Microsoft SQL Server e Oracle - Análise e discussão de Sistemas de Gestão de Bases de Dados

Conteúdo

Abstract

O Microsoft SQL Server [3] é um sistema de gestão de base de dados que foi desenvolvido pela Sybase com a parceria da Microsoft e lançado em 1989. Em 1994 esta parceria terminou com o lançamento da versão para Windows NT e desde então a Microsoft mantém a manutenção do produto.
As suas linguagens de perguntas são o Transact-SQL (T-SQL) e ANSI SQL.
De entre os vários Sistemas de Gestão de Bases de Dados, o SQL Server é reconhecido pela sua alta performance e por ter as licenças mais caras, comparativamente a outros SGBDs, tais como o Oracle.
Neste artigo, analisamos o Microsoft SQL Server em várias vertentes, tais como armazenamento e estrutura de ficheiros, indexação e hashing, transações, etc.
Com o suporte de tal análise, pretende-se abrir uma discussão que coloca frente a frente ambos os SGBDs, Microsoft SQL Server 2019 (última versão estável) e Oracle SQL Developer 18c (utilizado nas aulas), destacando as vantagens e desvantagens da utilização do sistema estudado.

Indexação e Hashing

2.1 Tipos de índices

Um índice é uma estrutura em disco associada a uma tabela ou view que acelera a recuperação de tuplos. Um índice contém chaves criadas a partir de uma ou mais colunas. Essas chaves são armazenadas numa estrutura (B-Tree) que permite que o SQL Server localize o(s) tuplo(s) associado(s) aos valores da chave de forma rápida e eficiente.

O SQL Server suporta vários tipos de índices 2.1, XML, spatial, full-text, clustered, nonclustered, dentro de outros e com a mais recente aquisição de hash indexes em 2014. Apesar da vasta lista de tipos de índices, bitmaps não são suportados para indexação.

2.1.1 Clustered

- Os índices clustered classificam e armazenam os tuplos na tabela ou **views** com base nas suas chaves (ids). Essas são as colunas incluídas na definição do índice. Apenas pode haver um índice clustered por tabela, porque os próprios tuplos só podem ser armazenados por uma ordem.
- É implementado como uma estrutura de índice de B-Tree que oferece suporte à pesquisa rápida dos tuplos, com base nos seus valores de chave de índice clustered.
- Caso uma tabela não tenha um índice clustered, os seus tuplos são guardados desordenadamente numa estrutura chamada heap 2.2.

Listing 2.1: Comando SQL

```
CREATE CLUSTERED INDEX CIX_Customers_CustomerID
    ON dbo.Customers (CustomerID);
```

Primary storage format	Index type
Disk-based rowstore	
	Clustered
	Nonclustered
	Unique
	Filtered
	Spatial
Columnstore	
	Clustered columnstore
	Nonclustered columnstore
Memory-optimized	
	Hash
	Memory-Optimized nonclustered
Others	
	Computed columns
	XML
	Full-Text

Tabela 2.1: Tipos de índices

2.1.2 Nonclustered

- Índices **nonclustered** têm uma estrutura separada dos tuplos. Um índice nonclustered contém os valores da chave de índice e cada entrada do valor da chave tem um apontador para o tuplo que contém o valor da chave.
- O apontador do tuplo chama-se **row locator**. A estrutura deste depende se as páginas de dados são armazenadas numa **heap** ou numa tabela em cluster. Para uma heap, um row locator é um apontador para o tuplo. Para uma tabela clustered, o row locator é a chave de índice clustered.
- Pode-se adicionar colunas não chave ao nível da folha do índice não clustered para ignorar os limites da chave de índice existentes e executar pesquisas totalmente cobertas e indexadas.

Listing 2.2: Comando SQL

```
CREATE NONCLUSTERED INDEX CIX_Customers_CustomerID
    ON dbo.Customers (CustomerID);
```

Um índice nonclustered compartilha o conceito de **B+ Tree** com as mesmas garantias de desempenho. No entanto, tais índices não afetam a organização das páginas de dados, que podem ser agrupadas ou não. Alguns recursos opcionais de índices nonclustered são:

Unique

- Os índices clustered e nonclustered podem ser **unique**. No entanto por default, a criação de um atributo unique cria um índice nonclustered.
- Este tipo de índice garante integridade dos dados guardados na coluna e obriga que a mesma não tenha valores duplicados, tal como **unique constraints**.
- Mesmo que a coluna já seja do tipo unique, criar um índice do tipo unique fornece informação adicional ao mecanismo de otimização de perguntas que produz planos de execução mais eficientes.

Listing 2.3: Comando SQL

```
CREATE UNIQUE INDEX AK_UnitMeasure_Name
    ON Production.UnitMeasure (Name);
```

Filtered

Os índices filtered são mais pequenos e custam menos a manter que maior parte dos índices. São índices bastante úteis para colunas com pequeno número de valores relevantes.

Existem várias vantagens na utilização de índices filtered:

- Melhor performance nas perguntas e plano de qualidade, pois é mais pequeno que uma índice full-table clustered e tem estatísticas filtradas, que são mais exatas que as estatísticas de índice full-table clustered, porque cobre só os tuplos no índice **filtered**.
- Reduz o custo de manutenção do índice em comparação com um índice full-table nonclustered porque é menor e só é mantido quando os dados no índice são alterados. É possível ter um grande número de índices filtrados, especialmente quando eles contém dados que são alterados com pouca frequência. Da mesma forma, se um índice filtered contiver apenas os dados modificados com frequência, o tamanho menor do índice reduz o custo de atualização das estatísticas.
- Reduz o custo de armazenamento do índice quando um índice de **full-table** não é necessário. Pode-se substituir um índice full-table nonclustered por vários índices filtered sem aumentar significativamente os requisitos de armazenamento.

Apesar das várias vantagens deste índice, existem algumas desvantagens na utilização do mesmo:

- Não podem ser criados em **views**.

- Não podem ser criados em coluna com tipo de dados **CLR**.
- Não suportam operações **LIKE**.

Listing 2.4: Comando SQL

```
CREATE NONCLUSTERED INDEX FIBillOfMaterialsWithEndDate
    ON Production.BillOfMaterials (ComponentID, StartDate)
    WHERE EndDate IS NOT NULL;
```

Included columns

Trata-se de um índice **nonclustered** que é estendido para incluir colunas que não sejam chaves, permitindo que estas possam ser do tipos de dados não permitidos por índices de chave primária e que não sejam considerados pelo SQL Server no cálculo de número de índices de colunas do tipo chave ou o seu tamanho.

Este tipo de índice pode melhorar significativamente o desempenho da consulta quando todas as colunas da consulta são incluídas no índice como colunas do tipo chave ou não chave. Existe melhoria no desempenho porque o mecanismo de otimização de perguntas pode localizar todos os valores da coluna dentro do índice, dados da tabela ou índice **clustered** não são acedidos, resultando em menos operações de I/O do disco.

Apesar de ser um índice com as suas vantagens, existem algumas desvantagens na utilização do mesmo:

- Colunas não chave só podem ser definidas em índices nonclustered.
- Não suporta os tipos de dados text, ntext e image.

Listing 2.5: Comando SQL

```
CREATE NONCLUSTERED INDEX IX_Address_PostalCode
ON Person.Address (PostalCode)
INCLUDE (AddressLine1, AddressLine2, City, StateProvinceID);
```

2.1.3 Columnstore

Columnstore são dados organizados logicamente como uma tabela com linhas e colunas e armazenados fisicamente num formato de dados de coluna.

Um índice columnstore é usado para armazenar, recuperar e gerir dados.

Um **rowgroup** é um grupo de linhas que são comprimidas no formato columnstore ao mesmo tempo. Um rowgroup geralmente contém um número máximo de linhas, que é de 1.048.576 linhas.

Um delta rowgroup é um índice de **clustered B-Tree** que é usado apenas com índices columnstore.

Um índice columnstore pode ter mais que um **delta rowgroup**. O conjunto delta rowgroups chama-se deltastore.

Um segmento de coluna é uma coluna de dados dentro do rowgroup.

- Cada rowgroup contém um segmento de coluna para cada coluna na tabela.
- Cada segmento de coluna é compactado e armazenado em armazenamento físico.
- As operações são executadas em grupos de linhas e segmentos de coluna.
- Pequenas cargas e inserts vão para o **deltastore**.
- Para algumas perguntas, o processador de perguntas do SQL Server pode tirar partido do layout columnstore para melhorar os tempos de execução das mesmas.
- Permite melhor desempenho para perguntas comuns como filters, aggregates, grouping e joins.
- É especialmente apropriado para conjuntos de dados de grandes dimensões, do tipo warehousing(clustered) e para realizar análises em tempo real de carga de trabalho OLTP(nonclustered).
- Um índice columnstore oferece uma ordem de magnitude de melhor desempenho do que um índice de B-Tree.

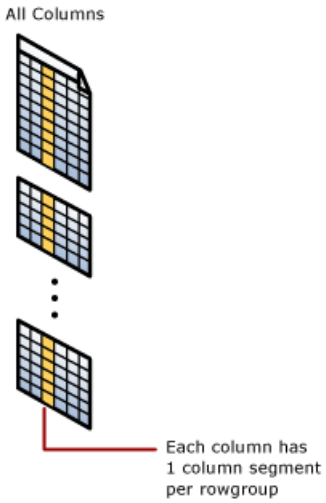

Figura 2.1: Segmento da coluna

Um índice clustered columnstore é o armazenamento físico de toda a tabela.

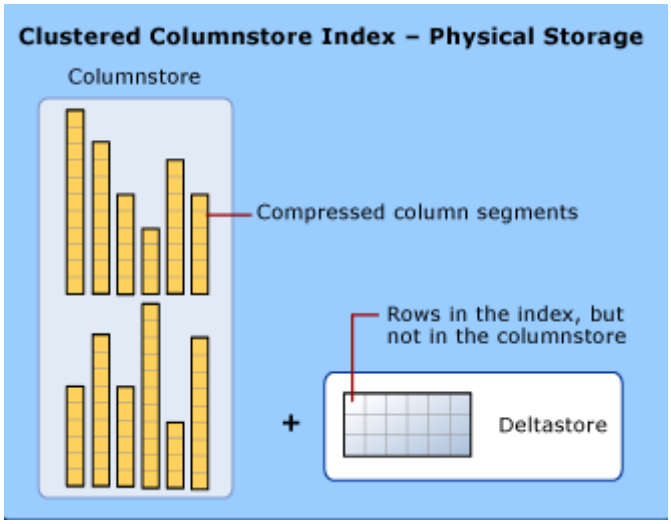

Figura 2.2: Índice de clustered columnstore

Listing 2.6: Comando SQL

```
CREATE CLUSTERED COLUMNSTORE INDEX cci ON Sales.OrderLines;
```

2.1.4 Computed columns

Os índices podem ser criados em computed columns. Além disso, as computed columns podem ter a propriedade **PERSISTED**. Significa que o SQL Server armazena os valores calculados na tabela e atualiza-os quando quaisquer outras colunas das quais a coluna calculada depende são atualizadas. O SQL Server usa esses valores quando cria um índice na coluna e quando o índice é referenciado numa consulta.

Para indexar uma computed column, esta deve ser determinística e precisa. No entanto, o uso da propriedade PERSISTED expande o tipo de computed columns indexáveis para incluir:

- Com base em funções Transact-SQL e CLR e métodos de tipo CLR definidos pelo utilizador que são marcados como determinísticos pelo mesmo.
- Com base em expressões que são determinísticas conforme definido pelo SQL Server, mas imprecisas.

Computed columns derivadas dos tipos de dados image, ntext, text, varchar(max), nvarchar(max), varbinary(max) e xml podem ser indexadas como uma coluna chave ou included não chave, desde que o tipo de dados da computed column seja permitido como uma coluna de chave de índice ou coluna sem chave.

Este tipo de índices é um pouco restrito e pode provocar erros inesperados, como por exemplo em operações de inserts ou update que funcionavam anteriormente, em que a computed column resulta num erro aritmético (sendo c = a/b e a = 1 e b = 0, por exemplo).

Este tipo de índices para ser criado é necessário que:

- Todas as referências de função na computed column tenha o mesmo proprietário da tabela.
- As expressões têm de ser deterministas.
- As expressões têm que ser precisas (não podem ser do tipo float ou real).

Listing 2.7: Comando SQL

```
CREATE TABLE t1 (a INT, b INT, c AS a/b);
CREATE UNIQUE CLUSTERED INDEX Idx1 ON t1(c);
INSERT INTO t1 VALUES (1, 0);
```

2.1.5 Spatial

Um índice spatial fornece a capacidade de executar determinadas operações com mais eficiência em objetos espaciais numa coluna do tipo de dados de geometria.

No SQL Server, os índices spatial são criados usando B-Trees, o que significa que os índices devem representar os dados espaciais bidimensionais na ordem linear das **B-Tree**. Portanto, antes de ler os dados num índice, o SQL Server implementa uma decomposição hierárquica uniforme do espaço. O processo de criação do índice decompõe o espaço numa hierarquia de quatro níveis, nível 1 (o nível superior), nível 2, nível 3 e nível 4.

Cada nível sucessivo decompõe ainda mais o nível acima dele 2.3, de modo que cada célula de nível superior contenha uma grade completa do próximo nível. Num determinado nível, todas as grades têm o mesmo número de células ao longo de ambos os eixos (por exemplo, 4x4 ou 8x8), e as células são todas do mesmo tamanho.

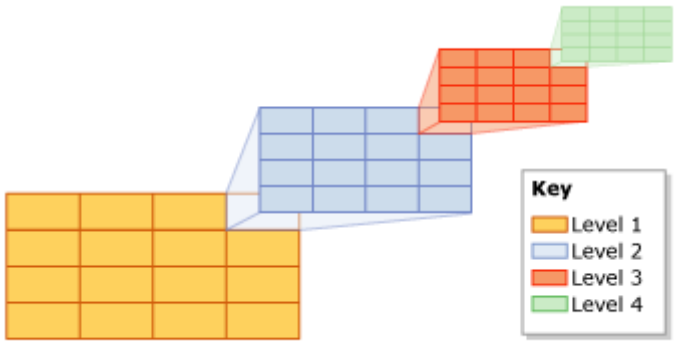

Figura 2.3: Hierarquia de 4 níveis

Listing 2.8: Comando SQL

```
CREATE SPATIAL INDEX SIndx_SpatialTable_geometry_col1
    ON SpatialTable(geometry_col)
    WITH ( BOUNDING_BOX = ( 0, 0, 500, 200 ) );
```

2.1.6 XML

Os índices XML podem ser criados em colunas de tipo de dados XML. Estes indexam todas as **tags**, valores e caminhos nas instâncias XML na coluna e beneficiam do desempenho da consulta. É benéfico usar estes índices quando:

- As perguntas em colunas XML são comuns.
- Os valores XML são relativamente grandes e o resultado das perguntas são relativamente pequenos. A criação do índice evita a análise de todos os dados em tempo de execução e beneficia das pesquisas de índices para um processamento de perguntas eficiente.

Existem dois tipos de índices XML:

- PRIMARY - representação fragmentada e persistente de **BLOBs XML**. Para cada objeto binário XML (BLOB) na coluna, o índice cria várias linhas de dados. O número de linhas no índice é aproximadamente igual ao número de nós no objeto binário XML. Quando uma pergunta recupera a instância XML completa, o SQL Server fornece a instância da coluna XML. As consultas em instâncias XML usam o índice XML PRIMARY e podem retornar valores escalares ou sub-árvores XML usando o próprio índice.
- SECONDARY - usados para melhorar a eficiência das perguntas. Para serem criadas é necessário que já exista um índice do tipo PRIMARY e que o seu tipo seja PATH, VALUE ou PROPERTY.

Listing 2.9: Comando SQL

```
CREATE PRIMARY XML INDEX PXML_ProductModel_CatalogDescription
    ON Production.ProductModel (CatalogDescription);
```

2.1.7 Full-text

Um tipo especial de índice funcional baseado em **tokens**. Este tipo de índice é construído e mantido pela Microsoft Full-Text Engine para o SQL Server. Proporciona

um suporte eficiente para pesquisas sofisticadas de palavras em dados de caracteres em string.

Criar e manter um índice de full-text envolve preencher o índice usando um processo chamado population.

Tipos de population

- Full population - que é o default, que consome bastantes recursos, sendo que em períodos de muita carga, full population deve ser atrasado.
- Automatic or manual population based on change tracking - em que o processo de população é feito manualmente ou automaticamente, de acordo com a modificação de dados da tabela base.
- Incremental population based on a timestamp - que popula de acordo com um timestamp (coluna que tem de ser criada).

Listing 2.10: Comando SQL

```
CREATE FULLTEXT INDEX ON HumanResources.JobCandidate(Resume)
    KEY INDEX ui_ukJobCand
    WITH CHANGE_TRACKING=MANUAL;
```

2.2 Heaps (Tabelas sem Clustered Indexes)

A heap é uma tabela sem um índice clustered. Um ou mais índices nonclustered podem ser criados em tabelas armazenadas como uma heap. Os dados são armazenados na heap sem especificar uma ordem. Normalmente, os dados são armazenados inicialmente na ordem em que as linhas são inseridas na tabela, mas o SQL Server pode mover dados na heap para armazenar as linhas com eficiência.

Utilizações da Heap

Quando uma tabela é armazenada como uma heap, as linhas individuais são identificadas por referência a um identificador de linha de 8 bytes (RID) que consiste no número do arquivo, número da página de dados e slot na página (FileID:PageID:SlotID). O ID da linha é uma estrutura pequena e eficiente.

Heaps podem ser usadas como tabelas de preparação para operações de **inserts** grandes e não ordenadas. Como os dados são inseridos sem impor uma ordem restrita, a operação de insert geralmente é mais rápida do que o insert equivalente num índice clustered.

Quando não utilizar Heap

- Quando os dados são retornados por ordem, com frequência.
- Quando os dados forem agrupados com frequência.
- Quando os intervalos de dados forem consultados com frequência.
- Quando não houver índices **nonclustered** e a tabela for grande, a menos que o que seja pretendido é retornar todo o conteúdo da tabela sem nenhuma ordem especifica.
- Quando os dados forem atualizados com frequência. Ao atualizar um registo e a atualização usar mais espaço nas páginas de dados do que está a ser usado no momento, o registo terá de ser movido para uma página de dados que tenha espaço livre suficiente.

A criação de uma tabela heap depende somente da existência de um índice clustered. Caso este exista, tem de ser removido para que a tabela possa ser uma heap, mas esta operação pode trazer problemas, pois se a tabela já tiver índices nonclustered, estes serão apagados, pois remover ou criar índices clustered, força que a tabela inteira seja reescrita.

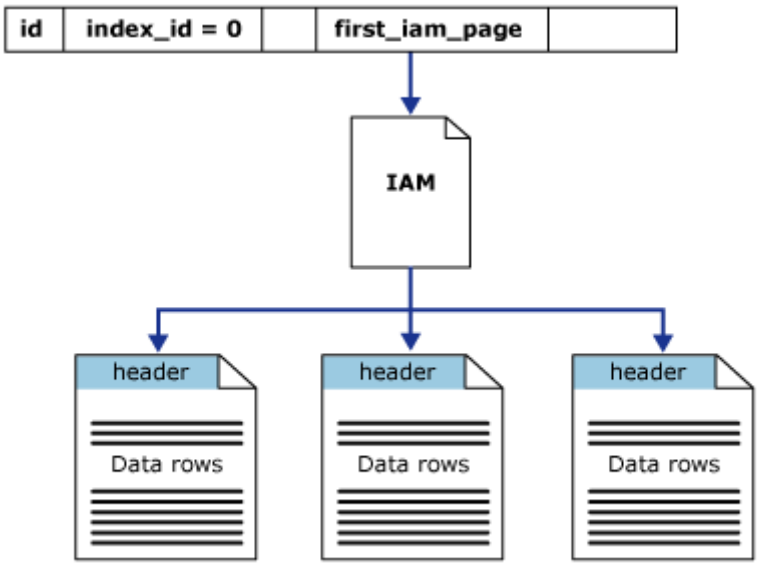

Figura 2.4: Estrutura de uma heap

2.3 Arquitetura de páginas e extensões

A unidade fundamental de armazenamento de dados do SQL Server é a página. O espaço em disco alocado para um arquivo de dados (.mdf ou .ndf) numa base de dados é logicamente dividido em páginas numeradas de 0 a n. As operações de

I/O de disco são executadas ao nível da página. Ou seja, o SQL Server lê ou grava páginas de dados inteiras.

As extensões são uma coleção de oito páginas fisicamente ligadas e são usadas para gerir as páginas com eficiência. Todas as páginas são organizadas em extensões.

Páginas

As páginas contém até 8KB de dados sendo que as base de dados do SQL Server têm 128 páginas por 1MB. Cada página começa com 96 bytes de cabeçalho onde é guardado a informação do sistema. Informação esta que inclui o número da página, o tipo de página 2.3, a quantidade de espaço livre na página e o ID da unidade de alocação do objeto que possui a página.

- Data
- Index
- Text/Image
- Global Allocation Map, Shared Global Allocation Map
- Page Free Space (PFS)
- Index Allocation Map
- Bulk Changed Map
- Differential Changed Map

As linhas de dados são colocadas na página em série, começando imediatamente após o cabeçalho. Uma tabela de deslocamento de linha começa no final da página e cada tabela contém uma entrada para cada linha da página. Assim, a função da tabela de deslocamento da linha é ajudar o SQL Server a localizar linhas numa página muito rapidamente. As entradas na tabela de deslocamento de linha estão na sequência inversa da sequência das linhas na página.

Suporte a linhas grandes

As linhas não podem conter páginas, no entanto, partes da linha podem ser movidas para fora da página da linha para que a linha possa ser maior. A quantidade máxima de dados e sobrecarga contida numa única linha numa página é de 8.060 bytes (8 KB). No entanto, isso não inclui os dados armazenados no tipo de página Texto/Imagem.

Essa restrição é relaxada para tabelas que contém colunas do tipo varchar, nvarchar, varbinary ou sql_variant. Quando o tamanho total da linha de todas as colunas fixas e variáveis numa tabela excede o limite de 8.060 bytes, o SQL Server move dinamicamente uma ou mais colunas de tamanho variável para páginas na unidade de alocação ROW_OVERFLOW_DATA, começando pela coluna com

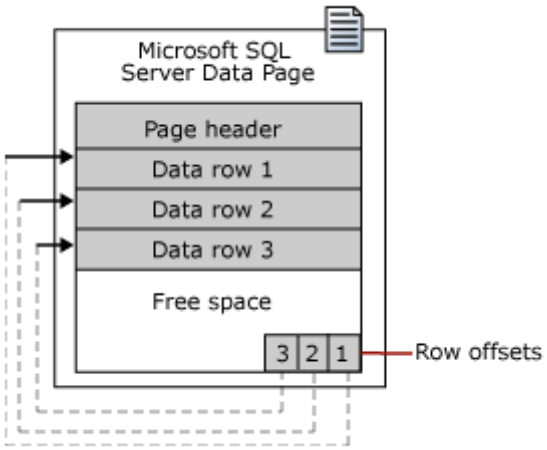

Figura 2.5: Estrutura de uma Página

a maior comprimento. Isto é feito sempre que uma operação de insert ou update aumenta o tamanho total da linha além do limite.

Quando uma coluna é movida para uma página na unidade de alocação ROW_OVERFLOW_DAT um apontador de 24 bytes na página original na unidade de alocação IN_ROW_DATA é mantido. Se uma operação subsequente reduzir o tamanho da linha, o SQL Server moverá dinamicamente as colunas de volta para a página de dados original.

Extensões

As extensões são a unidade básica na qual o espaço é gerido. Uma extensão são oito páginas fisicamente contíguas, ou 64 KB. Significa que as base de dados do SQL Server têm 16 extensões por megabyte.

O SQL Server tem dois tipos de extensões (2.6):

- **Uniform** - todas as oito páginas na extensão só podem ser usadas pelo objeto proprietário.
- **Mixed** - extensões são compartilhadas por até oito objetos. Cada uma das oito páginas na extensão pode pertencer a um objeto diferente.

Gestão de espaço livre

Page Free Space (PFS) registam o estado de alocação de cada página. O **PFS** tem 1 byte para cada página, registando se a página está alocada e, em caso afirmativo, se está vazia, 1 a 50 por cento cheia, 51 a 80 por cento cheia, 81 a 95 por cento cheia ou 96 a 100 por cento cheia.

Depois de uma extensão ser alocada a um objeto, o SQL Server usa as páginas **PFS** para registar quais páginas da extensão estão alocadas ou livres. A quantidade de espaço livre numa página é mantida apenas para páginas heap e de texto/imagem. É usado quando o SQL Server precisa localizar uma página com espaço livre

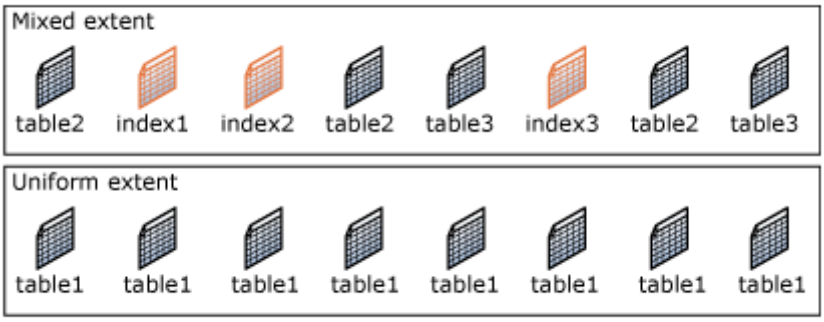

Figura 2.6: Tipos de extensões

disponível para conter uma linha recém-inserida. Os índices não exigem que o espaço livre da página seja monitorizada 2.7, porque o ponto no qual se insere uma nova linha é definido pelos valores da chave de índice.

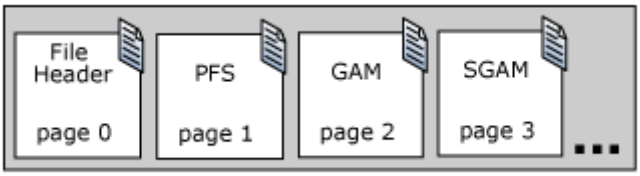

Figura 2.7: Organização de extensões

Gestão do espaço usado por objetos

Uma página **Index Allocation Map (IAM)** mapeia as extensões numa parte de 4 GB de um arquivo de base de dados usado por uma unidade de alocação. Uma unidade de alocação é um dos três tipos:

- **IN_ROW_DATA** - Contém uma partição de uma **heap** ou índice.
- **LOB_DATA** - Contém tipos de dados de objetos de grande dimensão como **XML** e binários.
- **ROW_OVERFLOW_DATA** - Contém dados de comprimento variável armazenados em colunas VARCHAR, NVARCHAR, VARBINARY ou SQL_VARIANT que excedem o limite de tamanho de linha (8.060 bytes).

Cada partição de uma heap ou índice contém pelo menos uma unidade de alocação IN_ROW_DATA. Pode também conter uma unidade de alocação LOB_DATA ou ROW_OVERFLOW_DATA, dependendo do esquema da heap ou índice.

Uma página IAM cobre um intervalo de 4 GB num arquivo e tem a mesma cobertura de uma página GAM ou SGAM. Se a unidade de alocação contiver extensões de mais de um arquivo ou mais de um intervalo de 4 GB de um arquivo, haverá várias

páginas do IAM ligadas numa cadeia IAM. Portanto, cada unidade de alocação tem pelo menos uma página IAM para cada arquivo no qual possui extensões.

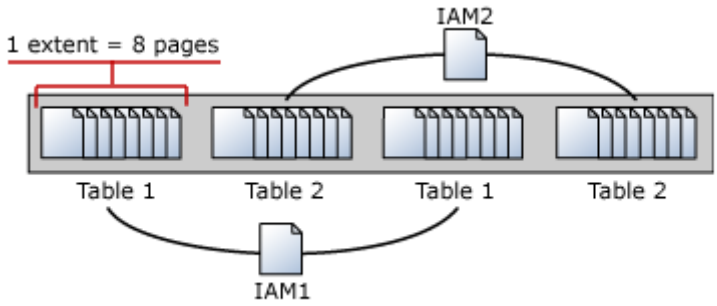

Figura 2.8: Páginas IAM

Uma página IAM tem um cabeçalho que indica a extensão inicial do intervalo de extensões mapeadas pela página. Também possui um **bitmap** no qual cada bit representa uma extensão. O primeiro bit no mapa representa a primeira extensão no intervalo, o segundo bit representa a segunda extensão e assim por diante. Se um bit for 0, a extensão que ele representa não é alocada à unidade de alocação que possui o IAM. Se o bit for 1, a extensão que ele representa será alocada.

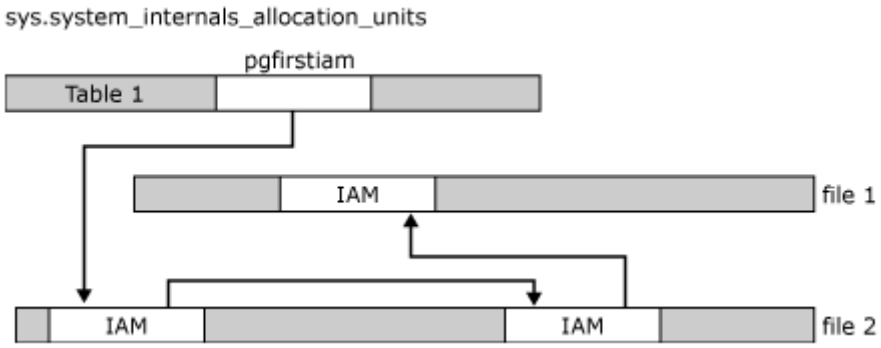

Figura 2.9: Páginas IAM

Gestão de extensões modificadas

O SQL Server usa duas estruturas de dados internas para gerir extensões modificadas por operações de bulk copy e extensões modificadas desde o último backup completo. Essas estruturas de dados aceleram bastante os backups diferenciais. Também aceleram o log de operações de bulk copy quando uma base de dados usa um modelo de recuperação bulk-logged. Como as páginas **Global Allocation Map (GAM)** e **Shared Global Allocation Map (SGAM)**, essas estruturas são bitmaps em que cada bit representa uma única extensão.

- **Differential Changed Map (DCM)** - Verifica quais as extensões que foram alteradas desde a última instrução BACKUP DATABASE.

- **Bulk Changed Map (BCM)** - Verifica quais as extensões que foram modificadas por operações em log em massa desde a última instrução BACKUP LOG.

2.4 Hash index

Um índice de hash consiste numa matriz de apontadores e cada elemento da matriz é chamado de **hash bucket**.

- Cada bucket tem 8 bytes, que são usados para armazenar o endereço de memória de uma linked list de entradas de chaves.
- Cada entrada é um valor para uma chave de índice, mais o endereço da sua linha correspondente na tabela com otimização de memória subjacente.
- Cada entrada aponta para a próxima entrada numa linked list de entradas, todas encadeadas ao bucket atual.

A função hash é aplicada às colunas de chave de índice e o resultado da função determina em qual bucket essa chave é adicionada. Cada bucket tem um apontador para linhas cujos valores de chave com hash são mapeados para esse bucket.

A função de hash usada para índices de hash tem as seguintes características:

- A função hash é determinística. O mesmo valor de chave de entrada é sempre mapeado para o mesmo bucket no índice de hash.
- Várias chaves de índice podem ser mapeadas para o mesmo bucket.
- A função de hash é equilibrada, significa que a distribuição de valores da chave de índice em buckets normalmente segue uma distribuição de Poisson ou bell curve.
- A distribuição de Poisson não é uma distribuição uniforme.
- Se duas chaves de índice forem mapeadas para o mesmo bucket, haverá uma colisão de hash. Um grande número de colisões poderá afetar o desempenho das operações de leitura.

Contagem de buckets do índice de hash

A contagem do bucket é especificada no momento da criação do índice e pode ser alterada com a operação REBUILD.

Na maioria dos casos, a contagem de buckets seria idealmente entre 1 e 2 vezes o número de valores distintos na chave de índice. Nem sempre se pode prever quantos valores uma determinada chave de índice terá. O desempenho geralmente ainda é bom se o valor BUCKET_COUNT for inferior ou igual a 10 vezes o número real de valores-chave.

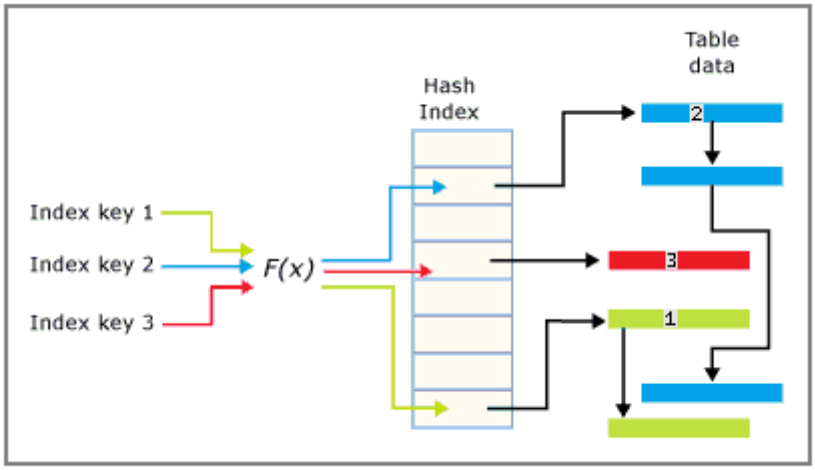

Figura 2.10: Índice de Hash

Adicionar mais buckets não reduz o encadeamento de entradas que compartilham um valor duplicado. A taxa de duplicação de valor é usada para decidir se uma hash é o tipo de índice apropriado, não para calcular a contagem de buckets.

A ter em atenção ao utilizar índices de hash:

- Excelente quando o predicado na cláusula WHERE especifica um valor exato para cada coluna na chave de índice de hash.
- Fraco quando o predicado na cláusula WHERE procura um intervalo de valores na chave de índice ou quando estipula um valor específico para a primeira coluna de uma chave de índice de hash de duas colunas, mas não especifica um valor para outras colunas da chave.
- Um índice de hash pode existir apenas numa tabela com otimização de memória.
- Um índice de hash pode ser definido como UNIQUE ou nonclustered.

Com um índice de hash, os dados são acedidos por meio de uma tabela de hash na memória. Estes índices consomem uma quantidade fixa de memória, que é uma função da contagem de buckets.

Listing 2.11: Comando SQL

```
ALTER TABLE MyTable_memop
ADD INDEX ix_hash_Column2 UNIQUE
HASH (Column2) WITH (BUCKET_COUNT = 64);
```

2.5 Bw-Tree

Os índices nonclustered na memória são implementados usando uma estrutura de dados chamada Bw-Tree [2], originalmente imaginada e descrita pela Microsoft Research em 2011.

A Bw-Tree pode ser entendida como um mapa de páginas organizado por ID de página (PidMap), um serviço para alocar e reutilizar IDs de página (PidAlloc) e um conjunto de páginas ligadas entre elas e entre o mapa de páginas. Esses três subcomponentes de alto nível compõem a estrutura interna básica de uma árvore Bw.

A estrutura é semelhante a uma B-Tree no sentido de que cada página tem um conjunto de valores de chave que são ordenados e há níveis no índice, cada um apontando para um nível inferior e as folhas apontam para uma linha de dados. No entanto, existem várias diferenças.

Assim como os índices de hash, várias linhas de dados podem ser ligadas. Os apontadores de página entre os níveis são IDs de página lógica, que são deslocamentos numa tabela de mapeamento de páginas que, por sua vez, possuem o endereço físico de cada página.

Point lookups são semelhantes às B-Trees, exceto que, como as páginas são ligadas em apenas uma direção, o SQL Server segue os apontadores da página certa, em que cada página não folha tem o valor mais alto do seu filho, invés do valor mais baixo, que é o caso nas B-Tree.

Se uma página de nível de folha tiver que ser alterada, o SQL Server não modifica a própria página. Invés disso, cria um registo delta que descreve a alteração e anexa-o à página anterior. Em seguida, atualiza o endereço da tabela de mapa de página dessa página anterior para o endereço do registo delta que agora se torna o endereço físico dessa página.

Existem três operações diferentes que podem ser necessárias para gerir a estrutura de uma Bw-Tree:

- Delta consolidation 2.11
- Split page 2.12
- Merge page 2.13

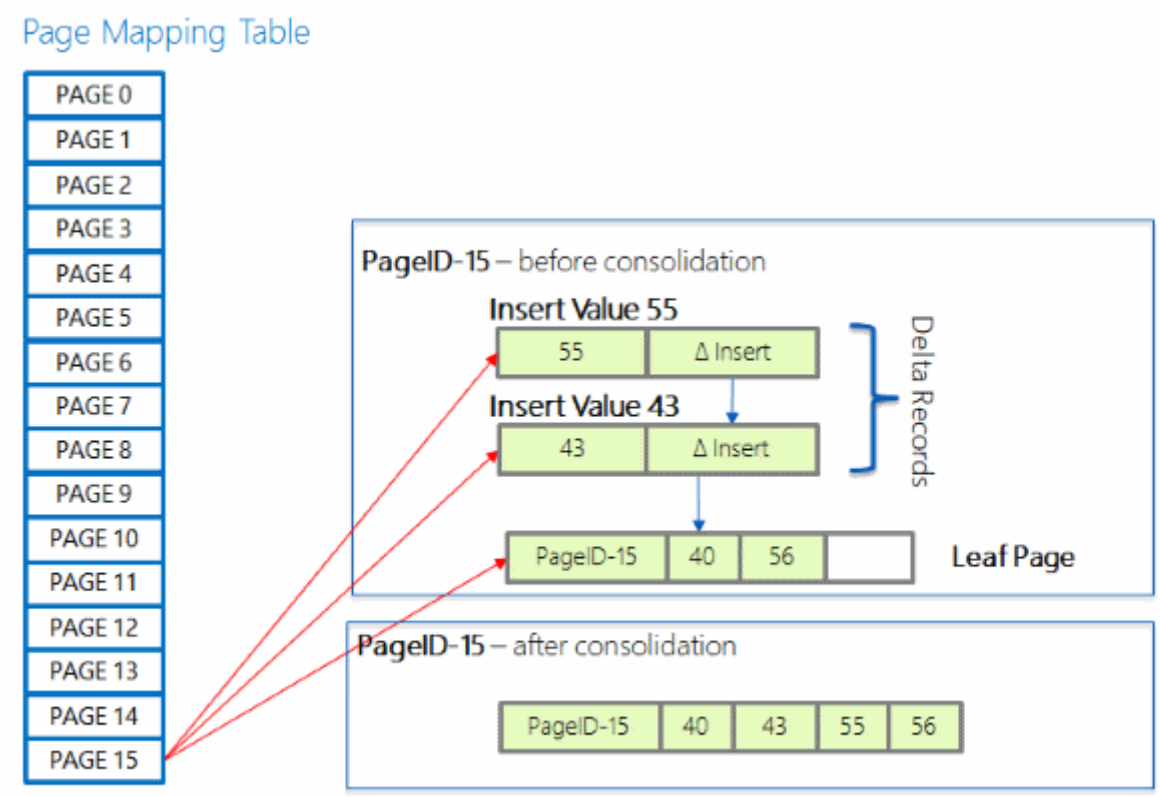

Figura 2.11: Delta consolidation

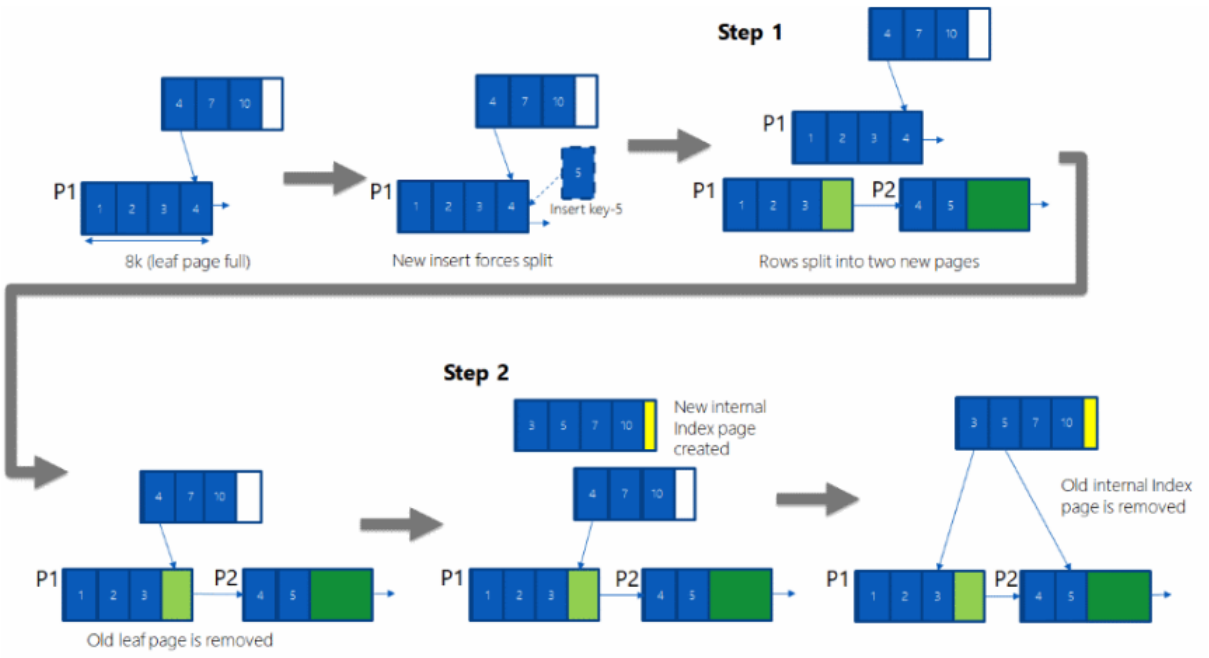

Figura 2.12: Split page

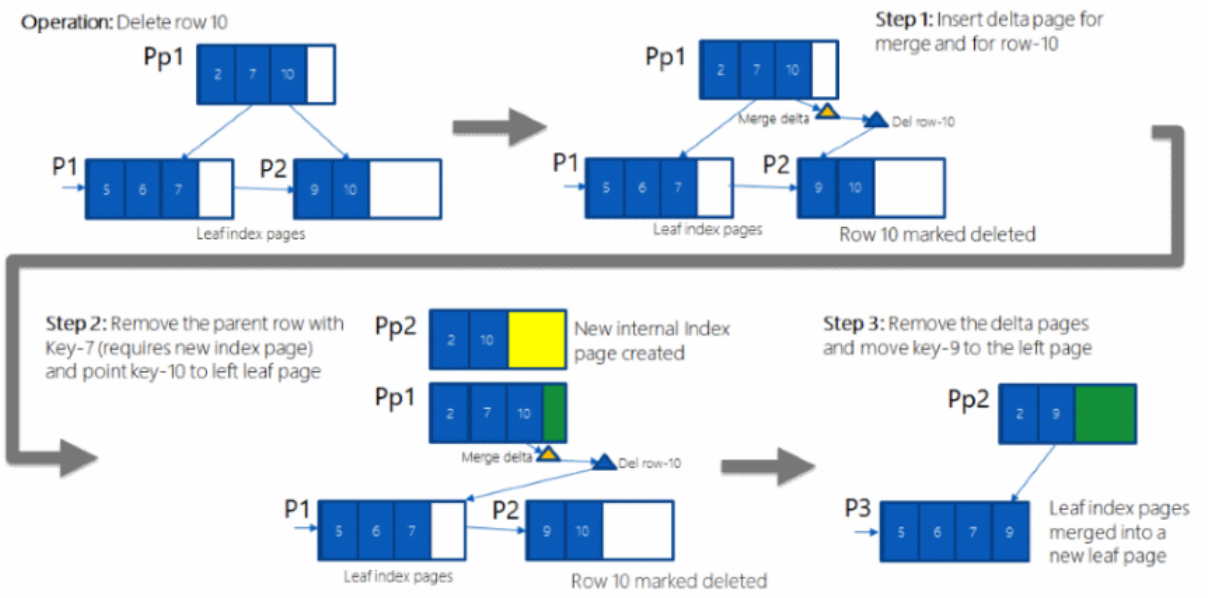

Figura 2.13: Merge page

Processamento e otimização de perguntas

3.1 Modos de execução

O SQL Server pode processar perguntas de dois modos distintos:

- **Modo Linha**, utilizado em bases de dados tradicionais
- **Modo Batch**, utilizado em cenários de Data Warehousing

Em Modo Linha, ao executar uma pergunta, lê-se uma linha integralmente e depois, para cada linha lida extraem-se as colunas necessárias para o resultado dessa pergunta.

Em Modo Batch, as linhas são processadas em vários lotes (*batches*). Este modo de execução opera sobre dados compactados sempre que possível e elimina o operador de exchange utilizado no **Modo Linha** levando assim a melhor desempenho e paralelismo. Adicionalmente se os dados a serem acedidos estiverem em índices *columnstore*, os lotes de linhas são lidos em segmentos de colunas e o SQL Server lê apenas as colunas necessárias para o resultado da pergunta.

3.2 Operações de JOIN suportadas

As operações de join suportadas pelo SQL Server são [4]:

- Inner Join
- Left Outer Join
- Right Outer Join
- Full Outer Join
- Cross Join

De notar que Inner Join pode ser utilizado na cláusula WHERE e FROM de um SELECT enquanto que qualquer Outer Join ou Cross Join apenas pode ser utilizado na cláusula FROM.

Estas operações são implementadas por estes diferentes algoritmos:

- Nested Loop Join
- Merge Join
- Hash Join
- Adaptive Join

Na fase de otimização de perguntas todos estes algoritmos são avaliados de maneira a escolher o mais eficiente para a pergunta em questão.

3.2.1 Nested Loop Join

Este tipo de join funciona fazendo *matching* de uma das linhas de uma das tabelas de input com todas as linhas da outra tabela, repetindo este processo para cada linha da primeira tabela. Este algoritmo é particularmente eficiente se a tabela externa do join for pequena, com, no máximo, dez linhas, e a tabela interna do join estiver indexada nas colunas do join e for grande, no entanto nos restantes casos Nested Loop Joins geralmente não são as soluções mais eficientes.

Existem 3 tipos de Nested Loop Join:

- Naive Nested Loop Join: Em que se percorre as tabelas integralmente
- Indexed Nested Loop Join: Em que se utiliza um índice sobre uma das tabelas para otimizar a operação não tendo que percorrer as tabelas integralmente
- Temporary Index Nested Loop Join: Em que se constrói um índice como parte do plano de execução da pergunta e no fim do mesmo, este é eliminado

Existe também o atributo OPTIMIZED que quando utilizado faz uso de um Batch Sort na tabela interna quando esta é de grandes dimensões

3.2.2 Merge Join

Este algoritmo requer que as tabelas estejam ordenadas pela coluna onde vai ser feito o *join*. Graças a este requerimento é possível "prever"quando deixará de haver *matches* e parar o algoritmo antes de se percorrer as tabelas na totalidade.

Existe também o Mesh Join de muitos-para-muitos que lida com dados duplicados em ambas as tabelas de *input*, neste caso, utiliza-se uma tabela auxiliar para guardar linhas e quando são encontradas entradas duplicadas uma dessas entradas vai para o início das linhas duplicadas á medida que cada linha duplicada da outra entrada seja processada. Adicionalmente caso exista um predicado residual, após o Merge Join ser efetuado, apenas as linhas que satisfazem esse predicado são retornadas.

Quando a operação é feita sobre 2 tabelas de tamanhos similares a sua eficiência é comparável à de um Hash Join, no entanto, se o tamanho das tabelas for muito diferente a sua eficiência diminui. A sua eficiência também diminui caso as tabelas não sejam ordenadas previamente visto que este processo é obrigatório e demorado. Também é altamente eficiente caso o volume de dados seja grande e seja possível aceder-lhes através de um índice *B-Tree*.

3.2.3 Hash Join

Este algoritmo é utilizado em casos em que as tabelas são de grandes dimensões, não estão ordenadas e não têm índices associados. É também utilizado em resultados intermédios de perguntas complexas porque estes resultados, geralmente, não têm índices associados e porque como a otimização de perguntas apenas consegue estimar o tamanho de resultados intermédios, o que implica que os algoritmos utilizados têm que ser eficientes e, caso o tamanho dos resultados intermédios seja maior do que o esperado, têm que degradar suavemente.

O algoritmo de Hash Join tem 2 inputs *build* e *probe* e tem 2 fases, fase *build* e fase *probe* que são onde o processamento destes inputs é feito, a otimização de perguntas garante que o input *build* é o mais pequeno dos dois, no entanto é possível que a otimização esteja errada, nestes casos ocorre um *role reversal* que simplesmente troca os inputs, esta troca não é utilizada no cálculo do custo de um plano de execução.

Existem 3 tipos de Hash Join:

- In-Memory Hash Join
- Grace Hash Join
- Recursive Hash Join

In-Memory Hash Join

Neste caso primeiro ocorre a fase *build* em que é calculado o *hash* da chave de cada linha do input *build* e depois é construida uma *hash table* em memória.Caso exista memória suficiente para armazenar o input *build* na totalidade então todas as linhas deste são armazenadas na *hash table* em memória, senão são utilizados *hash buckets* para armazenar as linhas. De seguida passa-se para a fase *probe* em que se faz o *hash* da chave de cada linha do input *probe* e utilizam-se então os resultados da fase *build* para produzir *matches*.

Grace Hash Join

Este tipo de Hash Join ocorre quando não existe memória livre suficiente para guardar a *hash table* resultante do input *build*. A solução passa por dividir o Hash Join em vários passos mais pequenos, cada um deles com uma fase *build* e *probe*. Inicialmente, os dois inputs são processados e divididos em partições através de uma função de *hash* sobre o *hash* das chaves de cada linha, efetivamente dividindo um Hash Join em vários Hash Joins com inputs mais pequenos que possivelmente já são válidos para um In-Memory Hash Join.

Recursive Hash Join

Um Recursive Hash Join ocorre quando o tamanho do input **build** é tal que seria necessário vários níveis de junção, partição e níveis de partição para executar. Neste tipo de Hash Join utilizam-se grandes operações I/O assíncronas de modo que uma

thread possa manter vários discos em uso, de modo a tornar estas partições o mais rápido possíveis.

Nem sempre é possível à otimização de perguntas prever qual o melhor algoritmo de Hash Join a utilizar pelo que geralmente começa-se por user o In-Memory Hash join e gradualmente passa-se para o Grace Hash Join e depois para o Recursive Hash Join de acordo com o tamanho do input *build*

3.2.4 Adaptive Join

Este algoritmo é utilizado em Modo Batch e permite atrasar a decisão entre usar um Nested Loop Join ou um Hash Join. Esta decisão passa a ser tomada após ter sido processada a primeira tabela do *join* e, com base em certos parâmetros, decidir qual algoritmo de join utilizar.

Este algoritmo tem requisitos de memória mais elevados do que um plano de execução para um Nested Loop Join equivalente, esta memória extra é utilizada caso o algoritmo escolhido passe a ser um Hash Join. Este algoritmo pode ser ligado e desligado com os seguintes comandos:

- `ALTER DATABASE SCOPED CONFIGURATION SET BATCH_MODE_ADAPTIVE_JOINS = ON`
- `ALTER DATABASE SCOPED CONFIGURATION SET BATCH_MODE_ADAPTIVE_JOINS = OFF`

3.2.5 Utilização um algoritmo

Para forçar a utilização de um algoritmo pela otimização de perguntas, é apenas preciso colocar os seguintes prefixos à operação de *join*:

- LOOP: Para utilizar Nested Loop Join
- MERGE: Para utilizar Merge Join
- HASH: Para utilizar Hash Join

Exemplo[5]:

```
SELECT p.Name, pr.ProductReviewID
FROM Production.Product AS p
LEFT OUTER HASH JOIN Production.ProductReview AS pr
ON p.ProductID = pr.ProductID
ORDER BY ProductReviewID DESC;
```

3.3 Processamento de perguntas

Uma pergunta é lida e repartida em unidades lógicas, como palavras chave, expressões, operadores e identificadores, de seguida é construida uma árvore que descreve as transformações necessárias para chegar aos resultados da pergunta.[9]

Após este processo, a otimização de perguntas cria vários planos de execução a partir desta árvore e escolhe o que tem o menor custo passando assim para a fase de execução após a qual temos o resultado da pergunta.

3.4 Otimização de perguntas

A otimização de perguntas passa por criar vários planos de execução. Estes planos de execução têm informação de como executar a pergunta, desde a ordem em que aceder ás tabelas até aos métodos de extrair dados de cada tabela, de computar cálculos, de filtrar, agregar e ordenar dados de um tabela. A dita otimização de perguntas é assim escolher, de entre todos os planos gerados, o mais eficiente.

3.4.1 Planos de execução

No SQL Server a otimização de perguntas é feita com base no custo de um plano que está relacionado com a quantidade de recursos que consome. Em alguns casos as perguntas têm complexidade elevada o que leva ao número de planos possível ser extremamente grande, nestes casos, a otimização de perguntas utiliza algoritmos para calcular o plano com o custo razoavelmente perto do menor custo efetivamente possível.

Este custo, também chamada de *Cardinality Estimate* é baseado em 2 fatores[7]:

- Número de linhas que a que cada nível do plano tem que aceder
- Modelo de custo dos algoritmos a utilizar na pergunta, indicados pelos operadores nela presentes

O plano escolhido nem sempre é o com menor custo mas sim aquele que calcula os resultados mais rapidamente com menor custo. Depois de um plano de execução de uma pergunta ter sido escolhido este é guardado em *cache* para poder ser utilizado posteriormente sem ter que ser calculado novamente. Estes planos guardados podem ser apagados manualmente ou em até pelo SQL Server caso haja necessidade de libertar memória removendo os que são menos utilizados. Também é possível recompilar planos, isto pode ser uma operação forçada pelo utilizador ou caso hajam mudanças em *views* ou *procedures* que sejam referenciados pelo plano, ou até se as estatísticas utilizados pelo plano forem atualizadas.

É possível ver os planos de execução estimado bem como o atual de uma pergunta clicando nos botões na *toolbar* ao lado direito do botão de execução da pergunta, ou alternativamente com o atalho **ctrl**+**l** para o plano de execução estimado e **ctrl**+**m** para o plano de execução escolhido.

3.4.2 Parametrização

A parametrização de perguntar é utilizada para que perguntas que apenas diferem em valores de constantes não tenham que ter planos de execução diferentes, por exemplo[9]:

```
SELECT -
FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 1;
```

```
SELECT -
FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 4;
```

Estas duas perguntas diferem apenas no valor a ser comparado com *ProductSubcategoryID* pelo que não seria necessário terem planos de execução diferentes. No geral, o SQL Server tenta detetar situações onde parametrização faz sentido e aplica-a mas quando as perguntas se tornam mais complexas isso deixa de ser possível.

É então possível parametrizar manualmente as perguntas de modo a "ajudar"o SQL Server a perceber que se pode efetivamente parametrizar a pergunta. Para o caso acima, a parametrização seria:

```
DECLARE @MyIntParm INT
   SET @MyIntParm = 1
   EXEC sp_executesql
   N'SELECT -
   FROM AdventureWorks2014.Production.Product
   WHERE ProductSubcategoryID = @Parm',
   N'@Parm INT',
   @MyIntParm
```

3.5 Materialização

É possivel materiazlizar *views* no SQL Server mas neste caso são chamadas de Indexed Views e tratam-se efetivamente de *views* com índices. O primeiro índice criado sobre uma view tem que ser um índice *clustered* único mas indices subsequentes não têm que ser *clustered*.

Existem algumas configurações necessárias para que as Indexed View tenham resultados consistentes e sejam mantidas corretamente[8]: Outro requerimento é que

SET options	Required value	Default server value
ANSI_NULLS	ON	ON
ANSI_PADDING	ON	ON
ANSI_WARNINGS1	ON	ON
ARITHABORT	ON	ON
CONCAT_NULL_YIELDS_NULL	ON	ON
NUMERIC_ROUNDABORT	OFF	OFF
QUOTED_IDENTIFIER	ON	ON

a *view* tem que ser determinística, isto é, todas as expressões na lista do SELECT bem como na cláusula de WHERE e GROOUP BY são deterministas e é recomendado que não se use dados do tipo *float* pois o resultado pode variar de processador para processador, assim colunas com dados do tipo *float* apenas podem aparecer em colunas não-chave. Existem ainda outros requerimentos ligados a permissões que podem ser consultados na referência [8].

Exemplo de criação de Indexed View:
Passo 1: Criar a *view*

```
IF OBJECT_ID ('Sales.vOrders', 'view') IS NOT NULL
DROP VIEW Sales.vOrders ;
GO
CREATE VIEW Sales.vOrders
    WITH SCHEMABINDING
    AS
        SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Revenue,
        OrderDate, ProductID, COUNT_BIG(*) AS COUNT
        FROM Sales.SalesOrderDetail AS od, Sales.SalesOrderHeader AS o
        WHERE od.SalesOrderID = o.SalesOrderID
        GROUP BY OrderDate, ProductID;
GO
```

Passo 2: Criar um índice *clustered* único sobre a view

```
CREATE UNIQUE CLUSTERED INDEX IDX_V1
    ON Sales.vOrders (OrderDate, ProductID);
GO
```

Gestão de transações e controlo de concorrência

4.1 Introdução

Segundo a documentação da Microsoft [6], "Uma transação é uma única unidade de trabalho. Se uma transação for bem sucedida, todas as modificações de dados feitas durante a transação são confirmadas e tornam-se permanentes na base de dados. Se uma transação se deparar com erros e tiver de ser abortada ou for feito um ROLLBACK específico, todas as modificações são apagadas."
Também importante é referir que, antes e após uma transação bem sucedida, a base de dados tem que estar num estado de consistência.

4.2 Propriedades Básicas

As transações, Transact-SQL, em MS SQL Server, respeitam as propriedades ACID, que passamos a relembrar:

- **Atomicidade**: A transação vê todas as suas operações a perdurarem na base de dados, ou nenhuma.
- **Consistência**: A execução isolada de uma transação numa base de dados consistente, preserva a consistência.
- **Isolamento**: As transações são independentes, ou seja, transações não devem reconhecer a execução concorrente das restantes, mesmo que executadas de forma concorrente.
- **Durabilidade**: Uma vez que as operações efetuadas na transação são confirmadas, através de um COMMIT, perduram de forma permanente na base de dados.

Transações Nested são permitidas e funcionam sob o princípio de que a transação que está nested, mesmo que declare um COMMIT, aguarda que a transação exterior declare um COMMIT (ou um ROLLBACK).

4.3 Modos de Transação

O MS SQL Server apresenta quatro tipos de modos de transação, **Autocommit**, **Explícito**, **Implícito** e **Batch-Scoped**.

Em relação ao primeiro primeiro trata cada instrução como uma transação em si. Podemos considerar o mais simples de entre os modos.
De seguida, quando explicitamente se inicia uma transação, através do comando BEGIN TRANSACTION ou BEGIN DISTRIBUTED TRANSACTION e, seguido de um bloco de instruções, damos informação ao SGBD que queremos terminar através de um COMMIT ou ROLLBACK.
As transações implícitas, ainda que terminadas com um COMMIT ou ROLLBACK, são aquelas que são iniciadas implicitamente quando a anterior termina.
As transações batch-scoped são transações aplicadas a um conjunto MARS, que sigifica Multiple Active Result Sets. Ou seja, é necessária uma sessão MARS aberta, e a utilização de um COMMIT, caso contrário, ROLLBACK automático.

4.4 Comandos

- **BEGIN TRANSACTION**

  Este comando permite começar uma transação genérica.

- **BEGIN DISTRIBUTED TRANSACTION**

  Este comando permite começar uma transação distribuída, ou seja, implica a utilização de cloud, como o Microsoft Azure.

- **COMMIT TRANSACTION**

  Tal como é transversal em SQL, um COMMIT de uma transação confirma as alterações feitas à base de dados, as quais se garante que perdurem.

- **COMMIT WORK**

  Esta declaração funciona de forma idêntica a COMMIT TRANSACTION, exceto que COMMIT TRANSACTION aceita um nome de transação definido pelo utilizador.

- **SAVE TRANSACTION**

  O MS SQL Server permite que um utilizador coloque marcadores dentro de transações, funcionando como uma espécie de divisores internos. Este comando permite colocar um desses marcadores, que define um ponto para o qual uma transação pode regressar se parte da transação for cancelada condicionalmente. Se uma transação for transferida de volta para um ponto de salvamento, deve proceder à sua conclusão com mais declarações T-SQL, se necessário, e uma declaração COMMIT TRANSACTION, ou deve ser totalmente cancelada, transferindo a transação de volta para o seu início.

Todas as declarações ou procedimentos da transação são anulados.
Não é possível utilizar este comando em transações distribuídas!

- **ROLLBACK TRANSACTION**

  Este comando aborta uma transacão, ou faz recuar até um marcador interno à transação. Pode utilizar a TRANSAÇÃO ROLLBACK para apagar todas as modificações de dados efectuadas desde o início da transação ou para um marcador/savepoint.
  Também liberta os recursos detidos pela transação!

- **ROLLBACK WORK**

  Semelhante à semântica do COMMIT, a declaração deste comando funciona de forma idêntica a ROLLBACK TRANSACTION, exceto que ROLLBACK TRANSACTION aceita um nome de transação definido pelo utilizador.

4.5 Isolamento e Níveis de Isolamento

No MS SQL Server, é possível escolher entre vários níveis de isolamento, cada um com as suas particularidades, permitindo gestão de transações isoladas, tal como está explícito nas propriedades ACID.

Os cinco níveis são os seguintes:

1. READ COMMITTED

   Este é o nível default no MS SQL Server. Ideal para a prevenção de **dirty reads**, pois, as operações de uma transação não podem ler dados que tenham sido modificados e não confirmados (confirmados por um COMMIT) por outras transações.

2. READ UNCOMMITTED

   Tal como o nome sugere, difere do READ COMMITED na medida em que as operações de uma transação podem ler dados que não foram necessariamente confirmados (confirmados por um COMMIT). Isto, como é óbvio, abre caminho a **dirty reads**, que não podem ser evitados nem mesmo com a utilização de locks, pois é semelhante à utilização da opção NOLOCK em todas as tabelas em todas as declarações SELECT de uma transação.

3. REPEATABLE READ

   Opera de forma igual ao READ COMMITTED, com a seguinte particularidade: uma primeira transação não pode ler dados que tenham sido modificados mas ainda não confirmados por uma segunda transação, o que requer que a segunda transação termine para que a primeira possa fazer a leitura desses dados.

4. SNAPSHOT ISOLATION

O nome deve-se ao facto de, para uma transação neste nível de isolamento, os dados são lidos de uma snapshot/estado transacionavelmente consistente da base de dados, capturada antes do início dessa mesma transação. Quer nível de isolamento, quer no SERIALIZABLE existe a garantia de não ocorrerem **leituras-fantasma***.

5. SERIALIZABLE

Este rígido nível de isolamento estabelece que uma transação não possa aceder a dados modificados por outra transação (contudo, não confirmados através de um COMMIT) até que esta termine, garantido que os dados acedidos pela mesma se mantêm consistentes. Além disso, uma transação também não pode inserir linhas cujos valores das chaves tenham valores em comum com as chaves das linhas acedidas por uma outra transação.

* leituras-fantasma ocorrem quando uma transação executa uma consulta mais do que uma vez e recebe resultados distintos, e geralmente acontecer quando uma outra transação insere ou apaga dados da mesma tabela, onde os domínios das restrições quer da consulta quer da inserção ou remoção se intersetam.

Neste SGBD, alternar entre níveis de isolamento é possível através do comando

Listing 4.1: Comando SQL

```
SET TRANSACTION ISOLATION LEVEL
{NIVEL_DESEJADO};
```

Estes níveis de isolamento apresentam a seguinte ordem crescente de rigidez e comparação com os presentes no Oracle: **1. READ UNCOMMITED** (igual ao Oracle), **2. READ COMMITED** (igual ao Oracle), **3. REPEATABLE READ** (também conhecido como READ-ONLY no Oracle), **4. SNAPSHOT ISOLATION** (não existe no Oracle), **5. SERIALIZABLE** (igual ao Oracle).

4.6 Deadlocks

Deadlocks têm potencial para acontecer quando se utilizam protocolos com locks, como é o caso do MS SQL Server. Quando duas ou mais transações concorrentes pretendem aceder a dados em que estão protegidos por locks, pelo que o mecanismo deste SGBD está preparado para abortar uma das transações.

O MS SQL Server tem um monitor para deteção da ocorrência de deadlocks e, por default, é executado a cada 5 segundos mas, com um deadlock a ocorrer, este valor pode chegar a 100 milissegundos. [10] A variação deste intervalo de espera depende da frequência com que são detetados deadlocks.
A transação escolhida para ser abortada é selecionada consoante um critério: os

recursos utilizados; após a deteção de um deadlock, é automaticamente executado um ROLLBACK sob a mesma.

4.7 Consistência

O MS SQL Server tem um protocolo de verificação de consistência que pode ser ativado pelo comando geral

```
DBCC CHECKDB
```

e tem as seguintes especificações.

- Executa o comando

  ```
  DBCC CHECKALLOC
  ```

  , que verifica o espaço alocado pela base de dados.

- Executa o comando

  ```
  DBCC CHECKTABLE
  ```

  , que verifica a integridade dos dados de todas as tabelas e vistas da base de dados.

- Executa o comando

  ```
  DBCC CHECKCATALOG
  ```

  , que verifica a consistência de catálogos da base de dados.

- Verifica e valida as ligações entre metadados e diretorias do sistema de ficheiros, assim como os dados do Service Broker*.

* Service Broker é um monitor de mensagens e framework de comunicação entre serviços de bases de dados distribuídas, com suporte do Microsoft Azure.

NOTA: Um utilizador do SGBD necessita de permissões de administrador, ou ser o detentor da base de dados, para poder verificar a consistência.

4.8 Exemplos de Destaque

- Exemplo 1 (Utilização dos mecanismos de isolamento, savepoints, entre outros)

```
CREATE TABLE people(
   id INT PRIMARY KEY NOT NULL,
   name_person VARCHAR(50) NOT NULL,
);
```

```
-- colocar o nível de isolamento em READ UNCOMMITED
SET TRANSACTION ISOLATION LEVEL
READ UNCOMMITTED

BEGIN TRANSACTION

   INSERT INTO people (id, name_person) VALUES (1, 'José');

   -- savepoint save1 criado após a inserção da pessoa José
   SAVE TRANSACTION save1

   -- são inseridas outras duas pessoas na base de dados, a Maria e o Paulo
   INSERT INTO people (id, name_person) VALUES (2, 'Maria');
   INSERT INTO people (id, name_person) VALUES (3, 'Paulo');

   -- rollback para o savepoint marcado por save1
   ROLLBACK TRANSACTION save1

-- COMMIT, a este ponto, confirma as alterações até ao savepoint save1
COMMIT

-- esta consulta verifica que a tabela people só contêm o tuplo (1, 'José')
select * from people;
```

- Exemplo 2 (Verificação de Consistência)

```
-- Verifica a consistência da base de dados corrente
DBCC CHECKDB;
GO

-- Verifica a consistência da base de dados dada por SBD
    -- Verifica sem índices non-clustered
DBCC CHECKDB (SBD, NOINDEX);
GO

-- Verifica a consistência da base de dados corrente
    -- Opção ativada para silenciar mensagens informativas
DBCC CHECKDB WITH NO_INFOMSGS;
GO
```

Oracle SQL Developer vs MS SQL Server

Apesar de serem duas base de dados relacionais, existem algumas diferenças entre o MS SQL Server e Oracle SQL Developer. [1]

MS SQL Server é uma base de dados menos poderosa, mais limitada a nível de indexação, recursos de acesso a dados e em algumas situações, a relação entre objetos e relações pode ser corrompida.

- Oferece suporte e documentação online e suporte ao produto ao vivo.
- Fornece opção de personalização avançada para mapeamentos de tipo de dados e excluir e renomear objetos.
- Exibe mensagens de erro e aviso sobre a migração numa janela de progresso.
- Um recurso de monitorização de atividade com filtragem e atualização automática.
- Possui processamento de perguntas integrados para melhorar a saída do otimizador de perguntas e torná-las mais eficientes.

O SQL Developer possui recursos exclusivos, como processamento paralelo de perguntas, recursos de tratamento de erros 24 horas por dia, 7 dias por semana entre outras vantagens.

- Facilidade de recuperação de dados.
- Foi o primeiro RDBMS que foi construído exclusivamente para fins comerciais.
- Lida com uma grande quantidade de dados rapidamente.
- É altamente escalável, portátil, distribuída e programável.
- A confiabilidade e a integridade dos dados são mantidas porque seguem as propriedades ACID.
- Suporta uma ferramenta de recuperação que realiza backups regulares da base de dados e auxilia nas recuperações da base de dados.

Na seguinte tabela 5.1 conseguimos ver as principais diferenças entre os dois:

SQL Server	**SQL Developer/Oracle**
Simples de usar e uma sintaxe fácil	Mais complexo mas com sintaxe mais eficiente
Transact-SQL e ANSI-SQL	PL/SQL
Sem índices bitmap invés disso usa índices baseados em funções e chaves inversas	Permite índices bitmap
5 níveis de isolamento de transações	Não incluí nível de isolamento SNAPSHOT (ou equivalente)
Não permite acessos concorrentes	Permite acessos concorrentes
Usa método de linhas ou páginas bloqueadas Não permite leituras durante a sua modificação	Usa uma cópia do registo, para que possa ser possível ler os dados durante a sua modificação
Não permite execução paralela	Permite execução paralela
Suporte Windows e Linux	Suporte para várias plataformas
Valores mudam antes de commit	Valores só mudam depois de commit

Tabela 5.1: SQL Server vs SQL Developer/Oracle

Bibliografia

[1] Hevo. Sql server vs sql developer, 2021.

[2] Microsoft. The bw-tree: A b-tree for new hardware platforms, 2021.

[3] Microsoft. Sql server, 2021.

[4] Microsoft. Sql server join, 2021.

[5] Microsoft. Sql server join hints, 2021.

[6] Microsoft. Transactions (transact-sql), 2021.

[7] Microsoft. Sql server cardinality estimation, 2022.

[8] Microsoft. Sql server indexed views, 2022.

[9] Microsoft. Sql server query processing architecture guide, 2022.

[10] SQLShack. What are sql server deadlocks and how to monitor them, 2017.