5. Como podemos obter paralelismo com mais de um disco? Explico os tipos de RAID e como eles podem ser usados por um SGBD.

As duas formas mais comuns de se paralelizar o acesso a dados são:

1. Dividindo os dados bit a bit: um exemplo dessa configuração seria mapear cada bit de um byte para um disco de um conjunto de 8 discos. Com essa divisão, toda leitura usa todos os discos, de modo que a quantidade de acessos que podem ser processados por segundo é a mesma, mas a taxa de leitura de dados é multiplicada pela quantidade de discos.
2. Dividindo os dados bloco a bloco: os blocos são distribuídos igualmente entre os discos. A vantagem dessa organização está na leitura de arquivos grandes: nesse caso, é possível ler *n* blocos no mesmo tempo em que se leria 1 bloco com 1 disco, ou seja, uma alta taxa de leitura. Ao ler um único bloco, somente um disco é usado (e nesse caso não há aumento de velocidade), mas os outros discos ficam ociosos e podem executar outras ações independentemente dessa leitura.

Essas duas ideias são usadas juntamente com mecanismos de redundância ou correção de erros para elaborar tipos de RAID (*Redundant Array of Independent Disks*), conjuntos de discos que oferecem diferentes graus de paralelismo, redundância, detecção e correção de erros.

Mecanismos de redundância ou correção de erros geralmente usados são:

1. Espelhamento: trata-se do uso de 1 disco adicional para armazenar o conteúdo de 1 dos discos usados, como uma cópia. E todas as escritas são executadas no disco original e na sua cópia. Isso significa que, para usar o espelhamento, é necessário usar o dobro de discos. No entanto, há uma vantagem: a falha de um disco não causa perda de dados pois há uma cópia em outro disco.
2. Bits de paridade: associar um bit de paridade a cada byte, que indica se a quantidade de bits no byte que estão no nível 1 é par (paridade = 0) ou ímpar (paridade = 1). A vantagem desse mecanismo é que ele torna possível detectar erros. Para verificar a paridade, basta comparar o bit de paridade armazenado com a calculada usando os bits do byte desejado. Caso os valores não sejam iguais, há um erro. Usando informações adicionais (como códigos de detecção de erros extras), é possível corrigir os erros usando a informação de paridade. Pode-se dedicar discos inteiros ao armazenamento de paridade ou distribuir os bits de paridade entre todos os discos do sistema.

Os tipos de RAID são (sendo N o número total de discos envolvidos):

- RAID 0 (block striping): um conjunto de discos com dados divididos bloco a bloco, sem redundância. (Todos os N discos são de dados. Nenhuma tolerância à falha.)
- RAID 1 (mirroring): espelhamento (redundância, sem *ECC*) de dados divididos bloco a bloco. (N/2 discos são de dados “originais” e N/2 discos são de dados “espelhados”. Máxima tolerância à falha, de N - 1 discos.)
- RAID 2 (bit striping): Similar ao RAID 0, mas com divisão de dados bit a bit, implementa uso de bits de paridade (*ECC - Error correcting code*) por byte (assim como faz a memória), gravados em discos dedicados. (Alguns discos do array são usados para

dados, outros para paridade, na proporção da quantidade de bytes disponíveis para dados no array. Tolerância à falha de 1 disco.).

- RAID 3 (byte striping): usa as mesmas técnicas que o RAID 2, mas utiliza apenas 1 bit de paridade por setor, aproveitando-se do fato de que a controladora de discos pode informar o erro na leitura de um bit específico do setor. Usar essa informação com o bit de paridade calculado por setores dos demais discos permite corrigir erros. (Um disco de paridade para vários discos de dados, na proporção da quantidade de setores disponíveis para dados no array. Tolerância à falha de 1 disco.).
- RAID 4 (block striping): usa as mesmas técnicas que o RAID 3, mas intercala a paridade por bloco e não mais por bit. Oferece paralelismo (igual ao RAID 0), detecção e correção (similar ao RAID 3) de erros, mas aumenta o número de acessos a disco para uma escrita de um único bloco, que passa a precisar de 4 acessos a disco: 2 escritas, uma no disco onde estão os dados e outra no disco de paridade; e 2 leituras, uma dos dados antigos e outra da paridade antiga, para poder calcular a nova paridade. (Um disco de paridade para vários discos de dados, na proporção da quantidade de blocos disponíveis para dados no array. Tolerância à falha de 1 disco.).
- RAID 5 (“(block + parity) striping”): similar ao RAID 4, mas implementa paridade distribuída de forma idêntica aos dados (todos os discos são usados). A vantagem de distribuir a paridade é que todos os discos podem ser acessados para obter ou escrever dados, enquanto na abordagem usada em RAIDs anteriores um disco era reservado para paridade. Isso implica em um aumento na quantidade de acessos que podem ser atendidos em um período de tempo (nível de I/O). (Os N discos são, ao mesmo tempo, de dados e de paridade. Tolerância à falha de 1 disco.).
- RAID 6 (“(block + 2*parity) striping”): o esquema de redundância P + Q. Similar ao RAID 5, mas usa informações adicionais de redundância para adicionar tolerância a múltiplas falhas, como ECCs alternativos (p. ex: Reed-Solomon), ao invés da paridade “simples” (p. ex: Código de Hamming). É, portanto, resistente a duas falhas simultâneas de discos do array. (Os N discos são, ao mesmo tempo, de dados e de paridade. Tolerância à falha de 2 discos.).

Fonte: livro do Korth (ver aqui https://robot.bolink.org/ebooks/Database%20System%20Concepts%206e%20By%20Abraham%20Silberschatz,%20Henry%20Korth%20and%20S%20Sudarshan.pdf)

https://en.wikipedia.org/wiki/Standard_RAID_levels

12. Como funciona um índice baseado em árvore B+?

A indexação por árvore B+ segue um esquema multinível, mas com uma estrutura particular. Uma árvore B+ é composta por vários nós que contém uma determinada quantidade (maior que 1) de chaves de busca e ponteiros. As chaves de busca dentro de um nó estão sempre ordenadas em ordem crescente. A quantidade de ponteiros na árvore é um parâmetro dela, chamado *fan-out* (que vamos chamar aqui de N). O papel desses ponteiros muda dependendo se o nó é uma folha ou não.

- Se o nó não é uma folha (nó interno): um ponteiro Pn aponta para um nó na camada inferior da árvore que possui uma chave de busca Kx (sendo x qualquer número) maior ou igual a Kn-1 (caso n-1 seja maior que 0) e menor que Kn.
- Se o nó é uma folha: um ponteiro Pn na folha aponta para um registro no arquivo de dados com chave de busca Kn. Se n = N, esse ponteiro é o último da folha e aponta para a próxima folha da árvore. Isso é útil para buscas em que há vários registros com a mesma chave de busca.

Todos os nós exceto o raiz possuem um limitante inferior de tamanho: nenhum deles pode ter menos de N/2 ponteiros.

Abaixo, vamos detalhar como funciona a busca, a inserção e a remoção dos registros.

Busca de registro:

A busca toma como argumento um valor para a chave de busca (digamos, V). A execução se dá da seguinte forma:

1. Atribuir como nó inicial da busca a raiz.
2. Se o nó atual da busca for interno, varrer o nó até encontrar uma chave Ki que seja maior ou igual a V.
    a. Se Ki = V, encontramos o valor e o próximo nó é o apontado pelo ponteiro Pi+1 (que aponta para nós com chaves maiores ou iguais a Ki).
    b. Se Ki > V, não encontramos V. O próximo nó deve ter uma chave de valor menor que Ki, logo é o nó referido pelo ponteiro Pi.
    c. Se não encontramos Ki maior ou igual a V, V é maior que a maior chave de busca desse nó. Por isso, o próximo nó é o apontado por PN (o último ponteiro).
3. Se o nó atual for uma folha:
    a. Buscar uma chave Ki = V
        i. Se não for encontrada, o programa deve fornecer um erro de “registro não encontrado”
        ii. Caso seja encontrada, o ponteiro Pi fornece a posição do primeiro registro no arquivo de dados com a chave de busca V. Varremos, então, a folha em busca de outros registros de chave V. Para isso,

vamos iniciar um processo iterativo usando como condição de parada Ki != V.

1. Caso essa condição não valha no final da folha, usa-se PN para ir à próxima folha e retomamos o passo anterior.

Uma modificação do passo 2-a-ii também permite uma busca por um intervalo de chaves. Nesse caso, efetuamos a busca na árvore pelo valor mínimo do intervalo, e a condição de parada é alterada para Ki != [valor máximo do intervalo], fazendo a varredura na folha retornar todos os nós com chaves dentro do intervalo.

Inserção de registro:

Utilizando o mesmo procedimento que é empregado para a busca de registros, encontramos o nó folha no qual o registro estaria. Nós então inserimos o registro (contendo uma chave de busca e um ponteiro para onde o registro está armazenado) nesse nó de forma a manter as chaves de busca organizadas.

Temos que considerar casos nos quais um nó tem que ser dividido. De forma simplificada, o procedimento de inserção de um registro R funciona da seguinte maneira:

1. Encontra-se o nó folha no qual R seria inserido.
    a. Caso haja espaço para a inserção ele é inserido de forma a manter as chaves de busca dos registros da folha ordenadas e a inserção é concluída
    b. Caso não haja espaço a folha é dividida em dois nós de forma que, de uma folha que tinha n registros, os $\lceil n/2 \rceil$ registros permaneçam na mesma e o restante seja transferido para a nova folha criada. A menor chave da nova folha criada tem que ser inserida na folha pai daquela que foi dividida:
        i. Se há espaço na folha pai, a inserção é feita de forma ordenada e o algoritmo de inserção é encerrado
        ii. Se não há espaço para inserção o nó é conceitualmente expandido, a chave é inserida e os ponteiros são distribuídos, $\lceil n/2 \rceil$ ficam no nó original e o restante vai para o novo nó. As chaves que ficam entre os ponteiros que ficaram no original, permanecem no original e as que ficam entre os ponteiros que foram para o novo nó vão para o novo nó. E a chave que fica entre o último ponteiro do nó inicial e o primeiro ponteiro do novo nó vai para o nó pai do que foi dividido

Caso o nó pai esteja cheio o mesmo também é dividido e o algoritmo continua recursivamente até que se chegue a um nó que não precise ser dividido ou se crie uma nova raiz. No pior caso a recursão sobe toda a árvore até que se crie uma nova raíz e a árvore aumenta um nível de profundidade.

Remoção de registro:

Usando a mesma técnica da busca, a chave a ser removida é encontrada no nó folha ao qual pertence e é então removida do mesmo. Todas as entradas à direita daquela deletada são deslocadas para a esquerda.

Caso o nó folha fique com um número inferior ao aceitável de entradas, ele deverá ser combinado com outro nó folha ou as entradas de algum nó folha deverão ser redistribuídas.

1. Se é possível combinar o nó com o vizinho à esquerda, a combinação é feita e a entrada do nó ai que apontava para o valor removido também deve ser removida junta do ponteiro que apontava para o valor
    a. Se a remoção não deixa o nó pai mais vazio do que a tolerância, o algoritmo termina
    b. Se a remoção deixa o nó pai mais vazio do que a tolerância, o mesmo é combinado com um vizinho do mesmo nível (um nó irmão), se possível, ou os ponteiros são redistribuídos de forma a manter o mínimo de $\lceil n/2 \rceil$ ponteiros em cada nó.
2. Se não é possível fazer a combinação, os valores dos nós folha são redistribuídos de forma a manter um mínimo de $\lceil n/2 \rceil$ valores em cada nó folha. O valor no nó pai pode ter que ser atualizado conforme a nova configuração do nó folha.

A remoção vai sendo aplicada de forma recursiva até que a raiz seja alcançada, um nó pai não fica com menos ponteiros do que o mínimo após a remoção ou uma redistribuição é realizada.

Ao fazer uma redistribuição é importante sempre verificar se o nó pai continua obedecendo às regras da árvore B+. Caso não esteja, pode ser necessário alterar os valores recursivamente.

Observação: Chaves de busca não-únicas:

Alguns sistemas permitem que mais de um registro tenha a mesma chave. Isso pode acarretar em uma perda de performance, por exemplo, na remoção de registros, pois o valor a ser deletado pode ter que ser buscado sequencialmente entre todos aqueles que possuem a mesma chave.

Uma solução simples é fazer com que as chaves sejam registradas compostas com outro atributo de valor único, o que facilita na hora de buscá-las.

Outra solução mais espacialmente eficiente é guardar a chave não única apenas uma vez com uma lista de ponteiros para os registros associada à chave. Porém essa brodagem acrescenta mais complexidade à implementação e também tem problemas na hora da remoção, pois continua sendo necessário buscar sequencialmente todas as entradas da lista.