufzunehmen: Sie erhöhen weder die Baumtiefe noch die Sortierlast, machen den Index aber breiter. Gefilterte Indizes mit WHERE-Bedingung sind das unterschätzte Werkzeug für Statusspalten – wenn von 40 Millionen Aufträgen 3000 den Status „offen“ haben, indiziere die 3000 und nicht die 40 Millionen.

Columnstore: Spalten statt Zeilen

Alles bisher Beschriebene ist Rowstore: Eine Seite enthält vollständige Zeilen. Columnstore dreht das um. Die Daten werden in Rowgroups von bis zu 1.048.576 Zeilen zerlegt, und innerhalb einer Rowgroup wird jede Spalte separat als Segment gespeichert und komprimiert. Weil in einer Spalte gleichartige Werte stehen, greifen Wörterbuch- und Lauflängenkompression hervorragend; Faktor fünf bis zehn gegenüber Rowstore ist normal. Eine Abfrage, die drei von sechzig Spalten braucht, liest auch nur drei Segmente. Dazu kommt Segment-Elimination: SQL Server merkt sich Minimum und Maximum je Segment und überspringt ganze Rowgroups ungelesen. Und die Verarbeitung läuft im Batch Mode, also in Blöcken von rund 900 Zeilen pro CPU-Instruktion statt Zeile für Zeile.

Columnstore: vom INSERT zur komprimierten Rowgroup



Skizze 2: Ladeweg und Löschweg eines Columnstore-Index. Kleine Batches landen im Delta Store, gelöschte Zeilen bleiben bis zum REORGANIZE physisch bestehen.

Der Preis steht in derselben Skizze. Ein einzelner INSERT landet nicht in einem Segment, sondern im Delta Store – einer ganz normalen Rowstore-Struktur. Erst ab 102.400 Zeilen pro Batch schreibt die Engine direkt in komprimierte Rowgroups. Ein DELETE löscht nichts, es setzt ein Bit in der Deleted Bitmap; ein UPDATE ist intern ein DELETE plus INSERT. Wer eine OLTP-Tabelle mit einzelnen Buchungen auf Columnstore umstellt, bekommt tausende winzige offene Rowgroups, eine explodierende Deleted Bitmap und Punktabfragen, die deutlich teurer sind als im B-Baum. Seit SQL Server 2019 hilft ein Hintergrundprozess, der offene Delta-Rowgroups nach einer internen Wartezeit komprimiert und stark ausgedünnte Rowgroups zusammenführt – das entschärft die Sache, hebt sie aber nicht auf.

Für gemischte Anforderungen gibt es den nicht geclusterten Columnstore-Index auf einer Rowstore-Tabelle: Das OLTP-Geschäft läuft weiter über den B-Baum, die Auswertungen über den Columnstore. Seit SQL Server 2022 zieht die Segment-Elimination auch bei Zeichenketten, Binärdaten, GUIDs und datetimeoffset mit höherer Genauigkeit, und Rowgroups lassen sich sogar für Präfixsuchen der Form LIKE 'ABC%' überspringen. Wer die Sortierung im Griff haben will, legt einen geordneten geclusterten Columnstore-Index an – dann überlappen die Segmente in der führenden Spalte kaum noch, und die Elimination greift deutlich besser.

Indextyp	Anzahl je Tabelle	Stärke	Preis
Clustered (Rowstore)	genau einer	Bereichsabfragen, Sortierung, kompakter Zeilenzugriff	bestimmt die physische Ordnung, Schlüsselwechsel sehr teuer
Non-Clustered (Rowstore)	bis 999	gezielte Suchen, Joins, abdeckende Abfragen	Speicher, Schreiblast, trägt den Clustered Key mit
Geclusterter Columnstore	genau einer (statt Rowstore)	Faktentabellen, Aggregate, Kompression	ungeeignet für Einzelsatz-OLTP
Nicht geclusterter Columnstore	einer	Auswertungen auf laufenden OLTP-Tabellen	zusätzlicher Schreibaufwand, Delete-Puffer

Gefilterter Index	im Rahmen der 999	kleine Teilmengen wie Status oder Mandant	greift nur, wenn das Prädikat passt
XML-Index	primär plus sekundäre	Suche in XML-Knoten und Attributen	sehr speicherhungrig, teure Pflege
Volltextindex	einer je Tabelle	Wort- und Phrasensuche in Textspalten	eigene Pflegelogik, eigener Katalog
Räumlicher Index	mehrere	Geodaten, Umkreis- und Schnittabfragen	nur für die Geometrie-Datentypen
Hash-Index (In-Memory)	mehrere	Punktabfragen auf speicheroptimierten Tabellen	keine Bereichsabfragen, Bucket-Anzahl muss passen

Zahlen, die du für Columnstore im Kopf haben solltest

Eine Rowgroup fasst maximal 1.048.576 Zeilen. Ladebatches ab 102.400 Zeilen gehen direkt in komprimierte Rowgroups, kleinere landen zunächst im Delta Store.

ALTER INDEX ... REORGANIZE räumt eine komprimierte Rowgroup auf, sobald mindestens zehn Prozent ihrer Zeilen als gelöscht markiert sind, und führt kleine Rowgroups bis zur Obergrenze zusammen.

Seit SQL Server 2016 erledigt REORGANIZE im Wesentlichen das, wofür man früher einen Rebuild brauchte – und zwar online. Seit SQL Server 2019 lässt sich auch ein geclusterter Columnstore-Index mit ONLINE = ON neu aufbauen.

Der Preis jedes Index: Speicher, Schreiblast, Seitenteilungen

Indizes werden gerne als kostenlose Beschleunigung verkauft. Sie sind es nicht. Jeder Index belegt Platz auf der Platte, im Buffer Pool, in jedem Backup, in jeder Wiederherstellung und – bei Verfügbarkeitsgruppen – in jedem Byte, das über die Leitung zum Sekundärknoten geht. Und jeder Index will bei Datenänderungen gepflegt werden.

Was eine Datenänderung wirklich anfasst

Ein INSERT schreibt in den Clustered Index und in jeden einzelnen Non-Clustered Index. Ein DELETE ebenso. Ein UPDATE ist gnädiger: Es fasst nur die Indizes an, die eine der geänderten Spalten enthalten – es sei denn, die geänderte Spalte gehört zum Clustered Key, dann wandert die Zeile und alles zieht mit. Zwölf Indizes auf einer Tabelle bedeuten also im schlechtesten Fall dreizehn Strukturen, die bei jeder Zeile gepflegt werden, dreizehnmal Sperren, dreizehnmal Protokolleinträge. Das ist der Grund, warum Ladeprozesse regelmäßig daran scheitern, dass jemand „vorsichtshalber“ noch ein paar Indizes angelegt hat.

Seitenteilungen und der Fill Factor

Wenn eine neue Zeile in eine volle Seite muss, teilt SQL Server die Seite: Er legt eine neue an, verschiebt etwa die Hälfte der Einträge dorthin und passt die Verkettung an. Das kostet CPU, es kostet Protokollvolumen, und es hinterlässt zwei halbvolle Seiten – Fragmentierung und niedrige Seitendichte in einem Vorgang. Genau deshalb ist ein aufsteigender Clustered Key so wertvoll: Am Ende des Baums wird nicht geteilt, sondern angehängt.

Der Fill Factor lässt bewusst Luft auf jeder Seite, damit Einfügungen nicht sofort teilen. Er ist aber kein Allheilmittel, sondern ein Tausch: Du bezahlst dauerhaft mit niedrigerer Seitendichte für gelegentlich vermiedene Teilungen. Die aktuelle Empfehlung von Microsoft ist eindeutig – lass den Fill Factor bei 100 beziehungsweise 0, außer bei Indizes, die nachweislich unter vielen Seitenteilungen leiden, etwa mit nicht sequenziellen GUIDs in der führenden Spalte. Dann liegt der

sinnvoller Bereich zwischen 70 und 95, und du prüfst hinterher nach, ob es etwas gebracht hat. Wer pauschal 80 über die ganze Datenbank legt, verschenkt zwanzig Prozent Speicher, zwanzig Prozent Buffer Pool und zwanzig Prozent Lesegeschwindigkeit.

Über-Indexierung erkennen

Die Kennzahl dafür heißt `sys.dm_db_index_usage_stats`. Sie zählt je Index, wie oft er über Seeks, Scans und Lookups gelesen und wie oft er durch Datenänderungen gepflegt wurde. Ein Index mit 4 Lesezugriffen und 18 Millionen Pflegevorgängen ist kein Index, sondern eine Steuer. Zwei Fallstricke gehören dazu: Die Zähler beginnen bei jedem Dienstneustart und bei jedem Failover von vorn, und Indizes, die seit dem Start gar nicht angefasst wurden, tauchen in der Sicht überhaupt nicht auf. Wer daraus Entscheidungen ableiten will, schreibt die Werte regelmäßig in eine eigene Historientabelle und schaut über mindestens einen vollständigen Geschäftszyklus – Monatsabschluss und Jahresabschluss inklusive.

Bei einem Kunden aus der Logistik lagen auf der Tabelle der Auftragspositionen 23 Indizes. Neun davon waren über acht Wochen kein einziges Mal gelesen worden, vier waren Präfix-Duplikate voneinander – der eine auf (KundeID), der nächste auf (KundeID, Datum), der dritte auf (KundeID, Datum, Status). Der erste und zweite waren überflüssig, weil der dritte sie mit abdeckt. Nach dem Aufräumen lief der nächtliche Import 40 Prozent schneller, und keine einzige Abfrage wurde langsamer.

Symptom	Wo du nachsiehst	Was es bedeutet	Was du tust
Index wird nie gelesen	<code>sys.dm_db_index_usage_stats</code>	<code>user_seeks</code> , <code>user_scans</code> und <code>user_lookups</code> bleiben null	erst deaktivieren, nach einem Zyklus löschen
Index wird nur gepflegt	<code>sys.dm_db_index_usage_stats</code>	<code>user_updates</code> hoch, Lesezugriffe minimal	Nutzen gegen Schreiblast abwägen, meist entfernen
Doppelte Abdeckung	<code>sys.indexes</code> und <code>sys.index_columns</code>	identische oder Präfix-gleiche Spalten	den schmaleren Index auflösen, INCLUDE zusammenführen
Viele Seitenteilungen	<code>sys.dm_db_index_operational_stats</code>	hohe <code>leaf_allocation_count</code> bei nicht sequenziellem Schlüssel	Schlüsselwahl prüfen, notfalls Fill Factor senken
Ständige Key Lookups	Ausführungsplan, Query Store	Nested Loops mit Key Lookup auf breiten Ergebnismengen	fehlende Spalten per INCLUDE ergänzen
Sperrkonflikte auf einem Index	<code>sys.dm_db_index_operational_stats</code>	hohe <code>row_lock_wait_in_ms</code> und <code>page_lock_wait_in_ms</code>	Zugriffsmuster prüfen, Index gegebenenfalls schmaler schneiden

Tipp: Erst deaktivieren, dann löschen

Ein Index, den du löschst, ist weg – samt Definition. Ein Index, den du mit `ALTER INDEX ... DISABLE` deaktivierst, behält seine Definition im Katalog, verbraucht aber keinen Speicher mehr und wird bei Datenänderungen nicht gepflegt. Stellt sich nach zwei Wochen heraus, dass der Quartalsabschluss ihn doch braucht, holst du ihn mit `ALTER INDEX ... REBUILD` zurück, statt die Definition aus einem alten Skript zu rekonstruieren.

Achtung bei der einen Ausnahme: Deaktivierst du den Clustered Index, ist die gesamte Tabelle nicht mehr abfragbar. Das ist kein Fehler, sondern Absicht – aber es überrascht zuverlässig jeden, der es einmal in der Produktion ausprobiert.

Missing-Index-Hinweise sind Hinweise, keine Befehle

Die Sichten `sys.dm_db_missing_index_details`, `sys.dm_db_missing_index_groups` und `sys.dm_db_missing_index_group_stats` sammeln, welche Indizes der Optimierer beim Erstellen eines Plans gerne gehabt hätte. Das ist wertvoll und gefährlich zugleich. Wertvoll, weil die Daten aus echter Last stammen. Gefährlich, weil der Optimierer die Vorschläge nie gegeneinander abwägt, keine sinnvolle Spaltenreihenfolge liefert, bestehende Indizes ignoriert und die Schreibkosten überhaupt nicht kennt. Wer die Vorschläge eins zu eins umsetzt, hat nach einem halben Jahr die 23 Indizes aus dem Beispiel oben.

Dazu kommen harte Grenzen: Die Daten sind nicht persistent. Ein Neustart, ein Failover oder das Offline-Setzen der Datenbank löscht sie, und eine Schemaänderung an einer Tabelle entfernt alle Hinweise zu dieser Tabelle. Die Detailsicht liefert zudem höchstens 600 Zeilen. Wer die Hinweise ernsthaft auswerten will, schreibt sie regelmäßig weg oder nutzt den Query Store, der die Pläne samt Hinweis dauerhaft speichert.

Indexdesign ist an dieser Stelle keine Wartungsaufgabe mehr, sondern eine Architekturentscheidung: Welche Zugriffsmuster muss das System tragen, welche Antwortzeiten sind zugesagt, wie viel Schreiblast verträgt das Ladefenster? Wenn du diese Fragen mit vertretbarem Aufwand nicht intern beantworten kannst, ist ein Blick von außen billiger als die dritte Runde Symptombehandlung – dafür gibt es die [SQL Server Beratung](#).

Fragmentierung und Seitendichte richtig messen

Jetzt zur eigentlichen Wartung. Und zu einer Unterscheidung, die in vielen Betrieben seit zwanzig Jahren durcheinandergeht: Fragmentierung und Seitendichte sind zwei verschiedene Dinge mit zwei verschiedenen Auswirkungen.

`sys.dm_db_index_physical_stats` richtig aufrufen

Die Funktion ist das Standardwerkzeug. Sie nimmt fünf Parameter: Datenbank-ID, Objekt-ID, Index-ID, Partitionsnummer und Modus. Der Modus entscheidet über alles. LIMITED ist die Voreinstellung, liest nur die Ebene oberhalb der Blattebene, ist billig – liefert dafür aber keine Seitendichte. Wer mit dem Standardaufruf arbeitet und sich wundert, warum `avg_page_space_used_in_percent` leer bleibt, hat genau diesen Fallstrick getroffen. SAMPLED liest jede hundertste Seite und liefert beide Kennzahlen mit brauchbarer Genauigkeit; das ist der Modus für den Alltag. DETAILED liest jede einzelne Seite jeder Ebene und ist exakt.

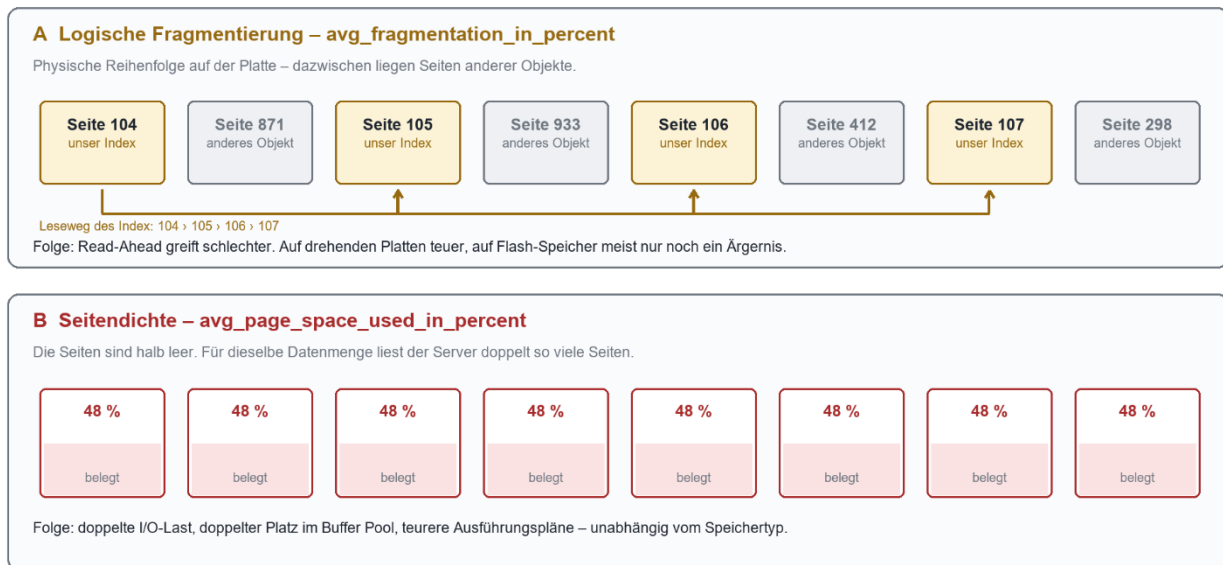
Warnung: DETAILED ist kein Alltagswerkzeug

Ein Aufruf im Modus DETAILED über eine ganze Datenbank scannt sämtliche Indexseiten. Auf einer 2-TB-Datenbank bedeutet das: der Buffer Pool wird leergespült, die Speicheranbindung läuft minutenlang am Anschlag, und die Anwender merken es. Ich habe mehr als einen Vorfall gesehen, bei dem nicht die Indexwartung das Problem war, sondern das Skript, das prüfen sollte, ob Indexwartung nötig ist.

Für den Regelbetrieb nimmst du SAMPLED, filterst auf `page_count` ab 1000 und schränkst über die Objekt-ID auf die Tabellen ein, die dich tatsächlich interessieren. DETAILED hebst du dir für die gezielte Analyse eines einzelnen verdächtigen Index auf.

Was die beiden Kennzahlen wirklich aussagen

Zwei verschiedene Krankheiten: Fragmentierung und Seitendichte



Merksatz: Fragmentierung ist ein Symptom, Seitendichte ist eine Rechnung.

Skizze 3: Logische Fragmentierung und Seitendichte beschreiben zwei unterschiedliche Probleme mit zwei unterschiedlichen Kostenprofilen.

avg_fragmentation_in_percent misst, wie oft die logische Reihenfolge der Indexseiten von ihrer physischen Reihenfolge abweicht. Das schadet, wenn viele Seiten am Stück gelesen werden, weil der Read-Ahead-Mechanismus dann nicht mehr in großen Blöcken vorauslesen kann, sondern viele kleine, wahlfreie Zugriffe absetzt. Auf drehenden Platten war das dramatisch. Auf modernem Flash-Speicher, auf einem ordentlichen SAN und erst recht in Azure SQL Database ist der Unterschied zwischen sequenziellem und wahlfreiem Zugriff so klein geworden, dass Microsoft für die Cloud-Dienste ausdrücklich festhält: Fragmentierung wirkt sich dort kaum noch auf die Abfrageleistung aus.

avg_page_space_used_in_percent misst die Seitendichte, also wie voll die Seiten sind. Und diese Zahl kostet immer. Eine Seitendichte von 50 Prozent heißt schlicht: doppelt so viele Seiten für dieselben Daten. Doppelt so viel Platz auf der Platte, doppelt so viel Platz im Backup, doppelt so viele Lesezugriffe, doppelt so viel Buffer Pool. Zusätzlich rechnet der Optimierer mit der höheren Seitenzahl und wählt womöglich einen anderen, schlechteren Plan. Microsoft formuliert das inzwischen so deutlich, wie man es in Produktdokumentation selten liest: In vielen Workloads bringt eine höhere Seitendichte mehr als eine reduzierte Fragmentierung.

Die 5-, 10- und 30-Prozent-Regel und was heute davon gilt

Die berühmten Schwellenwerte stammen aus einer Zeit, in der Datenbanken auf RAID-Verbünden aus 15.000er-Platten lagen. Sie lauten: unter 5 Prozent Fragmentierung nichts tun, zwischen 5 und 30 Prozent reorganisieren, über 30 Prozent neu aufbauen – jeweils nur für Indizes mit mindestens 1000 Seiten, weil bei kleineren die Messung ohnehin verrauscht ist und die Wartung nichts bringt. Als Startpunkt sind diese Werte weiterhin brauchbar. Als Dogma sind sie es nicht mehr. Microsofts aktuelle Empfehlung ist ausdrücklich, Wartungsentscheidungen nicht allein an festen Grenzwerten festzumachen, sondern den Nutzen im eigenen Workload zu belegen – und Reorganize als Standardverfahren zu betrachten, solange kein konkreter Grund für einen Rebuild spricht.

Kennzahl	Quelle	Aussage	Handlungsschw
----------	--------	---------	---------------

			elle in der Praxis
avg_fragmentation_in_percent	sys.dm_db_index_physical_stats (SAMPLED)	logische Unordnung der Seiten	ab 5 % beobachten, ab 30 % Rebuild erwägen
avg_page_space_used_in_percent	sys.dm_db_index_physical_stats (SAMPLED oder DETAILED)	Füllgrad der Seiten	unter etwa 75 % lohnt sich meist ein Rebuild
page_count	sys.dm_db_index_physical_stats	Größe des Index in Seiten	unter 1000 Seiten gar nicht erst anfassen
forwarded_record_count	sys.dm_db_index_physical_stats	Weiterleitungen im Heap	steigend: Clustered Index anlegen
deleted_rows / total_rows	sys.dm_db_column_store_row_group_physical_stats	Columnstore - Fragmentierung	ab 10 % REORGANIZE
user_seeks, user_scans, user_updates	sys.dm_db_index_usage_stats	Nutzen gegen Pflegeaufwand	Lesezugriffe nahe null: Index infrage stellen
leaf_allocation_count	sys.dm_db_index_operational_stats	Seitenteilungen auf Blattebene	auffällig hoch: Schlüssel oder Fill Factor prüfen

Columnstore misst man vollkommen anders

Für Columnstore-Indizes ist `sys.dm_db_index_physical_stats` das falsche Werkzeug. Fragmentierung ist hier definiert als Anteil der als gelöscht markierten Zeilen an allen Zeilen einer komprimierten Rowgroup, und das liest du aus `sys.dm_db_column_store_row_group_physical_stats`. Bei nicht geclusterten Columnstore-Indizes musst du zusätzlich den Löschpuffer berücksichtigen, den `sys.internal_partitions` unter dem Typ `COLUMN_STORE_DELETE_BUFFER` ausweist. Ebenso wichtig ist die Verteilung der Rowgroup-Größen: Hunderte Rowgroups mit je 20.000 Zeilen sind ein Ladeproblem, kein Wartungsproblem – die Lösung liegt dann im ETL-Batch, nicht im nächtlichen Job.

Rebuild, Reorganize – oder einfach nur Statistiken

Damit sind wir bei der Frage, die in jedem zweiten Betriebshandbuch falsch beantwortet ist. Es gibt nicht zwei Optionen, sondern vier: reorganisieren, neu aufbauen, nur Statistiken aktualisieren oder nichts tun. Die letzten beiden sind häufiger richtig, als es dem Berufsstolz guttut.

Reorganize: der schonende Weg

`ALTER INDEX ... REORGANIZE` arbeitet an Ort und Stelle. Die Engine sortiert die Blattseiten physisch so um, dass sie der logischen Reihenfolge entsprechen, und verdichtet sie dabei auf den eingestellten Fill Factor. Der Vorgang ist immer online: Es werden keine langfristigen Sperren auf dem Objekt gehalten, sondern nur kurze Sperren auf einzelnen Seiten. Lesende und schreibende Zugriffe laufen weiter. Er arbeitet inkrementell, und genau das ist sein unterschätzter Vorteil – brichst du ihn ab oder läuft dein Wartungsfenster aus, bleibt der bis dahin erreichte Fortschritt erhalten. Einen großen Index kannst du so über mehrere Nächte hinweg in Etappen aufräumen.

Zwei Einschränkungen solltest du kennen. Erstens braucht `Reorganize` freien Platz in derselben Datendatei, in der der Index liegt – nicht irgendwo in der Dateigruppe. Ist diese eine Datei voll,

scheitert der Vorgang mit Fehler 1105, obwohl die Dateigruppe insgesamt reichlich Platz hat. Zweitens funktioniert Reorganize nicht, wenn für den Index ALLOW_PAGE_LOCKS auf OFF steht. Und ein Detail, das regelmäßig für Verwirrung sorgt: Reorganize räumt nur die Blattebene auf, nicht die Zwischenebenen.

Rebuild: gründlich, aber teuer

ALTER INDEX ... REBUILD verwirft den Index und baut ihn vollständig neu. Alle Ebenen werden neu geschrieben, die Fragmentierung verschwindet restlos, und die Seitendichte entspricht danach exakt dem Fill Factor. Der Preis steht in drei Posten: Ressourcen, Sperren und Platz. Während des Rebuilds müssen zwei Kopien des Index gleichzeitig auf der Platte Platz finden. CPU, Speicher und I/O laufen deutlich höher als normal. Und das Transaktionsprotokoll wächst – bei einer Datenbank im vollständigen Wiederherstellungsmodell schreibt der Rebuild das komplette Indexvolumen ins Protokoll.

Standardmäßig ist ein Rebuild eine Offline-Operation mit einer exklusiven Sperre auf dem Objekt. Mit ONLINE = ON hält die Engine die Objektsperre nur ganz kurz am Anfang und am Ende, dafür pflegt sie während des gesamten Laufs jede Änderung in zwei Strukturen parallel – schreibende Zugriffe werden also etwas langsamer. Und hier eine Präzisierung, die viele Betriebskonzepte falsch haben: ONLINE = ON gibt es nur in der Enterprise Edition, einschließlich Developer und Evaluation. Die Standard Edition kann es nicht, auch in SQL Server 2022 und SQL Server 2025 nicht. Wer auf Standard fährt, plant Offline-Fenster oder beschränkt sich auf Reorganize. Punkt.

Wo Enterprise vorhanden ist, lohnt der Blick auf fortsetzbare Operationen. Mit RESUMABLE = ON – verfügbar für ALTER INDEX seit SQL Server 2017 und für CREATE INDEX seit SQL Server 2019, jeweils nur zusammen mit ONLINE = ON – lässt sich ein Rebuild pausieren, später fortsetzen oder abbrechen. sys.index_resumable_operations zeigt den Fortschritt. Das ist das Werkzeug für 400-GB-Indizes in Umgebungen mit dreistündigem Wartungsfenster. Ein Detail dazu ist wichtig: Eine pausierte fortsetzbare Operation kostet weiterhin Schreib-Overhead, solange sie nicht abgeschlossen oder abgebrochen ist. Wenn du sie nicht zu Ende führen willst, brich sie ab, statt sie unbegrenzt liegen zu lassen.

Nützliche Zusatzoptionen: SORT_IN_TEMPDB = ON verlagert die Sortierung in die tempdb, was hilft, wenn die tempdb auf schnellerem Speicher liegt. Ein höherer MAXDOP-Wert verkürzt die Laufzeit auf Kosten der CPU. Und WAIT_AT_LOW_PRIORITY mit MAX_DURATION und ABORT_AFTER_WAIT sorgt dafür, dass ein Online-Rebuild beim Warten auf seine kurze Abschlussperre nicht die halbe Anwendung blockiert, sondern nach einer definierten Zeit aufgibt.

Was mit den Statistiken passiert – der wichtigste Punkt

Ein Rebuild aktualisiert die Statistiken auf den Schlüsselspalten des Index, und zwar durch einen vollständigen Scan aller Zeilen – das entspricht einem UPDATE STATISTICS WITH FULLSCAN. Ein Reorganize aktualisiert überhaupt keine Statistiken. Das allein ist schon ein Praxishinweis: Wer nur reorganisiert, braucht einen separaten Statistik-Job.

Interessanter ist die Umkehrung. Microsoft hält in der Produktdokumentation inzwischen ausdrücklich fest, dass die Leistungsverbesserung, die Betriebe nach einem Index-Rebuild beobachten, in vielen Fällen gar nichts mit reduzierter Fragmentierung zu tun hat, sondern mit den nebenbei erneuerten Statistiken: Frische Statistiken führen zur Neukompilierung der Pläne, und ein vorher schlechter Plan wird plötzlich gut. Denselben Effekt bekommst du mit einem UPDATE STATISTICS in Minuten statt in Stunden. Wer seinen nächtlichen Rebuild rechtfertigen will, sollte diesen Vergleich einmal gemacht haben.

Drei Ausnahmen von der Fullscan-Regel gehören dazu, weil sie regelmäßig übersehen werden: Bei partitionierten Indizes und bei fortsetzbaren Operationen aktualisiert der Rebuild die Statistiken

nur mit der Standardstichprobe, nicht mit vollständigem Scan. Und Statistiken auf Spalten, die nicht Teil eines Index sind, fasst ein Rebuild ohnehin nie an.

Tipp: Die Reihenfolge, die sich bewährt hat

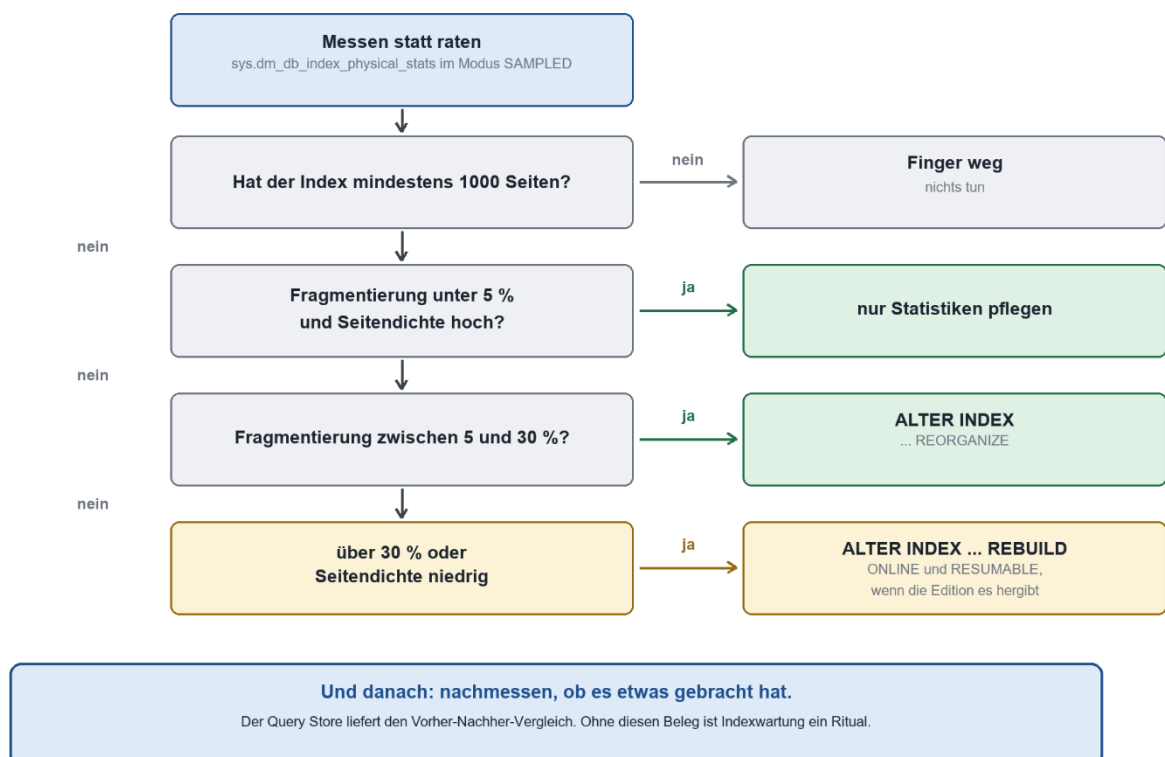
Erst Statistiken aktualisieren – gezielt und nur für das, was sich geändert hat. Dann messen, ob das Problem weg ist. Erst wenn es das nicht ist, über Reorganize nachdenken. Und erst wenn Seitendichte oder Fragmentierung nachweislich weh tun, den Rebuild ansetzen.

In der Praxis heißt das häufig: täglich Statistiken, wöchentlich Reorganize für die auffälligen Indizes, Rebuild nur anlassbezogen. Nicht umgekehrt.

Die vierte Option: gar nichts tun lassen

In Azure SQL Database, in Azure SQL Managed Instance mit der Aktualisierungsrichtlinie „Always up to date“ und in der SQL-Datenbank in Microsoft Fabric gibt es inzwischen die automatische Indexverdichtung, aktivierbar über `ALTER DATABASE ... SET AUTOMATIC_INDEX_COMPACTION = ON`. Sie hängt sich an den Aufräumprozess des Versionsspeichers und verdichtet kontinuierlich genau die Seiten, die kürzlich geändert wurden – mit minimalem Overhead und ohne Wartungsjob. Sie erhöht die Seitendichte, reduziert aber weder Fragmentierung noch aktualisiert sie Statistiken. Für lokal betriebene SQL-Server-Instanzen steht das Verfahren derzeit nicht zur Verfügung; wer dort läuft, bleibt bei Reorganize, Rebuild und Statistikpflege.

Entscheidungsweg der Indexwartung



Skizze 4: Der Entscheidungsweg. Jeder Zweig endet in einer konkreten Maßnahme – einschließlich der Maßnahme, nichts zu tun.

Verfahren	Sperren	Ressourcen	Wirkung	Statistiken	Abbrechbar
REORGANIZE	immer online, kurze	moderat, aber lange Laufzeit	Blattebene entfragmentiert,	keine Änderung	ja, Fortschritt bleibt

	Seitensperren		verdichtet auf Fill Factor		erhalten
REBUILD offline	exklusive Objektsperre	hoch, dafür kürzeste Laufzeit	alle Ebenen neu, optimale Seitendichte	Fullscan-Update	nein, alles verworfen
REBUILD online	kurze Sperre am Anfang und Ende	hoch plus Doppelpflege der Änderungen	wie offline	Fullscan-Update	nein, alles verworfen
REBUILD online resumable	kurze Sperre am Anfang und Ende	hoch, planbar über Zeitscheiben	wie offline	nur Standardstichprobe	ja, pausieren oder abbrechen
UPDATE STATISTICS	keine nennenswerten	gering, Minuten statt Stunden	nichts an der Struktur, aber bessere Pläne	genau das ist der Zweck	ja
Nichts tun	keine	keine	spart Ressourcen, die anderswo fehlen	automatische Aktualisierung greift weiter	entfällt

Automatisieren, ohne den Betrieb zu gefährden

Indexwartung von Hand anzustoßen ist eine Übergangslösung für die erste Woche. Danach gehört sie automatisiert – aber automatisiert heißt nicht „stur jede Nacht alles“, sondern geregelt, begrenzt und nachvollziehbar.

Wartungspläne: der bequeme Einstieg mit harten Kanten

Der Wartungsplan-Assistent im SQL Server Management Studio bietet die Tasks „Index neu organisieren“ und „Index neu erstellen“, die intern nichts anderes tun als ALTER INDEX ... REORGANIZE beziehungsweise REBUILD abzusetzen. Seit einigen Management-Studio-Generationen lassen sich dort auch Fragmentierungsschwellen, eine Mindestseitenzahl und der Scan-Modus einstellen – das ist mehr, als der schlechte Ruf dieser Werkzeuge vermuten lässt. Für kleine und mittlere Umgebungen mit ein paar hundert Gigabyte reicht das oft aus.

Wo Wartungspläne aufhören: Es gibt keine harte Zeitbegrenzung, du kannst keine Reihenfolge nach Dringlichkeit erzwingen, die Statistikpflege ist grob, und der Plan macht keinen Unterschied zwischen der 900-GB-Faktentabelle und der Stammdatentabelle mit 400 Zeilen. Der eingangs erwähnte Kunde mit den sieben Stunden Nacharbeit hatte exakt einen solchen Plan: „Index neu erstellen“ über alle Benutzerdatenbanken, täglich, ohne Schwellenwert.

Warnung: Indexwartung ist ein Hochverfügbarkeitsthema

Ein Index-Rebuild schreibt das gesamte Indexvolumen ins Transaktionsprotokoll. In einer Verfügbarkeitsgruppe geht dieses Volumen über die Leitung zu jedem Sekundärknoten und muss dort durch die Redo-Queue. Ein nächtlicher Vollausbau kann die Redo-Queue über Minuten aufblähen – und genau diese Minuten fehlen dir, wenn um 3 Uhr ein Failover fällig wird.

Weitere Nebenwirkungen: Log-Backups wachsen um Größenordnungen, Log Shipping und Replikation geraten in Rückstand, und lange Abfragen auf lesbaren Sekundärknoten können abgebrochen werden, um den Redo-Thread nicht zu blockieren. Bei Geo-Replikaten mit knapper Ausstattung kann der Rückstand so groß werden, dass ein Neuaufbau des Replikats nötig wird.

Wer eine Verfügbarkeitsgruppe betreibt, plant Indexwartung deshalb gemeinsam mit dem Failover-Konzept. Was dabei sonst noch schiefgeht, steht in [Häufige Fehler bei SQL Server Hochverfügbarkeit](#).

IndexOptimize von Ola Hallengren

Wenn du eine Umgebung jenseits des Wartungsplan-Assistenten betreibst, führt praktisch kein Weg an der frei verfügbaren Prozedur IndexOptimize vorbei. Sie ist seit vielen Jahren der De-facto-Standard, wird gepflegt, ist in tausenden Produktionsumgebungen erprobt und tut genau das, was ein Wartungsplan nicht kann: differenzieren. Sie liest die Fragmentierungswerte selbst aus, ordnet jeden Index einer von drei Stufen zu und wendet je Stufe eine konfigurierte Aktion an – mit sinnvoll gewählten Rückfallebenen, falls die Edition eine Online-Operation nicht hergibt.

Parameter	Vorgabe	Was er bewirkt	Was sich in der Praxis bewährt
@FragmentationLevel1	5	Untergrenze der mittleren Stufe	meist unverändert lassen
@FragmentationLevel2	30	Untergrenze der hohen Stufe	bei Flash-Speicher eher anheben
@FragmentationLow	NULL	Aktion unterhalb Level 1	bewusst leer lassen – nichts tun
@FragmentationMedium	Reorganize, dann Rebuild online, dann offline	Aktion zwischen Level 1 und 2	auf reines Reorganize kürzen
@FragmentationHigh	Rebuild online, dann offline	Aktion oberhalb Level 2	auf Standard Edition auf Reorganize umstellen
@MinNumberOfPages	1000	Untergrenze der Indexgröße	so lassen, kleinere lohnen nicht
@TimeLimit	unbegrenzt	Sekunden bis zum geordneten Abbruch	der wichtigste Parameter überhaupt – immer setzen
@UpdateStatistics	NULL	Statistikpflege für Index-, Spalten- oder alle Statistiken	auf ALL setzen
@OnlyModifiedStatistics	N	nur geänderte Statistiken anfassen	auf Y setzen
@Resumable	N	fortsetzbarer Rebuild	bei sehr großen Indizes und Enterprise auf Y
@SortInTempdb	N	Sortierung in die tempdb verlagern	auf Y, wenn tempdb schnell liegt
@LockTimeout	unbegrenzt	Wartezeit auf Sperren	auf wenige Minuten begrenzen
@LogToTable	N	Protokoll in die Tabelle CommandLog	immer auf Y – sonst rätst du hinterher

Der Parameter, der aus einem gefährlichen Job einen betriebstauglichen macht, ist @TimeLimit. Er sorgt dafür, dass die Prozedur nach der angegebenen Zeit keine neue Operation mehr beginnt und geordnet aussteigt. Zwei Stunden Wartungsfenster heißen dann tatsächlich zwei Stunden – und nicht sieben, weil zufällig die größte Tabelle als letzte drankam. Zusammen mit @LogToTable, das jede ausgeführte Anweisung samt Laufzeit in die Tabelle CommandLog schreibt, hast du außerdem endlich belastbare Zahlen darüber, was deine nächtliche Wartung eigentlich kostet.

Wartungsfenster planen

Die Standardantwort lautet „nachts oder am Wochenende“, und die ist meistens richtig und manchmal falsch. Falsch ist sie in Betrieben mit Schichtbetrieb rund um die Uhr, in denen das ruhigste Fenster der Sonntagvormittag ist. Falsch ist sie auch, wenn nachts der ETL-Lauf, die Vollsicherung und DBCC CHECKDB gleichzeitig laufen – dann konkurriert die Indexwartung mit drei anderen Ressourcenfressern und alle vier werden langsam.

- Reihenfolge festlegen: erst DBCC CHECKDB, dann Indexwartung, dann Statistiken. Prüfe die Konsistenz, bevor du Strukturen anfässt.
- Rollierend arbeiten: nicht jede Datenbank in jeder Nacht. Große Faktentabellen können auch wöchentlich oder monatlich drankommen.

- Log-Backup-Kadenz anpassen: Während eines Rebuilds solltest du häufiger sichern, sonst läuft die Protokolldatei voll oder wächst dauerhaft an.
- Bei Enterprise-Instanzen kann der Resource Governor die Wartung in eine eigene Arbeitsgruppe mit gedeckelten Ressourcen sperren – die Wartung dauert länger, stört aber weniger.
- Zuerst in der Testumgebung mit produktionsnahem Datenvolumen messen. Ein Job, dessen Laufzeit du nicht kennst, ist kein Job, sondern ein Risiko.

Monitoring: bevor der Anwender anruft

Der Query Store ist das Werkzeug, das die Diskussion über Indexwartung vom Bauchgefühl auf Zahlen umstellt. Er speichert Abfragetexte, Pläne, Laufzeitstatistiken und Wartestatistiken über die Zeit und übersteht Neustarts und Failover. Damit kannst du genau das tun, was Microsoft für Wartungsentscheidungen empfiehlt: einen Vorher-Nachher-Vergleich fahren. Kennzahlen vor der Wartung erfassen, Wartung ausführen, danach dieselben Abfragen vergleichen. Bringt es nichts, lässt du es künftighin. In SQL Server 2022 und neuer ist der Query Store für neu angelegte Datenbanken standardmäßig aktiv – bei aus älteren Versionen migrierten Datenbanken schaltest du ihn selbst ein.

Für die reine Indexbeobachtung reicht ein schlichter Agent-Job, der wöchentlich sys.dm_db_index_physical_stats im Modus SAMPLED über die relevanten Tabellen laufen lässt und Fragmentierung, Seitendichte und Seitenzahl mit Zeitstempel in eine Historientabelle schreibt. Nach drei Monaten siehst du daran, welche Indizes wirklich schnell degradieren und welche du seit Jahren aus Gewohnheit anfasst. Denselben Job erweiterst du sinnvollerweise um einen Schnappschuss von sys.dm_db_index_usage_stats und den Missing-Index-Sichten – beides sind flüchtige Daten, die bei jedem Neustart verloren gehen.

Der Data Collector mit dem Management Data Warehouse existiert weiterhin und sammelt Leistungsdaten periodisch in einer zentralen Datenbank, inklusive fertiger Berichte. Er ist nicht abgekündigt, aber er ist auch nicht mehr der Ort, an dem Microsoft die Entwicklung vorantreibt. Wenn du ihn im Haus hast und er läuft, nutze ihn. Wenn du bei null anfängst, investierst du deine Zeit besser in Query Store plus zwei, drei eigene Agent-Jobs, die genau das erfassen, was du wirklich auswertest.

Werkzeug	Wofür	Historie	Aufwand
Query Store	Abfrage- und Planentwicklung, Vorher-Nachher-Vergleich	persistent, übersteht Neustart und Failover	gering, ab SQL Server 2022 für neue Datenbanken aktiv
Agent-Job auf DMV-Basis	Fragmentierung, Seitendichte, Indexnutzung	so lang, wie du sie aufhebst	ein Skript und ein Zeitplan
sys.dm_db_index_usage_stats	Nutzen eines Index gegen seine Pflegekosten	nur seit dem letzten Start	keiner, aber ohne Wegschreiben wertlos
Missing-Index-Sichten	Hinweise auf fehlende Indizes	flüchtig, maximal 600 Zeilen	keiner, braucht aber fachliche Bewertung
Data Collector und Management Data Warehouse	periodische Leistungsdaten mit Berichten	persistent im eigenen Warehouse	spürbare Einrichtung und Pflege
Erweiterte Ereignisse	gezielte Analyse einzelner Effekte	so lang wie das Ziel	hoch, dafür sehr genau

Wichtig: Ohne Messpunkt ist Wartung Folklore

Bevor du an Schwellenwerten schraubst, brauchst du drei Zahlen: die Laufzeit der Wartung, das dabei erzeugte Protokollvolumen und die Antwortzeit der fünf wichtigsten Abfragen davor und danach. Wer diese drei Zahlen nicht hat, diskutiert über Glaubenssätze.

Und die unbequeme Erkenntnis dabei ist häufig: Ein Teil der nächtlichen Wartung lässt sich ersatzlos streichen, ohne dass irgendjemand es merkt – außer am Backup-Volumen und an der Failover-Zeit.

Fazit

SQL Server Indexoptimierung ist keine Nachtschicht, sondern eine Kette von Entscheidungen. Sie beginnt beim Entwurf, weil der Clustered Key die physische Ordnung der Tabelle festlegt und als Ballast in jedem Non-Clustered Index mitreist – eine Entscheidung, die sich später nur mit einem vollständigen Neuaufbau sämtlicher Indizes korrigieren lässt. Sie setzt sich fort bei der Frage, wie viele Indizes eine Tabelle wirklich trägt, denn jeder Index ist gekaufte Lesegeschwindigkeit auf Kredit, abbezahlt in Schreiblast und Speicher. Und sie endet bei der Wartung, die längst nicht mehr aus dem Reflex besteht, alles neu aufzubauen.

Die drei Punkte, die den größten Unterschied machen: Miss Seitendichte, nicht nur Fragmentierung – die Dichte kostet dich auf jedem Speichersystem bares Geld, die Fragmentierung auf modernem Flash meist wenig. Trenne den Statistik-Effekt vom Struktur-Effekt, denn der gefühlte Erfolg des Rebuilds ist überwiegend der Statistik-Effekt, und den bekommst du in Minuten statt in Stunden. Und begrenze deine Wartungsjobs hart in der Zeit, statt sie in den Betriebsbeginn hineinlaufen zu lassen.

Der Rest ist Handwerk: Reorganize als Regelfall, Rebuild als begründete Ausnahme, Columnstore nach eigenen Regeln behandeln, Ergebnisse im Query Store belegen und die Auswirkungen auf Verfügbarkeitsgruppen und Sicherungen einplanen. Wer das konsequent macht, hat am Ende kürzere Wartungsfenster, kleinere Datenbanken, schnellere Failover – und deutlich mehr Ruhe an Freitagnachmittagen.

Häufige Fragen zur SQL Server Indexoptimierung

Ab welchem Fragmentierungsgrad muss ich handeln?

Als Startwert: unter 5 Prozent gar nicht, zwischen 5 und 30 Prozent reorganisieren, darüber einen Rebuild erwägen – und nur bei Indizes ab etwa 1000 Seiten. Diese Werte sind Orientierung, keine Vorschrift. Microsoft rät ausdrücklich davon ab, Wartungsentscheidungen allein an festen Grenzwerten festzumachen. Prüfe stattdessen im Query Store, ob die Wartung die Antwortzeiten deiner wichtigen Abfragen messbar verbessert. Wenn nicht, hast du eine Zahl optimiert und sonst nichts.

Rebuild oder Reorganize – was nehme ich wann?

Reorganize ist der Regelfall: immer online, deutlich sparsamer, jederzeit abbrechbar, ohne Fortschrittsverlust. Rebuild nimmst du, wenn die Seitendichte wirklich niedrig ist, wenn du den Fill Factor ändern willst, wenn ein beschädigter Non-Clustered Index offline neu aufgebaut werden muss oder wenn du den vollständigen Statistik-Scan als Nebeneffekt brauchst. Und bedenke: Rebuild verlangt Platz für zwei Kopien des Index in der Datei.

Warum ist mein Index kurz nach dem Rebuild wieder fragmentiert?

Weil die Ursache nicht beseitigt wurde. Ein Index mit nicht aufsteigendem Schlüssel – typischerweise ein zufälliger GUID – bekommt bei jedem Einfügen wieder Seitenteilungen. Der Rebuild kuriert das Symptom bis zum nächsten Morgen. Die Lösung liegt im Schlüsseldesign, hilfsweise in einem moderat gesenkten Fill Factor für genau diesen Index. Ein zweiter

Grund: Bei sehr kleinen Indizes ist die gemessene Fragmentierung ohnehin weitgehend bedeutungslos, weil die Seiten in gemischten Extents liegen können.

Wie viele Indizes verträgt eine Tabelle?

Es gibt keine Zahl, nur ein Verhältnis. Eine reine Auswertungstabelle verträgt viele, eine Tabelle mit hoher Einfügelast wenige. Der belastbare Prüfstein ist `sys.dm_db_index_usage_stats` über einen vollständigen Geschäftszyklus: Wenn ein Index über Wochen Millionen Pflegevorgänge und eine Handvoll Lesezugriffe sammelt, ist er ein Kostenfaktor ohne Gegenwert. In der Praxis liegen die meisten OLTP-Tabellen mit vier bis acht durchdachten Indizes besser als mit zwanzig zufällig gewachsenen.

Kann ich Indexwartung in der Standard Edition online fahren?

Für Rebuilds nicht. Die Option `ONLINE = ON` ist der Enterprise Edition vorbehalten, einschließlich Developer und Evaluation, und daran hat sich auch in SQL Server 2022 und SQL Server 2025 nichts geändert. Fortsetzbare Operationen setzen `ONLINE = ON` voraus und fallen damit ebenfalls weg. Was du in der Standard Edition hast, ist `Reorganize` – und das ist immer online. Für viele Umgebungen reicht das vollkommen, wenn man den Rebuild-Reflex ablegt.

Soll ich den Fill Factor senken?

Im Regelfall nein. Ein Fill Factor unter 100 kostet dich dauerhaft Speicherplatz, Buffer-Pool-Kapazität und Lesegeschwindigkeit, um gelegentliche Seitenteilungen zu vermeiden. Microsoft empfiehlt 100 beziehungsweise 0 als Standard. Eine begründete Ausnahme sind einzelne Indizes mit nachweislich vielen Seitenteilungen und nicht sequenzieller führender Spalte; dort ist ein Wert zwischen 70 und 95 sinnvoll. Pauschal über die ganze Datenbank ist es fast immer ein Fehler.

Was mache ich mit Columnstore-Indizes?

Erstens: anders messen. Fragmentierung heißt hier Anteil gelöschter Zeilen je komprimierter Rowgroup und steht in `sys.dm_db_column_store_row_group_physical_stats`. Zweitens: `REORGANIZE` ist das Mittel der Wahl, es räumt ab zehn Prozent gelöschter Zeilen auf und führt kleine Rowgroups zusammen – seit SQL Server 2016 erledigt es im Wesentlichen das, wofür man früher einen Rebuild brauchte. Drittens: Seit SQL Server 2019 kümmert sich ein Hintergrundprozess selbständig um kleine offene und stark ausgedünnte Rowgroups, was den Wartungsbedarf in den meisten Fällen erübrigt. Bei partitionierten Tabellen baust du gezielt einzelne Partitionen neu auf, nicht die ganze Tabelle.

Kann ich einen schlecht gewählten Clustered Key nachträglich ändern?

Technisch ja, betrieblich ist es ein Projekt. Der Wechsel bedeutet, den Clustered Index neu zu erstellen, und dabei baut die Engine automatisch sämtliche Non-Clustered Indizes der Tabelle mit neu. Du brauchst Platz für zwei Kopien, ein realistisch bemessenes Fenster, einen Testlauf mit produktionsnahem Datenvolumen und einen Rückfallplan. Bei sehr großen Tabellen ist der Weg über eine partitionierte Kopie mit anschließendem Umschalten oft verträglicher als das Rebuild an Ort und Stelle.

Brauche ich den Data Collector noch, wenn ich den Query Store habe?

In den meisten Fällen nicht. Der Query Store deckt genau das ab, was du für Wartungsentscheidungen brauchst: Pläne, Laufzeiten, Wartestatistiken und Regressionen über die Zeit, und das persistent. Der Data Collector ist weiterhin verfügbar und liefert breitere Systemmetriken mit fertigen Berichten, verlangt dafür aber eine eigene Warehouse-Datenbank und laufende Pflege. Wenn er bereits eingerichtet ist und genutzt wird, spricht nichts dagegen. Für den Neuaufbau ist die Kombination aus Query Store und ein paar gezielten Agent-Jobs der schlankere Weg.

Der Anwender sagt, es sei langsam – wo fange ich an?

Nicht bei der Fragmentierung. Fang bei der konkreten Abfrage an: Welche ist langsam geworden, seit wann, und was sagt der Query Store über die Planänderung? In den meisten Fällen findest du dort eine Planregression durch veraltete Statistiken, einen Parameter-Sniffing-Effekt oder eine gewachsene Datenmenge, für die schlicht ein passender Index fehlt. Erst wenn nichts davon zutrifft, lohnt der Blick auf Seitendichte und Fragmentierung. Der nächtliche Rebuild ist die letzte Antwort auf diese Frage, nicht die erste.

Weiterlesen

- › Alle Beiträge und Grundlagen zum Thema: [SQL Server](#)
- › Wenn Indexwartung auf Verfügbarkeitsgruppen trifft: [Häufige Fehler bei SQL Server Hochverfügbarkeit](#)
- › Analyse und Begleitung im eigenen System: [SQL Server Beratung](#)