

Encontra-se presente neste guia de estudos, a matéria (resumida) necessária para o primeiro teste de Complementos de Base de Dados (CBD). Note-se que, os slides apresentados em aulas teóricas, incluem perto de 200 dispositivos.

Guia de Estudos para Complementos de Base de Dados – 2019/2020

Alexandre Coelho

Índice

1.	Metadados.....	3
	Acesso aos Metadados – MS SQL	3
	Sys Schema Views	3
	Information_Schema Views	3
	Information_Schema vs Sys Schema Views.....	3
	Information_Schema	3
	Sys Schema Views	4
	Bases de Dados de sistema.....	4
	MS SQL	4
	Information_Schema – Views dos Metadados	4
	Sys Schema – Views dos Metadados.....	5
	System Stored Procedures and Functions	5
	Stored Procedures de Sistema vs Consultas.....	6
	Stored Procedures – SP’s para acesso a Meta informação	6
2.	Storage	7
	Estrutura de Ficheiros	7
	Tipos de Armazenamento	7
	Discos Magnéticos	8
	RAID.....	9
	RAID 0	9
	RAID 5	10
	RAID 10 (1+0)	10
	Fatores de Decisão	10
	Recomendações RAID	11
	Variable-Lenght Records	11
	Organização dos Registos em Ficheiros	12
	Buffer Manager	14
	Files & Filegroups	15
3.	Índices.....	19
	Definições	19
	Índices Ordenados	20
	Índices “densos”	20
	Índices “esparsos”	20
	Índices Secundários	21

Índices Primários e Secundários	22
Índices B+ - Tree	23
Estrutura dos nós.....	24
Estrutura dos nós folha.....	24
Índices B-Tree	27
Hashing – estático.....	28
Bucket overflow.....	31
Índices <i>Hash</i>	31
<i>Hash</i> Dinâmico.....	32
<i>Hash</i> extensível vs outros métodos	33
4. Processamento de Queries.....	33
Parser	34
Planos de Execução	34
Plano Físico.....	38
5. Índices – MS SQL	39
Tipos de Índices MS SQL	39
Cover vs Composite Index	39
Filtered index	39

1. Metadados

As bases de dados têm de manter um conjunto de dados sobre os dados. Esta informação está persistida no que se costuma designar o **Catálogo** ou **Dicionário**.

Acesso aos Metadados – MS SQL

Sys Schema Views

- Uma das formas (preferencial = desempenho) de aceder aos Metadados;
- Por serem *views*, suportam a independência de eventuais alterações “físicas” às tabelas de sistema;
- Quer as *views*, quer as colunas, são auto-descritivas, de forma a aprender a informação relativa aos Metadados solicitados.

Information_Schema Views

- Seguem as definições standard ISO relativas às vistas sobre o catálogo;
- Apresentam a informação dos Metadados em formato independente de qualquer implementação das tabelas do catálogo;
- As aplicações quando usam esta coleção de *views* são tendencialmente mais portáteis entre diferentes SGBDs.

Information_Schema vs Sys Schema Views

Information_Schema

- “names friendly” (Vantagem);
- Joins através de names (Vantagem);
- Standard/potencialmente mais interoperável (Vantagem);
- Informação mais limitada (Desvantagem);
- Desempenho pode ser inferior (Desvantagem).

Sys Schema Views

- Melhor desempenho (Vantagem);
- Informação mais pormenorizada (Vantagem);
- Orientada a objetos (Característica);
- Joins por objectID (Característica);
- Menos inteligível (Desvantagem);
- Proprietário (Desvantagem).

Bases de Dados de sistema

MS SQL

- Contem a BD: **master**:
 - Armazena informações sobre as bases de dados e seus objetos existentes no SGBD;
 - Muito importante pois preserva (meta) informação sobre use DBs (Ex: logins):
 - Contudo há que assegurar que não se inscreva diretamente objetos sobre esta (USE 'uBD').
- Objetos de uma BD:
 - Tabelas que suportam os registos;
 - Tipos de dados (de sistema e definidos pelo utilizador);
 - Constraints;
 - Índices;
 - Views;
 - Stored Procedures;
 - Funções;
 - Triggers;
 - ...

Information_Schema – Views dos Metadados

- Cada view do **Information_Schema** contém meta informação sobre os objetos armazenados numa base de dados;

INFORMATION_SCHEMA (Database metadata)	
CHECK_CONSTRAINTS	one row for each CHECK constraint.
COLUMNS	one row for each column.
KEY_COLUMN_USAGE	one row for each column that is constrained as a key.
REFERENTIAL_CONSTRAINTS	one row for each FOREIGN KEY constraint.
TABLE_CONSTRAINTS	one row for each table constraint.
TABLES	one row for each table in the current database.
VIEW_COLUMN_USAGE	one row for each column that is used in a view definition.
VIEW_TABLE_USAGE	one row for each table that is used in a view.
VIEWS	one row for each view.
COLUMN_DOMAIN_USAGE	one row for each column that has an alias data type.
COLUMN_PRIVILEGES	one row for each column that has a privilege that is either granted to or granted.
CONSTRAINT_COLUMN_USAGE	one row for each column that has a constraint defined on the column.
CONSTRAINT_TABLE_USAGE	one row for each table that has a constraint defined on the table.
DOMAIN_CONSTRAINTS	one row for each alias data type that has a rule bound to it.
DOMAINS	one row for each alias data type.
PARAMETERS	one row for each parameter of a user-defined function or stored procedure.
ROUTINES	one row for each stored procedure and function.
ROUTINE_COLUMNS	one row for each column returned by the table-valued functions.
SCHEMATA	one row for each schema in the current database.
TABLE_PRIVILEGES	one row for each table privilege that is granted to or granted by the current user.

Sys Schema – Views dos Metadados

- Cada view do **Sys Schema** contém meta informação sobre os objetos armazenados numa base de dados

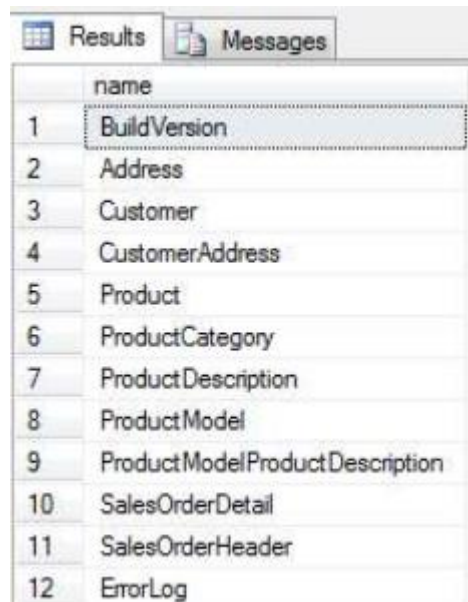
```
select s.name as 'SchemaName', o.name as 'TableName'
from sys.schemas as s
    inner join sys.all_objects as o
    on s.schema_id = o.schema_id
where s.name='sys' and
    o.type = 'V'
order by SchemaName, TableName;
```

System Stored Procedures and Functions

- Stored Procedures e functions de sistema retornam informação de catálogo;
- Tratam-se de sps e functions específicas do MS SQL
 - Contudo isoladas da implementação do catálogo subjacente.

Stored Procedures de Sistema vs Consultas

- Lista de tabelas de utilizador numa BD – “**Select name from sys.all_objects where type = ‘U’**”;
- Ou simplesmente, **exec sp_tables**.



	name
1	BuildVersion
2	Address
3	Customer
4	CustomerAddress
5	Product
6	ProductCategory
7	ProductDescription
8	ProductModel
9	ProductModelProductDescription
10	SalesOrderDetail
11	SalesOrderHeader
12	ErrorLog

Stored Procedures – SP’s para acesso a Meta informação

System stored procedures that implement data dictionary functions.	
sp_column_privileges	Returns column privilege information for a single table in the current environment.
sp_columns	Returns column information for the specified objects that can be queried in the current environment.
sp_fkeys	Returns logical foreign key information for the current environment.
sp_pkeys	Returns primary key information for a single table in the current environment.
sp_server_info	Returns a list of attribute names and matching values for SQL Server
sp_special_columns	Returns the optimal set of columns that uniquely identify a row in the table.
sp_sproc_columns	Returns column information for a single stored procedure or user-defined function in the current environment.
sp_statistics	Returns a list of all indexes and statistics on a specified table or indexed view.
sp_stored_procedures	Returns a list of stored procedures in the current environment.
sp_table_privileges	Returns a list of table permissions (such as INSERT, DELETE, UPDATE, SELECT, REFERENCES) for the specified table or tables.
sp_tables	Returns a list of objects that can be queried in the current environment. This means any table or view, except synonym objects.

2. Storage

Estrutura de Ficheiros

Uma BD é mapeada num conjunto de ficheiros persistidos em disco sobre o filesystem do sistema operativo.

Um ficheiro:

- Constituído por conjunto de blocos (blocks/pages):
 - E.g. 4 a 8 KB (mas pode ser redefinido).
- Um bloco conterá vários registos (em aplicações específicas um registo poderá estender-se por vários blocos, e.g. imagem).
- Registos nos ficheiros:
 - Fixed-Length records;
 - Variable-length records.

Tipos de Armazenamento

Características:

- Capacidade (e disponibilidade);
- Velocidade de Acesso;
- Preço (por unidade de dados);
- Fiabilidade.

Tipicamente, temos os níveis:

- **Cache:** rápida, de reduzida capacidade, gerida pelo sistema operativo;
- **Memória RAM:** para dados em operação, capacidade média/baixa, muito volátil;
- **Flash/SSD:** acesso rápido, capacidade média alta, não voláteis, caros melhora desempenho num nível adicional de cache;
- **Discos magnéticos:** Na grande maioria ainda os mais utilizados (devido ao custo) grande capacidade, performance qb (dependendo de outros fatores de otimização).
 - Suporta a persistência permanente das alterações à base de dados;
 - Embora suscetíveis de falha existem precauções possíveis.

Discos Magnéticos

Características:

- Ainda o modo mais comum/disseminado de persistência;
- Prato: duas faces;
- Sector: ~512 bytes;
- Pistas (Tracks): ~50000 to 100000 / prato
 - + Interiores : ~500 a 1000 sectores;
 - + Exteriores: ~1000 a 2000 sectores.
- Conjunto: de 1 a 5 pratos;
- Cilindro: Considerando o movimento solidário de todas as cabeças/braços!
- Só uma cabeça está ativa em cada instante;
- Controlador/Interfaces (velocidades): SATA, SATA II, SATA 3, SCSI,...

Medidas de Desempenho

- **Velocidade de rotação** (*Rotational Speed*): número de rotações por minuto;
- **Tempo de Busca** (*Seek Time*): tempo necessário para deslocar a cabeça de leitura para o cilindro pretendido;
- **Atraso de Rotação** (*Rotational Latency/average latency time*): tempo necessário para o disco na sua rotação, posicionar o sector pretendido, de forma a ser lido pela cabeça de leitura;
- **Track-To-Track Seeks**: tempo necessário para mover a cabeça de uma pista para a adjacente (factor relevante na leitura sequencial) [e.g. 0.6 ms (read); 0.9 ms (write)];
- **Average Seek Time**: tempo que em média as cabeças levam a pesquisar informação em pistas aleatórias (factor relevante na leitura aleatória) [e.g 5.2 ms (read); 6 ms (write)];
- **Access Time**: intervalo de tempo entre pedido de leitura/escrita e início da transferência de dados;
- **MTTF**: tempo médio entre falhas, medida de fiabilidade do disco.

Técnicas de recolha dos dados

- Persistidos em disco segundo a estrutura de blocos (blocks/pages)
 - Buffering;
 - Read-ahead;
 - Scheduling (e.g. elevator);
 - File system frag/defrag techniques;
 - Non-volátil memory (intermediary) buffers;
 - Log disk w/sequential read/write operations, pre definitive persistence

RAID

RAID – Redundant Array of Independent Disks

- Método de combinar vários discos rígidos de forma a aumentar a capacidade e performance, bem como, originar redundância de dados, prevenindo assim falhas do disco rígido.
- **Redundância = Fiabilidade** (e.g. mirroring/Shadowing)
 - Não será completamente eficiente;
 - Pode também não ser completamente eficaz! E se falharem os 2 (ou n – e.g. catástrofe natural)
 - ❖ Não pode haver a perfeita assunção de que as falhas dos discos são independentes.
- Paralelismo = melhoria de Performance (*striping* data, olha para o conjunto como se fosse um disco e blocos são escritos dispersos por vários discos).


RAID 0

- Usa a técnica de *data stripping* (segmentação de dados);
- Várias unidades de disco rígido são combinadas para formar um volume grande;
- Pode ler e gravar mais rapidamente do que uma configuração não-RAID, visto que segmenta os dados e acede a todos os discos em paralelo.

- 💾 Não permite redundância de dados;
- 💾 Necessita de pelo menos 2 unidades de disco rígido.

RAID 1


- Cria um espelho (*mirror*) do conteúdo de uma unidade, noutra unidade do mesmo tamanho;
- A réplica fornece integridade de dados otimizada e acesso imediato aos dados se uma de as unidades falhar.

 Requer um mínimo de duas unidades de disco rígido e deve consistir em um número par de unidades.

RAID 5

- Proporciona o equilíbrio entre redundância de dados e capacidade de disco;
- Segmenta todos os discos disponíveis num grande volume;
- O espaço equivalente a uma das unidades de disco rígido será utilizado para armazenar bits de paridade;
- Se uma unidade de disco rígido falhar, os dados serão recriados através dos bits de paridade.

RAID 10 (1+0)

- É a combinação de discos espelhados (RAID-1) com a segmentação de dados (*data stripping*) (RAID 0);
- Oferece as vantagens de transferência de dados rápida de segmentação de dados, e as características de acessibilidade dos arranjos espelhados;
-  A performance do sistema durante a reconstrução de um disco é também melhor que nos arranjos baseados em paridade.

Fatores de Decisão

- Custos de discos extra;
- Requisitos de performance;
- Requisitos de performance em caso de falha (operacional e de reconstrução);
- Implementação: Software vs Hardware (fiabilidade e desempenho).

Recomendações RAID

RAID 1

- Sistema Operativo;
- Transaction Log (devido à boa performance em escrita).

RAID 5

- Tabelas de Dados com acesso especialmente em modo de leitura – pois tem menos storage overhead, mas maior overhead de escrita (paridade);
- BDs temporárias;
- Backups.

RAID 10

- Tabelas de dados com acesso frequente em modo de Escrita;
- BDs temporárias (com requisitos de maior performance).

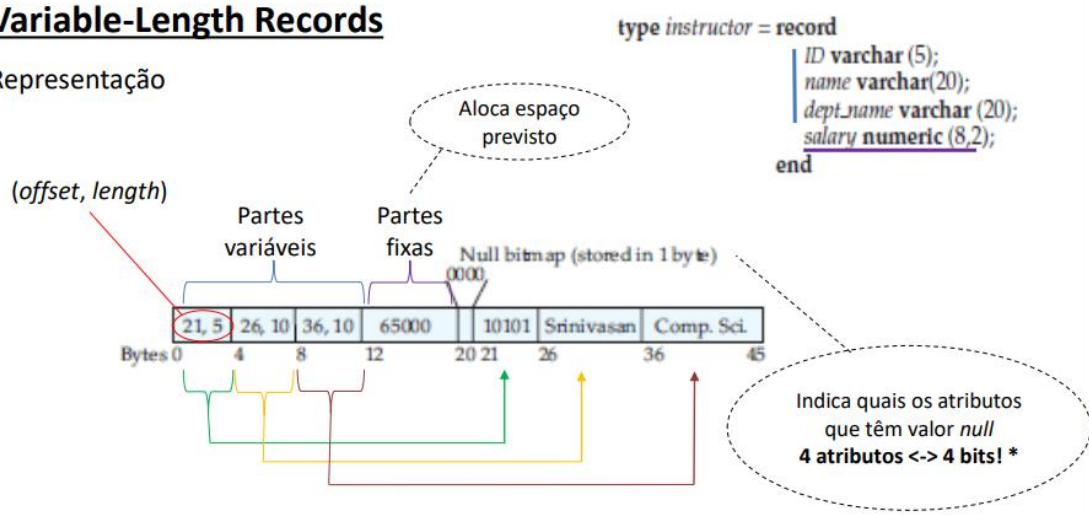
Fixed-Length Records

Variable-Lenght Records

- Armazenamento de múltiplos registos que:
 - Contenham campos de tipos com “comprimento variável”;
 - Ou, oriundos de “entidades tipo” diferente.
- Como representar estes registos? (e armazenar nos blocos?)
 - De forma a que os valores dos seus campos sejam acedidos/manipulados com facilidade

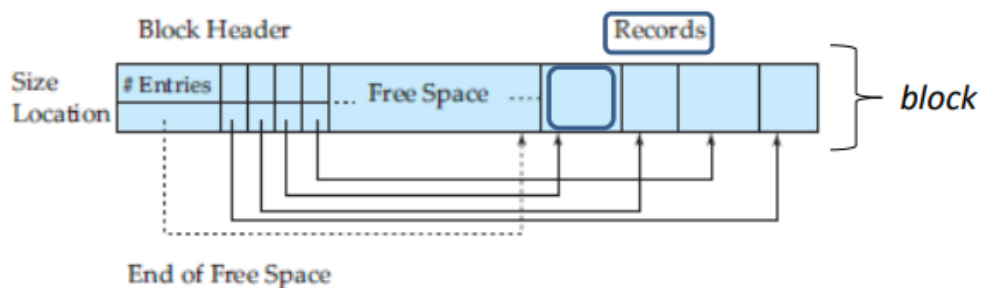
Variable-Length Records

Representação



Armazenamento

- Como armazenar então nos blocos registos de comprimento variável?
- Normalmente existe um *header* no início de cada block, com:
 - O número de registos persistidos;
 - A indicação do final do espaço livre no bloco;
 - Um array com a indicação da localização e tamanho de cada registo.



Organização dos Registos em Ficheiros

- Como se podem então organizar os registos pelos ficheiros?
- Alternativas:
 - 📁 **Heap file organization** – Um registo pode ser colocado em qualquer local do ficheiro em que exista o espaço. Não existe nenhuma ordenação. Tipicamente por cada “relação” (MR);
 - 📁 **Sequential file organization** – Os registos são guardados sequencialmente segundo a “chave de pesquisa”;
 - 📁 **Hashing file organization** – A localização é função do cálculo de uma função *hash* que mapeia o registo no seu bloco dentro do ficheiro.

Multi-table clustering

- Bases de dados simples e (expectavelmente) “pouco” populadas, têm tipicamente uma estrutura simplificada
 - Tuplos de uma relação são representados como fixed-length records;
 - E as relações podem estar mapeadas numa estrutura de ficheiros simples: 1 relação – 1 ficheiro.

- Em BDs mais exigentes (em performance e considerando dimensão e volume de dados)
 - O SGBD fará a gestão do(s) ficheiro(s);
 - Dos blocos nos ficheiros;
 - Dos registos a persistir nos blocos.
- Nos casos em que as múltiplas relações são guardadas no mesmo ficheiro
 - Existe por vezes a preocupação de que os blocos contenham registos de uma só relação.
- Contudo, pode justificar o acrescento de complexidade de juntar tuplos de relações diferentes num mesmo bloco;
- Consideremos:

```
Select dept_name, building, budget, ID, name, salary from
      department join instructor;
```

- Estes registos podem estar espalhados por diversos blocos forçando várias leituras de blocos diferentes

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

- Uma organização física dos registos das relações *department* e *instructor* que melhoraria a performance da anterior *join query*.

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

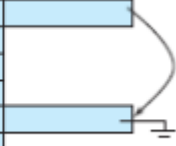
<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000



Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

- Contudo poderia haver agora *queries* (até mais simples) com pior performance e.g. “*Select * from department*”;
- Mesmo mantendo uma “lista” de departamentos através de apontadores,

Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	



- Pode requerer acesso a mais blocos, pois agora os blocos contendo também *instructors* conterão necessariamente menos departamentos
 - Dependerá dos dados;
 - Das queries pretendidas;
 - E da performance para aquelas que possam ser mais exigentes.

Buffer Manager

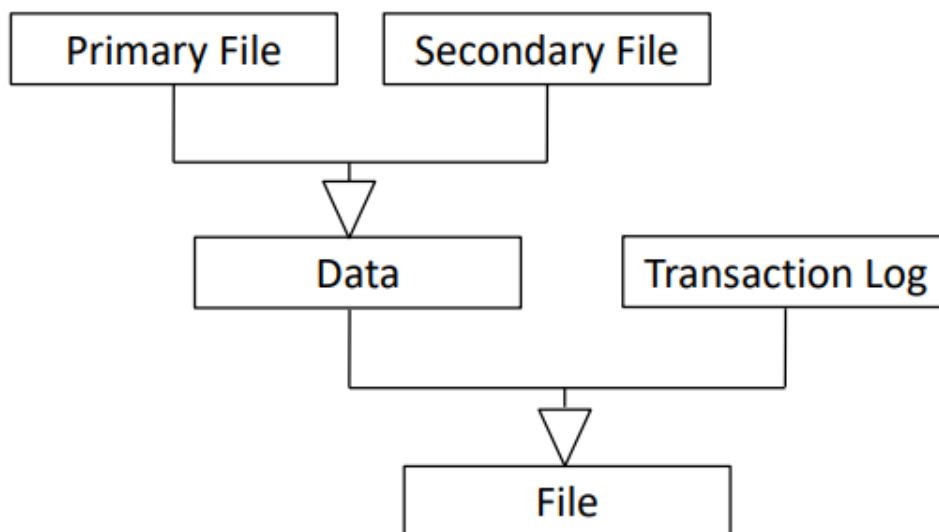
- Um dos objetivos de uma implementação de um SGBD é minimizar o número de transferências de blocos entre o disco e a memória.
 - Uma das formas é manter tantos *blocks* quanto possível em memória.
- O buffer é o local onde se mantêm cópias em memória dos dados (parcela) persistidos em disco;
- Buffer Management:
 - Sempre que há uma chamada query/execução ao SGBD
 - Se já existir em buffer o(s) bloco(s) respetivos este é (são) disponibilizado(s);
 - Se o bloco não está no buffer, este terá de alocar espaço para o recolher do disco;
 - Se necessário “livra-se” de algum(s) existente(s) (e.g. LRU, MRU)
 - Copiando-os de volta para o disco se a versão em memória for mais atualizada.
 - Este processo “interno” é transparente para o programa que solicita o SGBD.

Files & Filegroups

- As bases de dados são criadas tendo como suporte um conjunto de ficheiros específicos;
- Os ficheiros podem ser agrupados em *filegroups* de forma a facilitar a sua gestão, localização/persistência e performance no seu acesso;
- Uma base de dados tem que ser criada com pelo menos um ficheiro de dados e um ficheiro de log (exclusivos de cada base de dados).

Tipos de ficheiros (da base de dados):

- **Primário** (Primary data file);
- **Secundário** (Secondary data files);
- **Log de Transações** (Transaction log file).



Ficheiro Primário

- ✓ Contém informação de inicialização da base de dados
- ✓ Agrupa tabelas (de sistema):
 - geradas/atualizadas automaticamente aquando da criação de uma nova BD
 - contêm os utilizadores, objetos e permissões de acesso (tabelas, índices, views, triggers e stored procedures, ...)
- ✓ Contém referências para os restantes ficheiros da BD
- ✓ Dados do utilizador podem ser persistidos neste ou, num ficheiro secundário
 - Existe apenas e obrigatoriamente um ficheiro primário por base de dados
 - Neste poderão coexistir metadados e dados

➤ Extensão do ficheiro: .mdf

Exemplo:

```
CREATE DATABASE [AdventureWorksCBD]
    ON PRIMARY (
        NAME = [AdventureNewData],
        FILENAME = 'C:\ProjectoGrupo6\AdventureWorksCBD.mdf',
        SIZE = 3MB,
        FILEGROWTH = 1MB
    ),
```

Ficheiro(s) Secundário(s)

✓ São opcionais

– A base de dados pode não ter ficheiro secundário, se todos os dados estiverem armazenados num ficheiro primário

✓ São definidos pelo utilizador

✓ Podem armazenar tabelas/objetos que não tenham sido criados no ficheiro primário

✓ Algumas bases de dados beneficiam de múltiplos ficheiros secundários de forma a dispersar os dados por vários discos

➤ Permitem melhorar desempenho e escalabilidade, se/quando previsível que o Primary data file atinja o limite máximo de um ficheiro para o SO

➤ Extensão dos ficheiros: .ndf

Exemplo:

```
FILEGROUP [Products_AllItems] (
    NAME = [ProductDescription],
    FILENAME = 'C:\ProjectoGrupo6\ProductDescription.ndf',
    SIZE = 0,608,
    MAXSIZE = 0,8512,
    FILEGROWTH = 0,04864,
),
```

Log de Transações

Transaction Log File:

- ✓ Utilizado para armazenar o “rasto” de transações
- ✓ Persiste a informação necessária à recuperação da base de dados
- ✓ Todas as bases de dados têm pelo menos um log file
- Extensão dos ficheiros: .ldf
- ❖ Embora em sistemas simples a aproximação default é manter os data e transaction files na mesma path;
 - Em casos mais complexos/exigentes é recomendado que os ficheiros de dados e de log de transações sejam colocados em discos separados.

Exemplo:

```
LOG ON (  
    NAME = [AdventureNewData_LOG],  
    FILENAME = 'C:\ProjectoGrupo6\AdventureWorksCBD.ldf',  
    SIZE = 2MB,  
    FILEGROWTH = 10%  
)
```

Filegroups

- Permitem aumentar a performance da base de dados ao facilitarem a distribuição dos ficheiros de dados que os constituem, por vários discos ou sistemas RAID.
- Tipos de Filegroups
 - **Primário** (Primary filegroup)
 - **Utilizador** (User-defined filegroup)
 - **Por Omissão** (Default filegroup)
- Cada BD conterá ***sempre um Primary filegroup***
 - Este contem Primary and Secondary data files (alocados a este grupo!)
- O utilizador pode criar Filegroups para agregar data files com objetivos p.e.:
 - Gestão administrativa, alocação de espaço e localização específica

Primário:

- Contém o **ficheiro primário** e **todos os ficheiros** que **não** tenham sido explicitamente colocados noutra filegroup;
- Todas as **tabelas de sistema** estão no **Primary filegroup**.

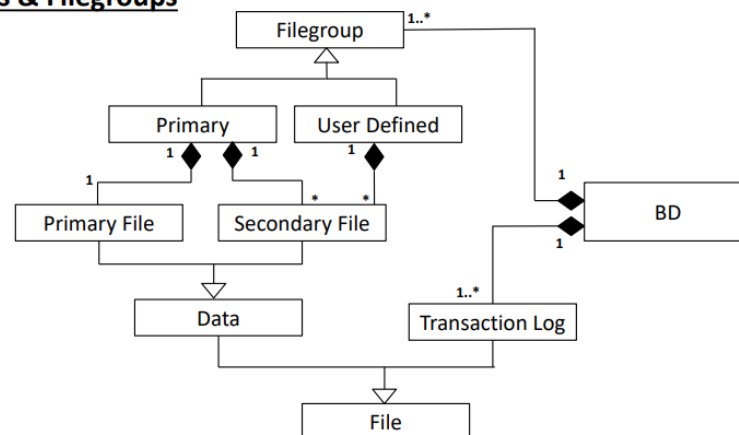
Utilizador:

- Inclui **qualquer filegroup definido pelo utilizador** durante o **processo de criação** (ou alteração) da **base de dados**;
- As tabelas e índices podem ser criadas e localizadas num filegroup específico, definido pelo utilizador.

Por Omissão:

- Armazena as páginas das tabelas e índices para os quais não tenha sido atribuído um filegroup específico
- Apenas um filegroup pode ser, num dado momento, filegroup por omissão
- Se não tiver sido especificado filegroup por omissão, é considerado como tal o filegroup primário
- Utilizadores com role db_owner podem alterar o filegroup por omissão.

Files & Filegroups



- Os data files Data1.ndf, Data2.ndf e Data3.ndf podem ser criados respetivamente em 3 discos diferentes e atribuídos a 1 filegroup e.g. fgroup1.
- Queries aos dados das tabelas persistidas nestes ficheiros terão eventualmente um melhor desempenho dado que solicitarão informação distribuídas pelos 3 discos
 - Um esquema semelhante pode ser obtido com um único ficheiro, mas criado sobre RAID (Redundant Array of Independent Disks).

3. Índices

Definições

- **Índices:**

Estruturas para aumentar o desempenho no acesso à informação.

- Exemplo: Catálogo de autores numa livraria.

➤ Fornecem um caminho de acesso aos registos.

- **Um ficheiro de indexação** é constituído por registos, no seguinte formato:

chave de pesquisa	apontador
----------------------	-----------

- *Chave de Pesquisa*: atributo ou conjunto de atributos usado para “procurar” os registos num ficheiro.

- Os ficheiros de indexação são de menor dimensão que os ficheiros de dados originais;
- A existência de índices não modifica as relações nem a semântica das consultas.

📁 Tipos de índices:

- **Índices ordenados** – as chaves de pesquisa são armazenadas por uma certa ordem.
- **Índices Hash** – as chaves de pesquisa são distribuídas uniformemente por “buckets” através de uma função de Hash.

Índice Primário, Clustered e Secundário

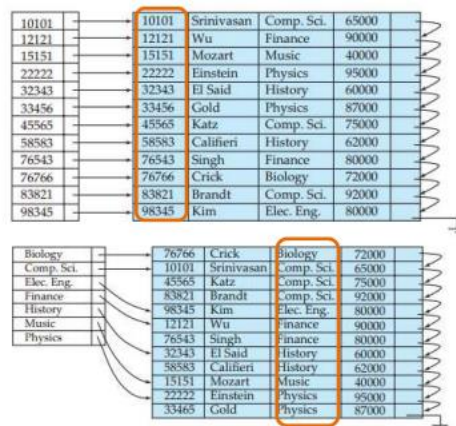
As entradas de indexação são ordenadas pela chave de pesquisa (e.g. autores no catálogo de uma livraria)

- **Índice Primário** – a ordenação coincide com a ordenação do ficheiro de dados, pelo campo que é também chave primária;
- **Índice Clustered** – é o índice sobre a chave de pesquisa que especifica a ordem sequencial do ficheiro de dados, contudo esse campo não é chave primária;
- **Índice secundário** (nonclustered index) – índice em que a chave de pesquisa especifica uma ordem diferente da ordem sequencial do ficheiro.

Índices Ordenados

Índices “densos”

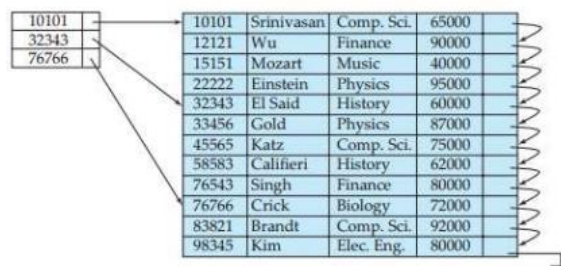
- A cada valor da chave de pesquisa corresponde um registo de índice
 - Se houver mais do que um registo com o mesmo valor apontado (caso possível se: chave de pesquisa <> chave primária) então o registo de índice apontará para o primeiro registo do valor apontado e os seguintes registos do mesmo valor serão subsequentes.
 - Está subentendida a ordenação do ficheiro de registos por ordem da chave de pesquisa
- Se for um **dense nonclustering index**, então cada entrada de registo de índice terá de guardar todos os ponteiros para todas as ocorrências do valor apontado.



Índices “esparcos”

Contém registos de índice, apenas para alguns dos valores da chave de pesquisa.

- Aplicável quando os registos estão ordenados sequencialmente pela chave de pesquisa;
- Para localizar um registo com o valor K da chave de pesquisa:
 - Localizar o registo de índice com o maior valor < K, da chave de pesquisa;
 - Pesquisar sequencialmente o ficheiro a partir do registo apontado pelo registo de índice.
- Ocupa menos espaço,
- Menor “overhead” nas operações de *insert* e *delete*;
- Normalmente mais lento da procura de registos.



Avaliação das diferentes técnicas de indexação

- Nenhuma é absolutamente melhor que outra em todos os aspetos;
- Há que considerar:
 - Tipos de acessos à informação:
 - Registos com um atributo, igual a um valor específico;
 - Registos com um atributo, com valor num determinado intervalo de valores.
 - Tempo de acesso;
 - Tempo para *insert*;
 - Tempo para *delete*;
 - Espaço adicional.
- Dica: uma entrada num índice esparsa por cada bloco
 - O custo de processamento está normalmente associado a encontrar e trazer o bloco do ficheiro do disco, uma vez no bloco o tempo de procurar dentro do mesmo é negligenciável.

Índices multinível

- Se o índice primário não “cabe” em memória o acesso torna-se dispendioso.
- De modo a diminuir o número de acessos a disco, tratar o índice primário como um ficheiro sequencial e criar um índice esparsa sobre o índice primário:
 - **Índice interno** – O ficheiro sequencial do índice primário;
 - **Índice exterior** – Índice esparsa do índice primário.
- Se o índice exterior não “couber” em memória, criar um novo nível de índice -> mais um nível;
- Todos os níveis dos índices devem ser atualizados nas operações de *insert*, *update* e *delete* !! (Update: pode ser abordado como um delete + insert).

Índices Secundários

Motivação

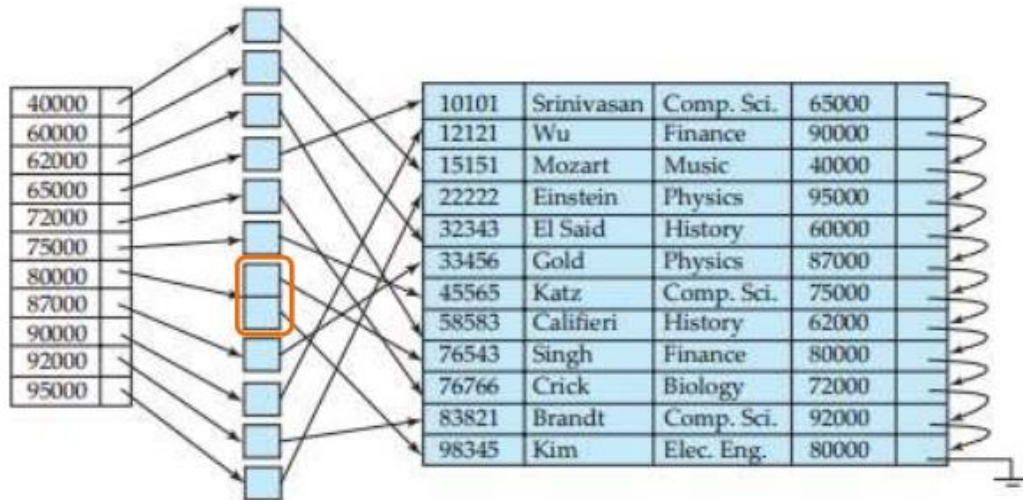
- Frequentemente, é necessário pesquisar registos por certos atributos, que não são o atributo da chave de pesquisa do índice primário ou *Clustered*.
- Exemplo – Na base de dados das contas bancárias, e os dados estão ordenados sequencialmente pelo número de conta.
 - Poderei pretender:
 - Obter todas as contas numa determinada agência;
 - Obter as contas em que o saldo está num determinado intervalo de valores.
- Devo então criar índices secundários, para outras chaves de pesquisa.

Exemplo:

Índice secundário pelo saldo de conta

- Os índices secundário terão de ser densos!

Os índices secundário terão de ser densos!



Índices Primários e Secundários

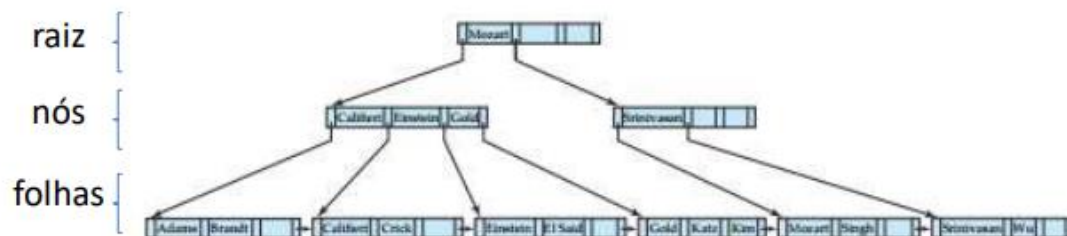
Considerações

- Os índices secundários têm de ser densos;
- Os índices aumentam o desempenho na consulta de registos;
- Quando a informação é modificada, cada ficheiro de índice também tem de ser atualizado
 - Introduce acréscimo de processamento quando existe modificação da informação.
- O varrimento sequencial sobre o índice primário é eficiente (os dados estão armazenados sequencialmente, na mesma ordem do índice);
- O varrimento sequencial sobre índices secundários é dispendioso
 - Cada acesso a um registo pode implicar o acesso a um novo bloco de disco.

Índices B+ - Tree

Características

- *Balanced Tree* como alternativa aos índices sequenciais:
- Desvantagem dos índices sequenciais:
 - O desempenho diminui à medida que a dimensão do ficheiro aumenta;
 - É necessária reorganização periódica do ficheiro.
- **Vantagem** dos índices B+- Tree:
 - Reorganização automática em alterações pequenas e locais, resultantes das operações de *insert* e *delete*;
 - Não é necessária a reorganização total frequente do ficheiro de modo a manter o desempenho.
- **Desvantagem** dos índices B+-Tree:
 - Acréscimo de processamento em (alguns) insert e delete;
 - Acréscimo de espaço ocupado.
- As vantagens sobrepõem-se às desvantagens, por isso são comumente utilizados



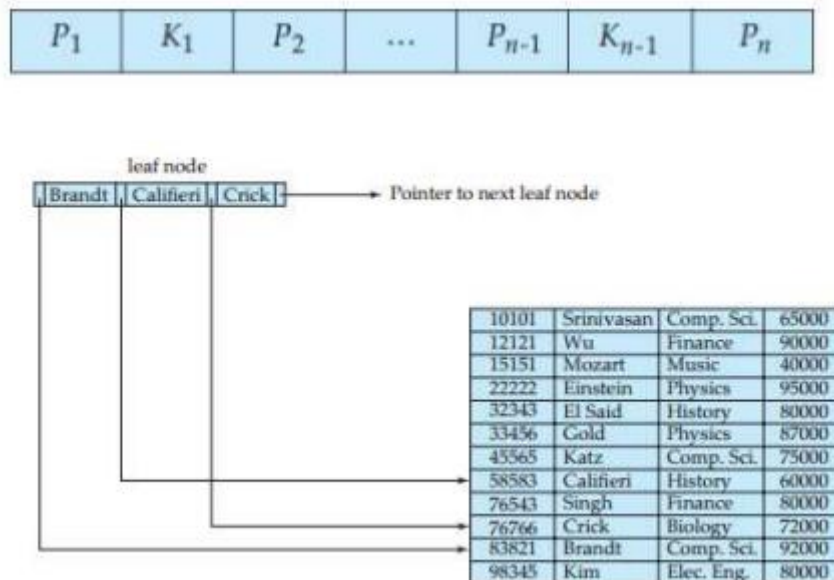
- Normalmente índices densos;
- Organização em árvore que têm as seguintes propriedades:
 - Todos os caminhos da raiz às folhas têm o mesmo comprimento;
 - Cada nó que não é raiz ou folha tem entre $\lceil n/2 \rceil$ e n filhos
 - Cada nó folha tem entre $\lceil (n-1)/2 \rceil$ e $n-1$ valores;
 - Casos especiais:
 - Se a raiz não é folha tem pelo menos 2 filhos;
 - Se a raiz é folha, pode ter entre 0 e $(n-1)$ valores

Estrutura dos nós



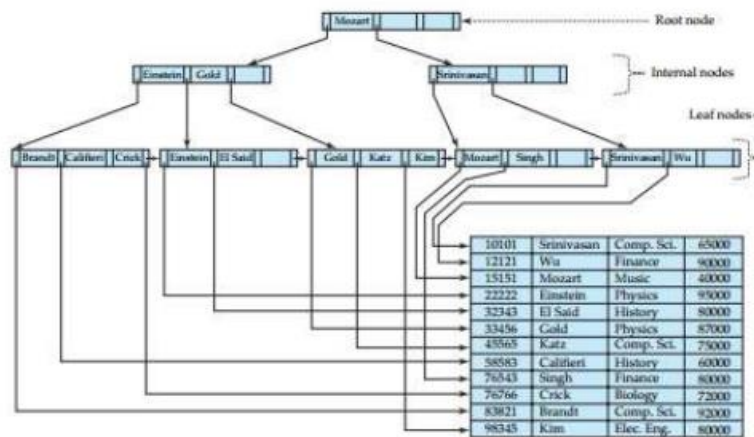
- K_i são os valores da chave de pesquisa;
- P_i são os apontadores para os filhos (para nós que não são folha) ou, apontadores para registos ou *buckets* de registos (para os nós folha);
- Os valores da chave de pesquisa estão ordenados no nó $K_1 < K_2 < K_3 < \dots < K_{n-1}$.

Estrutura dos nós folha



- O apontador P_i , aponta para um registo no ficheiro com o valor da chave de pesquisa K_i , ($i = 1, 2, \dots, n-1$)
- P_n aponta para a próxima folha ordenada pela chave de pesquisa.
- Constituem um índice esparsos multinível dos nós folha;
- Para um nó não folha com m ($< n$) apontadores:
 - A subárvore apontada por P_1 conterá chaves de pesquisa de valor $\leq K_1$;
- Para $2 \leq i \leq n-1$,
 - Todos os valores da chave de pesquisa da subárvore para a qual P_i aponta são maiores ou iguais a K_{i-1} e menores que K_{i+1} ;
- Contém até n apontadores, e pelo menos $\lceil n/2 \rceil$;
- O número de apontadores num nó é denominado de *fanout* do nó.

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-----	-----------	-----------	-------



Altura da árvore e Espaço necessário

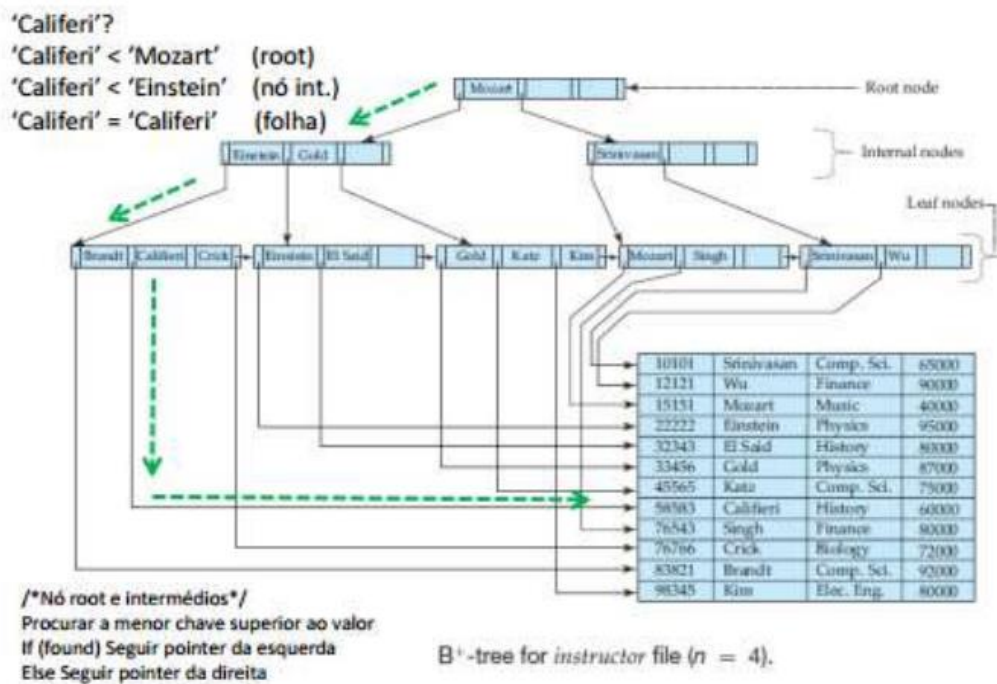
- Consideremos uma entrada no índice: 12 (chave pesquisa) + 8 (apontador) = 20 bytes;
- Com páginas/blocos de 8KB;
- Com cerca de 400 entradas por página,
 - Assumindo páginas semipreenchidas contendo cerca de 250 entradas;

Nível	N.º de chaves	Tamanho (bytes)
1	250	8 K
2	$250^2 = 62\ 500$	2 MB
3	$250^3 = 15\ 625\ 000$	500 MB

- Mesmo contendo 15 milhões de chaves, mantendo a raiz em memória (8K) é possível encontrar qualquer chave com um máximo de 2 acessos ao disco;
- Atualmente dada a memória disponível é possível com 1 ou mesmo nenhum acesso ao disco.

Consultas

- Obter os registos com o valor K da chave de pesquisa
 - Início no nó raiz
 - Examinar o nó para o valor mais pequeno da chave de pesquisa $> K$;
 - Se o valor existe, assumir que é K_i . Seguir para o nó filho apontado por P_i ;
 - Senão com $K \geq K_{m-1}$, seguir para o nó apontado por P_m ;
 - Se o nó “seguido” pelo apontador do passo anterior não é folha, repetir o procedimento anterior;
 - Quando o nó folha, para a chave i , $K_i = K$, seguir o apontador P_i para o registo ou *bucket*. Senão não existem registos com o valor K.



Características

✓ Vantagens:

- Menor número de nós;
- Por vezes não é necessário percorrer a árvore até às folhas para obter o valor da chave de pesquisa.

✗ Desvantagens:

- Só uma pequena fração de todos os valores da chave de pesquisa são obtidos mais “cedo”;
- Os nós não folha têm maior dimensão;
- As operações de *insert* e *delete* têm processamento mais complicado.
- Tipicamente as vantagens não se sobrepõem às desvantagens.

Hashing – estático

Definições

- Um *bucket* é uma unidade de armazenamento, que contém um ou mais registos (tipicamente um *bucket* corresponde a um bloco de disco);
- Num ficheiro com organização *hash*, o acesso direto a um bucket, através do valor da chave de pesquisa, é obtido utilizando uma função de *hash*;
- Com K o conjunto de valores da chave de busca e B o conjunto de todos os buckets,
 - **H é uma função de hash de K para B**
- A função de *hash*, é utilizada para aceder, inserir e remover registos;
- Registos com diferentes valores de pesquisa, podem estar associados ao mesmo *bucket*;
 - Assim o bucket tem de ser “varrido” sequencialmente para localizar o registo.

Exemplo de Ficheiro com organização *hash*

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califèri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98045	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

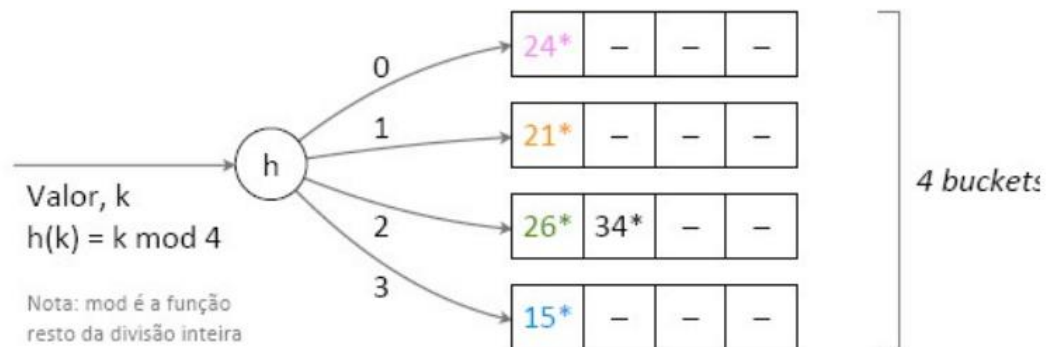
bucket 7

Funções de Hash

- No pior caso, a função hash mapearia todos os valores da chave de pesquisa, para o mesmo bucket;
 - Assim o tempo de acesso é proporcional ao número de valores da chave de pesquisa existentes;
- A função *hash* (dispersão) deve cumprir os seguintes requisitos:
 - Uma função **uniforme**, ou seja, cada *bucket* é atribuído o mesmo número de valores da chave de pesquisa (de todo o intervalo de valores possíveis);
 - Uma função **aleatória**, em que cada *bucket* contém, a cada momento, em média, aproximadamente o mesmo número de valores da chave de pesquisa.
- Tipicamente as funções calculam e utilizam a representação binária dos valores da chave de busca (por exemplo a representação binária dos caracteres).

Funções Hash (o princípio)

- Função h dispersa valores indexados por vários buckets
 - Cada valor indexado fica num só bucket



Exemplos

Organização *hash* do ficheiro de contas bancárias, utilizando o nome da agência como chave de pesquisa:

- Assumir 26 *buckets*, em que a função de *hash*, associa o valor binário i do primeiro carácter do nome, com o *bucket* i ;
- Não se obtém uma distribuição uniforme, pois é de esperar mais nomes a começar com a letra A do que com a letra X.

Organização *hash* do ficheiro de contas bancárias, utilizando o saldo como chave de busca:

- Assumir valor mínimo 1 e máximo 100.000;
 - Assumir 10 *buckets*, com o intervalos de 1-10.000, 10.001-20.000,...
 - A distribuição é uniforme pois cada *bucket* está associado com 10.000 valores da chave.
- Não é aleatória pois é de esperar que existam mais valores entre 1-10.000 do que em 90.001 e 100.000.

Bucket overflow

Pode ocorrer por:

- Número de *buckets* insuficiente;
- Alguns *buckets* têm mais valores que outros, denominado como *bucket skew*:
 - Múltiplos registos com o mesmo valor da chave de pesquisa;
 - A função de *hash* escolhida, resulta numa distribuição não uniforme dos valores da chave de pesquisa.
- Solução:
 - Cadeias de *overflow* – Os *buckets de overflow* são associados através de uma lista. É denominado *closed hashing*.

Índices Hash

Definições

- Um índice *hash*, organiza os valores da chave de pesquisa, com os respetivos apontadores para os registos, num ficheiro com uma estrutura de *hash*.
- Índices *hash*, são sempre índices secundários
 - **Porquê?**
- Eficientes em consultas de igualdades
 - **Porquê?**
- A função de *hash* associa os valores na chave a um conjunto fixo de *buckets*:
 - Se o número de *buckets* é reduzido, o aumento da BD, provoca *bucket overflow*, logo diminuição do desempenho;
 - Se o número de *buckets* é elevado (prevendo o aumento da BD), temos desperdício de espaço inicialmente;
 - Se a BD diminui, temos desperdício de espaço (devido ao *bucket overflow*)
 - Pode-se optar por uma re-organização periódica, mas é muito dispendioso
- Solução: **Hash dinâmico.**

Hash Dinâmico

Características

- Eficiente para bases de dados que aumentam e diminuem de dimensão;
- Permite a modificação dinâmica da função de hash
 - Em função do tamanho dos dados (embora processo condicione temporariamente acesso a alguns blocos).
- *Hash* extensível (uma forma de hash dinâmico)
 - A função de *hash* gera valores dentro de um intervalo alargado
 - Tipicamente inteiros de 32 bits;
 - Utiliza a cada momento um prefixo para endereçar um conjunto de *buckets*;
 - Sendo i o comprimento do prefixo e $0 \leq i \leq 32$;
 - Número máximo de buckets: 2^{32}
 - O valor de i varia com a dimensão da base de dados.

Exemplo:

<u>dept_name</u>	<u>$h(\text{dept_name})$</u>
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Hash function for *dept_name*.

search key = dept_name -> 32-bit hash values

Hash extensível vs outros métodos

✓ Vantagens

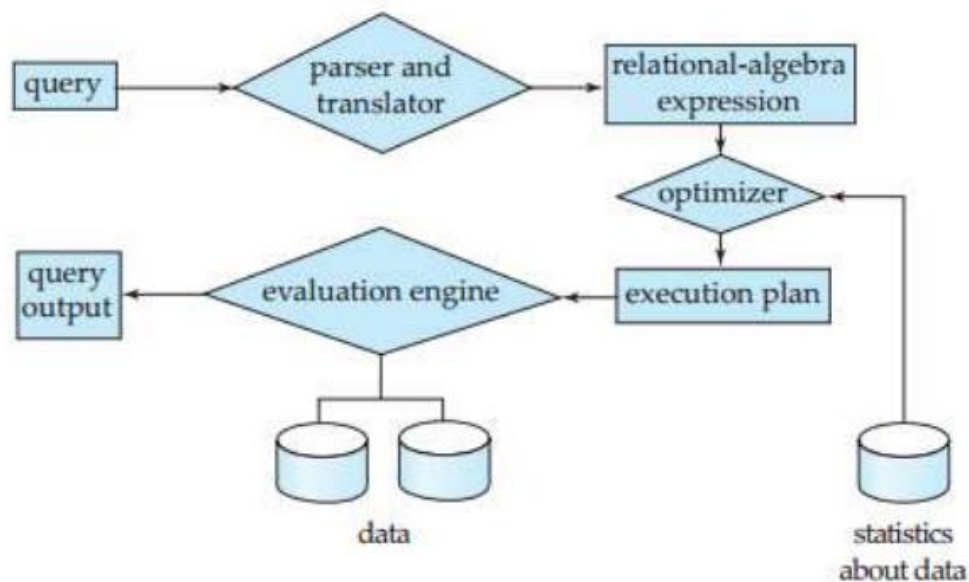
- O desempenho não se degrada com o aumento da dimensão do ficheiro;
- *Overhead* de espaço mínimo.

✗ Desvantagens

- Nível extra para obter o registo procurado;
 - A tabela de endereços de *buckets*, pode tornar-se muito grande (não caber em memória);
 - Alterar a dimensão da tabela de endereços de *buckets*, é uma operação dispendiosa.
- Mais adequado para critérios de pesquisa baseados em igualdades.

4. Processamento de Queries

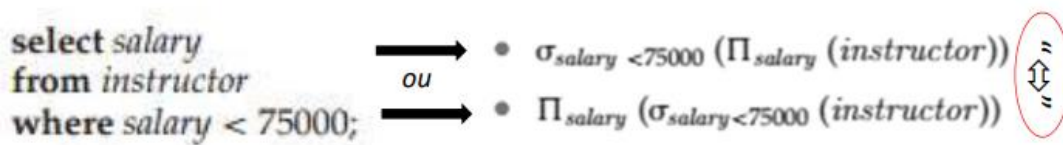
Fluxo



Parser

- Para o início do processamento “efetivo” o SGBD tem de traduzir/representar a query para uma forma “acionável” ao nível de “sistema”
 - O SQL é para humanos!
 - O formato suporta-se na *Álgebra Relacional* e suas equivalências.
- O **parser** realizar duas tarefas principais
 - A validação sintática (garantido que a *query* está “conforme” com as relações da BD a que se refere – utilizando o catálogo);
 - Traduz a *query* para uma árvore que representa as “suboperações” que terão de ser realizadas -> **Plano de Execução (ou Avaliação)**

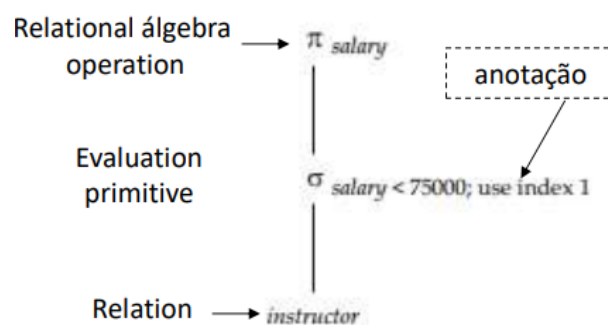
Exemplo:



Planos de Execução

- Diferentes planos de execução têm associados diferentes “**custos de execução**”;
- Não é da responsabilidade de quem elabora a *query*, estruturá-la da forma que reduza o seu “custo de execução” (bom pelo menos no sentido deste contexto ...);
- Será o *query optimizer* que gerará planos alternativos de execução e avaliará para cada o seu custo de execução afim de selecionar o que efetivamente será executado, considerando:

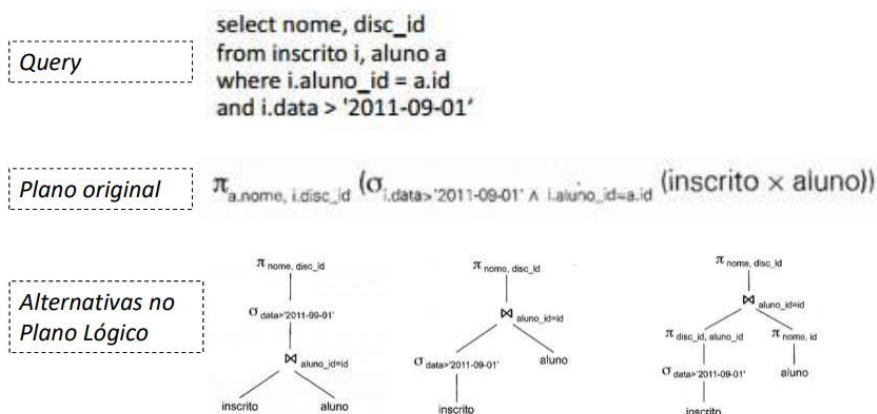
- As propriedades (e custos) dos operadores da álgebra relacional;
- Heurísticas.



A Geração das Alternativas

- A transformação da consulta inicial em alternativas na forma canónica **SPJ** (seleção-projeção-junção) dos operadores da álgebra relaciona, envolve:
 - Formar uma lista dos produtos das relações;
 - Aplicar a essa lista o predicado da cláusula WHERE;
 - Aplicar os agrupamentos;
 - Aplicar a essa lista o predicado da cláusula HAVING;
 - Aplicar as projeções;
 - Aplicar as ordenações.
- As regras de equivalência a álgebra relacional podem permitir grandes reduções no número de tuplos a processar (redução dos tuplos/dados nos processamentos intermédios);
- Também as heurísticas permitem ganhos de performance na execução, e.g.
 - Reorganizar as seleções para que as mais restritivas sejam executadas primeiro (o que reduz o número de tuplos a passar aos operadores ascendentes);
 - Descer projeções o mais possível (reduz o tamanho dos tuplos).

Exemplo:



Avaliação – Estimação do Custo

- A avaliação de alternativas lógicas de planos de execução poderá ponderar diversos fatores, incluindo:
 - Acessos ao disco;
 - Tempo de CPU;
 - E no caso de BDs distribuídas o custo da comunicação.
- Deverão ser considerados os custos (físicos) associados a cada “sub-operação” e combinados num custo total da *query*;
- Tipicamente a grande fatia do custo a ser ponderada (e mais “facilmente” estimada) prende-se com as necessidades das operações relativamente ao acesso ao disco.

Avaliação – Estimação do Custo II

- Contudo as estimativas são de fiabilidade questionável porque:
 - A resposta dependerá dos conteúdos e ocupação do *memory buffer*;
 - Com vários discos dependerá de como os dados estão distribuídos
 - E.g. um plano A pode precisar de mais leituras que B e ainda assim ser mais rápido na execução dependendo da distribuição dos dados pelos discos!
- Na prática, é **impossível determinar um custo exato**, da mesma forma que **não é possível gerar em tempo útil todas as alternativas possíveis de processamento de uma dada consulta**;
- O custo calculado pelo SGBD não é o custo real de execução, mas um valor que permite comparar planos alternativos entre si:
 - Alguns SGBD comparam efetivamente o custo calculado com o obtido para atualizar algoritmos de estimativas.

Avaliação – Estimação de Custo III

- Outras operações que não a seleção têm cálculos mais complexos!
 - Joins, sorts,...
- Ainda, terá de haver lugar a avaliação de expressões para lá de operações individuais (i.e. de toda árvore e seu contexto de execução), considerando a modalidade:
 - *Materialization*: persistência de resultados temporários (relações) em disco se excedem buffer
 - Há que juntar ao somatório de custo das operações mais estas!
 - *Pipelining* : Modalidade que vai disponibilizando às operações ascendentes resultados à medida que se disponha destes (por oposição a somar o processamento de cada operador em tempo integral).

Estatísticas sobre os custos

- As estatísticas são muitas vezes calculadas com base numa amostra dos dados
 - E.g. umas dezenas ou centenas de milhares de tuplos numa relação de milhões;
 - Deve ser uma amostra aleatória (evitar enviesamentos).
- As estatísticas não estão sempre a ser atualizadas!
- A informação estatística sobre as relações pode ser atualizada com periodicidade controlada pelo DBA
 - Comandos para atualizar estatísticas da BD: UPDATE STATISTICS; ANALYZE.
- As atualizações podem ser despoletadas “manualmente” ou periodicamente e/ou associadas a eventos (e.g. threshold de NR, disparity between estimates actual query execution performances/costs).

Plano Físico

- O plano físico (Execution Plan) contém o acoplamento entre os operadores relacionais e os operadores físicos escolhidos para os implementar;
- Um plano físico especifica como a consulta vai ser executada, sendo que a sua construção parte do plano lógico anterior, adicionando os operadores físicos mais adequados a cada operação lógica, juntamente com os respectivos custos associados;
- Geralmente, um plano lógico pode originar vários planos físicos;
- Os operadores físicos implementam as operações de álgebra relacional;
- O mesmo operador relacional pode ser implementado por mais do que um operador físico e o mesmo operador físico pode implementar mais do que um operador relacional
 - Exemplos:
 - O varrimento de uma tabela pode projetar ao mesmo tempo algumas colunas e efetuar restrições nos tuplos;
 - O operador físico “ordenar” pode ser usado por ORDER BY ou por uma junção por ordenação e fusão.

Observações

- Os planos de execução gerados são guardados:
 - Para poderem ser reutilizados;
 - Uma vez que uma recompilação tem um custo associado.
- Contudo,
 - A reutilização de um plano nem sempre é a melhor opção;
 - Dependerá da atual distribuição de dados na relação;
 - E eventuais alterações à *metadata* (e.g. remoção de uma *Constraint* ou *Index*).

5. Índices – MS SQL

Tipos de Índices MS SQL

- *Clustered*
 - Ordenado (agrupado);
 - “Armazena” a informação na estrutura de índice.
- *Non-clustered*
 - Apenas “aponta” para a informação.
- Uma tabela **apenas pode conter um índice clustered** e até **999 non-clustered**
- Uma tabela ***sem um índice clustered é denominada de heap.***
- Por defeito o SQLServer cria um índice *unique clustered* na PrimaryKEY, senão houver já criado, outro índice clustered. Nesse caso a PRIMARY KEY terá de ser **indexada em modo nonclustered**.

Cover vs Composite Index

- **Composite:**
 - Indexa um conjunto de colunas.
- **Cover:**
 - Indica num índice non-clustered, informação adicional para ser armazenada juntamente no índice.

Filtered index

- Exclui do índice um conjunto de registo de acordo com um critério
 - Tem o potencial de melhorar a performance, considerando o número de entradas e operações de atualização.

Exercício: Índice com múltiplas chaves de pesquisa

- Exemplo:

“Select account_number from account where branch_name = ‘Perryridge’ and balance = 1000

- Estratégias possíveis:
 - Usar um índice para branch_name e testar o valor de balance;
 - Usar um índice para balance e testar o valor de branch_name;
 - Usar índices para branch_name e balance e intersectar os resultados.
- Índices com chaves compostas, estratégias possíveis:
 - Chaves de pesquisa com mais de um atributo – Exemplo:
(branch_name,balance);
 - Mais eficiente que índices separados;
 - Também é eficiente em: branch_name = ‘Perryridge’ and balance < 1000;
 - Não é eficiente em: branch_name < ‘Perryridge’ and balance = 1000;

(Algumas) Linhas orientadores para Indexação

- Para tabelas frequentemente atualizadas, utilizar poucas colunas indexadas;
- Numa tabela com muitos dados, max taxa de atualização baixa, é recomendada a utilização de vários índices de acordo com o tipo de *queries* previsto;
- Colunas indexadas em modo *clustered index* devem ter valores de pequena dimensão, não nulos e preferencialmente/tendencialmente únicos;
- Em regra, quantos mais valores duplicados existirem na coluna indexada pior é a performance da indexação;
- Em índices compostos, interessa a ordem das colunas; colunas cujo os valores serão testados na clausula WHERE das *queries* devem ser colocados em primeiro no índice (colunas com os valores “mais únicos” devem ser colocados no fim);
- É muitas vezes desnecessário indexar tabelas de reduzida dimensão;
- Em geral, quando não é a chave de organização do dados deve ainda assim indexar-se a chave primária;
- É normalmente pertinente indexar chaves estrangeiras;
- Regularmente constituem-se índices secundários em atributos nestas condições:
 - Selection or join criteria;
 - ORDER BY;
 - GROUP BY;
 - Operações que envolvem ordenação como a UNION ou o DISTINCT
 - Atributos utilizados em computação de funções, exemplo – “**Select branchNo, AVG(Salary) from Staff Group by branchNo;**