

TEMA 6 – Parte 1  
Administración de las estructuras lógicas de almacenamiento

6.1. ESTRUCTURAS LÓGICAS DE ALMACENAMIENTO.

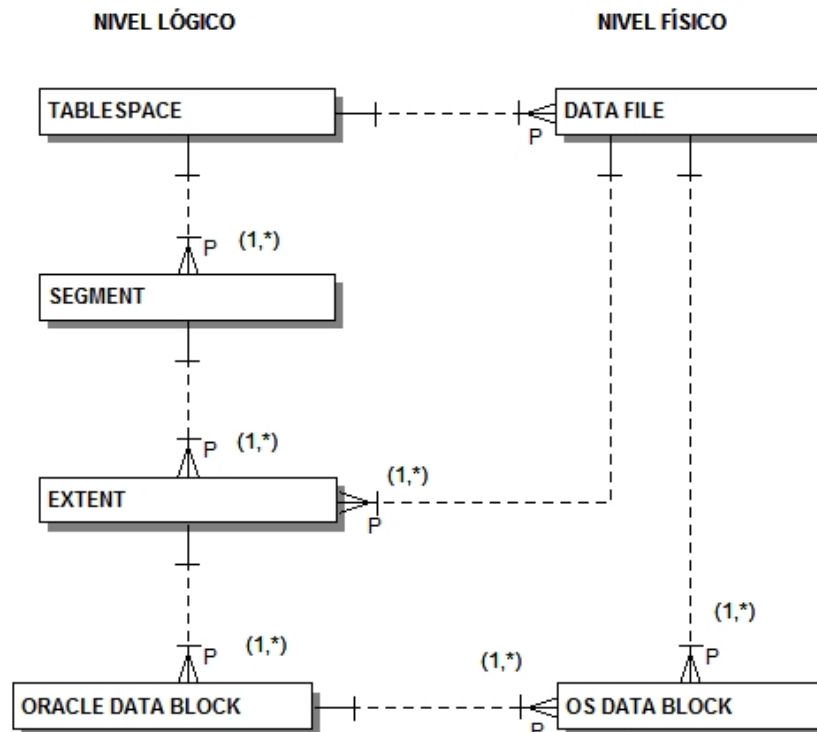
En los siguientes 2 capítulos se revisarán 2 aspectos muy importantes de una base de datos:

- Estructuras lógicas de almacenamiento (este capítulo).
- Estructuras físicas de almacenamiento (capítulos posteriores).

Las estructuras lógicas de almacenamiento son reconocidas únicamente por la base de datos, no por el sistema operativo. Las principales estructuras lógicas de una base de datos son:

- Bloques de datos.
  - Extensiones
  - Segmentos
  - Tablespaces.
- A nivel lógico la base de datos reserva espacio empleando las estructuras anteriores.
  - A nivel físico, los datos se almacenan en archivos llamados **data files** divididos en bloques de datos.
    - Otros archivos que representan a las estructuras físicas son: control files, Online Redo Log (archivos que son visibles o conocidos por el sistema operativo).

La siguiente figura muestra un modelo relacional que describe las estructuras lógicas y su relación con las estructuras físicas.

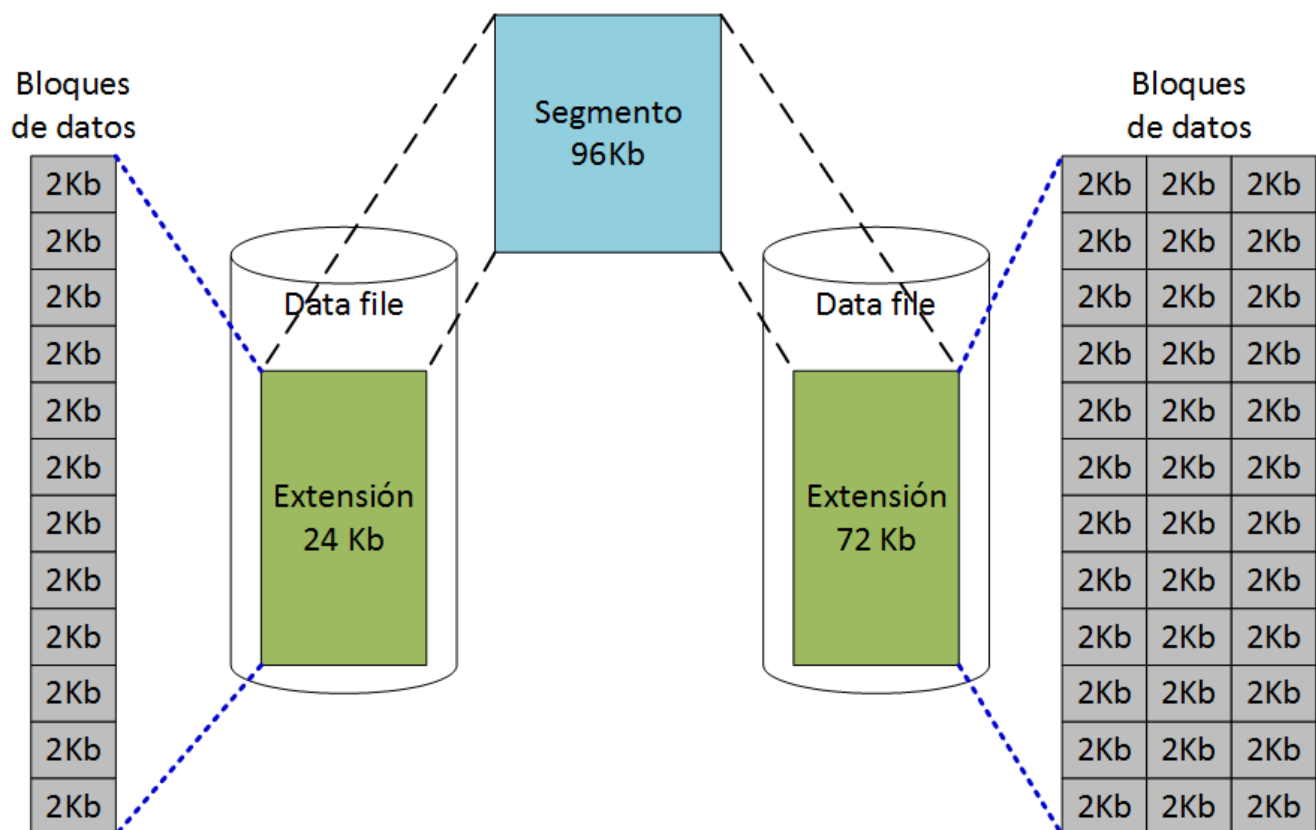


Observar la correspondencia que existe entre las estructuras físicas y lógicas:

- A nivel lógico un tablespace está integrado por varios *segmentos*.
- A nivel físico un tablespace está formado por uno o más *data files*.
- A nivel lógico, un data file está formado por un conjunto de extensiones
- A nivel físico un data file está formado por un conjunto de bloques de datos creados y administrados por el sistema operativo.
- Finalmente, un bloque de datos Oracle puede estar formado por varios bloques de datos creados y administrados por el sistema operativo. Típicamente es solo uno.

### 6.1.1. Jerarquías de las estructuras de almacenamiento.

Un segmento está formado por un conjunto de extensiones y su vez una extensión está formada por un conjunto de bloques de datos.

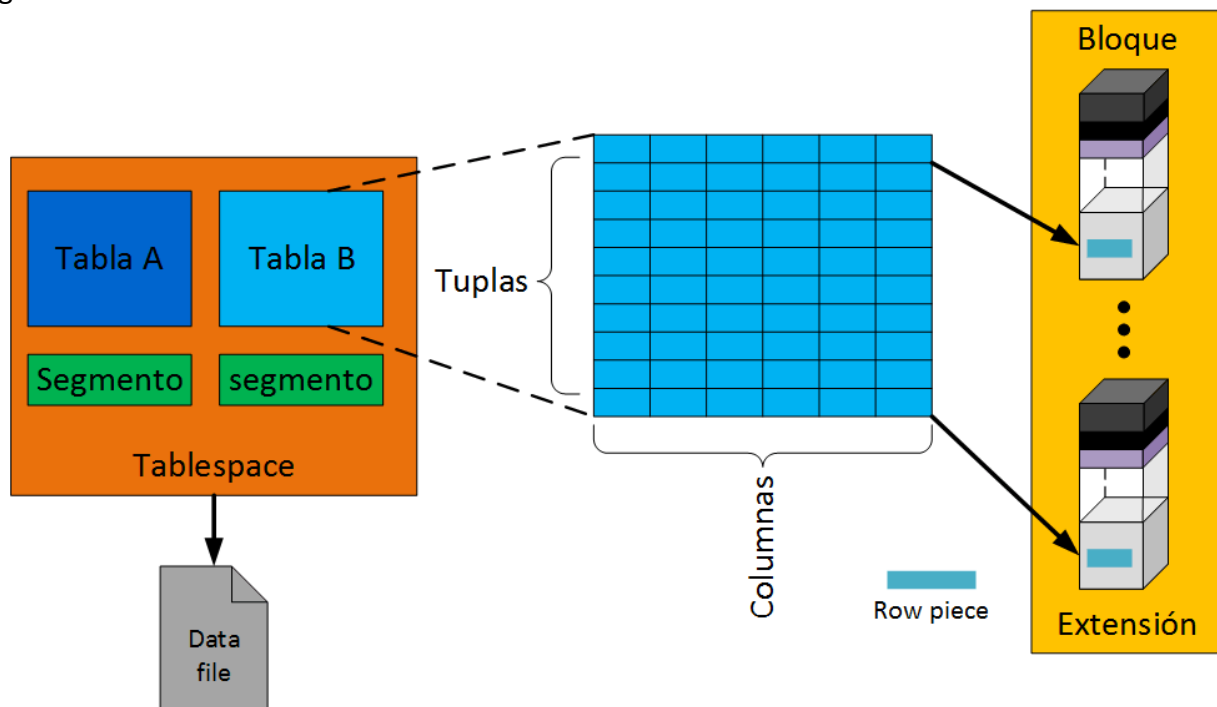


De menor a mayor nivel de generalidad se tienen las siguientes estructuras:

- **Data block**: Unidad lógica mínima de almacenamiento. Cada bloque está formado por un conjunto de bytes del disco. En el ejemplo anterior, cada bloque se forma por 2KB.
- **Extent**: Conjunto de bloques de datos contiguos que se reservan para almacenar información específica. En el ejemplo anterior, existen 2 extensiones. La primera de 24KB formadas por 12 bloques y la segunda de 72 KB formada por 36 bloques.
- **Segment**: Conjunto de extensiones reservadas para almacenar los datos que generan objetos específicos de la base de datos. Por ejemplo, una tabla, un índice, etc.

- **Tablespace:** Unidad de almacenamiento que contiene uno o más segmentos. Cada segmento debe pertenecer a un solo tablespace. Por lo tanto, todas las extensiones que forman al segmento son asociadas al mismo tablespace.
  - Notar que las extensiones que integran a un segmento pueden ubicarse en data files diferentes. Ejemplo: la extensión 1 puede estar almacenada en el data file `data01.dbf` y la extensión 2 en el archivo `data02.dbf`.
  - El contenido de una sola extensión siempre debe estar en el mismo data file.

La siguiente figura muestra otra vista en la que se puede apreciar la relación entre las estructuras físicas y lógicas.



## 6.2. BLOQUES DE DATOS.

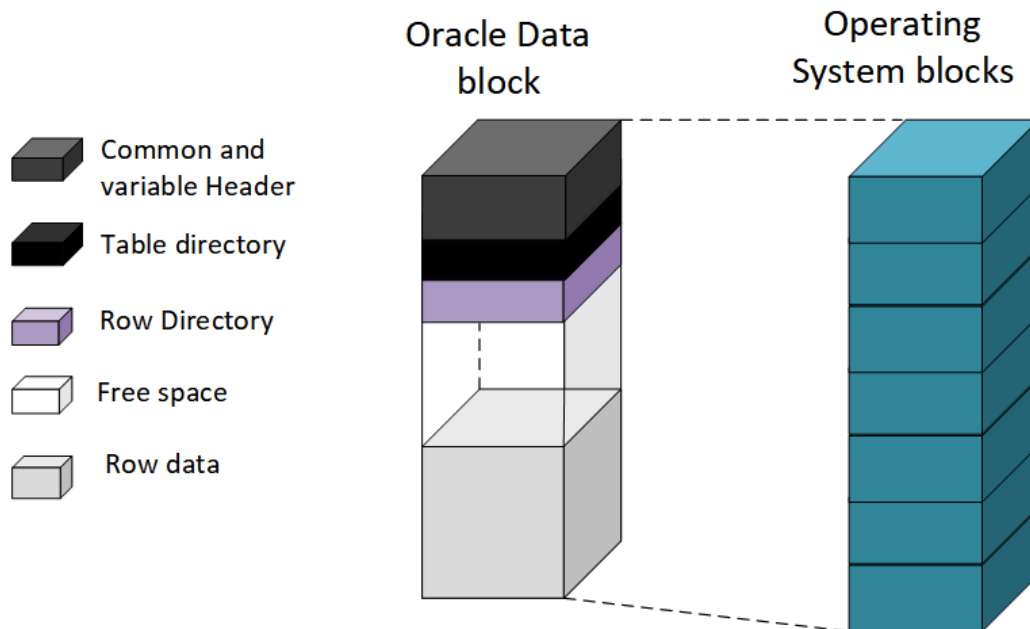
- Llamados también Oracle Data blocks o páginas.
- Representa la unidad mínima de operaciones I/O
- Un bloque de datos a nivel del sistema operativo representa de forma similar la unidad mínima de datos para realizar operaciones I/O.
- La estructura y formato de un bloque de datos Oracle es desconocido para el sistema operativo.
- Un bloque de datos puede estar formado de varios bloques de datos del sistema operativo.
- Como se revisó anteriormente, un bloque de datos cuenta con un tamaño especificado por el parámetro `db_block_size`. El tamaño estándar es de 4KB u 8KB.
- Si el tamaño difiere con respecto al tamaño de bloque del s.o., este debe ser un múltiplo.
- Para cada tablespace que se crea, se puede asignarle un tamaño de bloque diferente al valor del parámetro `db_block_size`.

### 6.2.1. Formato de un Bloque de datos.

- **Block Header:** Contiene información general acerca del bloque: disk address, tipo de segmento, información activa y transitoria de las transacciones que han modificado datos contenidos en el

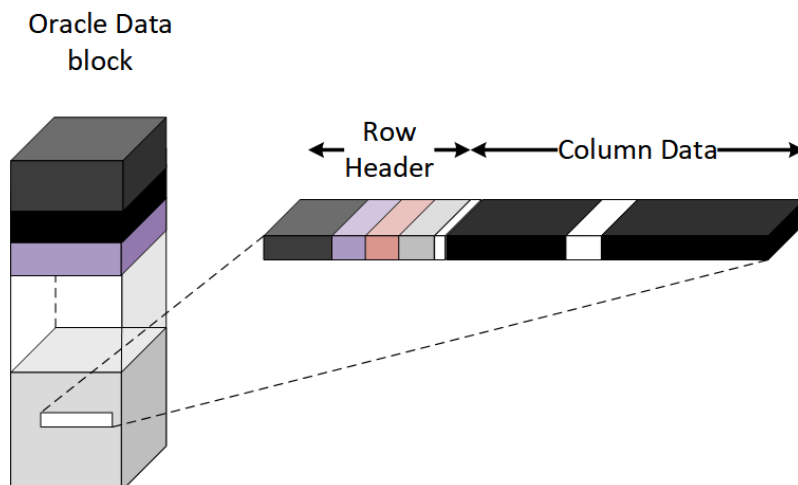
bloque (**transaction entry**). Típicamente el espacio que se reserva para guardar datos de las transacciones es de 23 bytes en la mayoría de los sistemas operativos.

- **Table directory:** Contiene metadatos de las tablas cuyos datos han sido almacenados en el bloque de datos. Varias tablas pueden almacenar parte de sus datos en el mismo bloque de datos.
- **Row Directory:** Indica la ubicación (puntero) dentro del bloque donde se encuentran ciertos registros de una tabla. El registro puede almacenarse en cualquier parte del área de datos del bloque (de abajo hacia arriba).
- En general el espacio que ocupa el header y los metadatos del bloque es aproximadamente entre 84 y 107 bytes.



### 6.2.2. Formato de un registro.

- Un registro de una tabla puede ser almacenado dentro uno o más bloques (típicamente en uno solo). Cuando un registro se almacena en varios bloques, a cada una de las partes almacenadas en el bloque se le conoce como "**row piece**". Si el registro cabe en un solo bloque solo se tendrá un **row piece** por cada registro.
- La longitud de cada row piece es variable.
- Al igual que un bloque de datos, el row piece tiene un formato que se emplea para administrar los datos que contiene.



### 6.2.2.1. Row header.

- Contiene la siguiente información:
  - Número de columnas
  - Información acerca de datos del registro almacenados en otros bloques de datos.
  - Generalmente un registro se almacena en un solo bloque, pero si por alguna razón el registro no cabe dentro de un mismo bloque, parte del registro será almacenada en otro.

### 6.2.2.2. Column data.

- Para cada columna del row piece se guarda la longitud del dato y el dato en sí.
- Si el tipo de dato de una columna es de longitud variable, esta sección puede crecer o contraerse dependiendo los valores de la columna.
- Como se mostró anteriormente, en el header del bloque de datos existe una sección llamada **row directory**. Cada entrada de este directorio contiene un puntero que lleva al inicio del registro.

### 6.2.2.3. ROWIDs

- La base de datos hace uso del concepto de ROWID para identificar de manera única y acceder de forma eficiente a los datos de un registro.
- Representa el mecanismo más eficiente y rápido para localizar a un registro.
- Un ROWID apunta a un data file, bloque y row number específico, formado por la siguiente estructura (emplea codificación Base 64)
  - **OOOOOO**: Data Object Number o ID que identifica el segmento de la base de datos.
  - **FFF**: El número de data file que contiene los datos del registro relativo a un tablespace.
  - **BBBBBB**: El bloque de datos que contiene al registro. Los números de bloque son relativos al data file al que pertenecen, no al tablespace. Dos registros con el mismo número de bloque pueden residir en dos data files diferentes del mismo tablespace.
  - **RRR**: El número de registro en el bloque.
- Los valores del ROWID no se almacenan de manera física. Su valor se infiere.

#### Ejemplo:

```
select rowid from mytable where id = 100;
```

ROWID

-----  
AAAPecAAFAAAABSA

### 6.2.3. Administración del espacio en bloques de datos.

- Los bloques de datos comienzan a llenarse de abajo hacia arriba.
- Lo anterior implica que el espacio disponible entre el “Row Data” y el header del bloque va disminuyendo conforme se agregan datos, cuando se hacen operaciones `update`, o cuando un dato se cambia de un valor `null` a un valor no nulo.
- La BD se encarga de administrar este espacio para optimizar desempeño y evitar desperdicio de espacios libres (fragmentaciones): *Automatic Segment Space Management*.

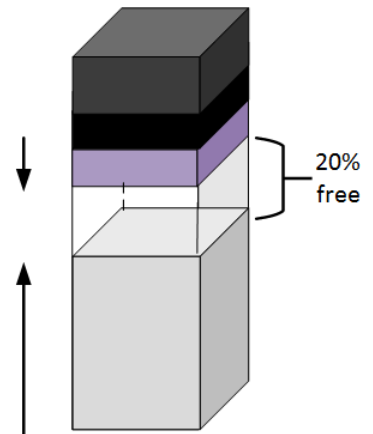
### 6.2.3.1. Porcentaje de espacio libre en bloques de datos.

- La cláusula `pctfree` se emplea al momento de crear una tabla para especificar el porcentaje mínimo de espacio que debe ser reservado en un bloque de datos para realizar operaciones sobre registros existentes de la tabla, por ejemplo, operaciones `update`.
- Especificar este porcentaje ofrece 2 principales beneficios:
  - Reducir la cantidad de espacios sin uso
  - Evitar una condición llamada **migración de registros** en la que un registro debe migrarse o moverse a otro bloque debido a que la nueva versión del registro ya no cabe en el espacio libre. Al contar con espacio libre reservado puede minimizar el riesgo de hacer dicho movimiento.
- Este parámetro resulta útil en especial en tablas donde no se esperan cambios considerables a los datos (tablas semi fijas).

#### Ejemplo:

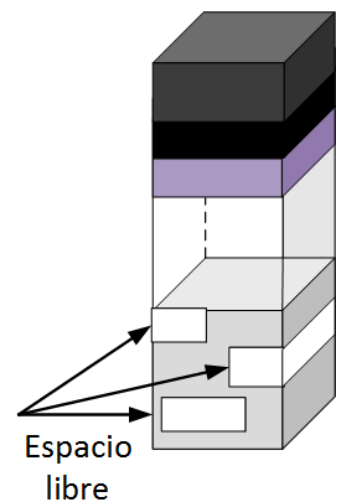
```
create table test(id number) pctfree 20;
```

- La instrucción anterior obliga a que el bloque tenga al menos un 20% de espacio libre el cual será destinado para realizar actualizaciones a los registros existentes.
- Si este espacio no se reservara, muy probablemente existirá la posibilidad de **migración de registros**.
- Instrucciones que pueden incrementar el espacio libre:
  - `delete`
  - `update` (el nuevo dato es de menor longitud). Si ocurre lo contrario, el registro pudiera requerir ser migrado a otro bloque
  - `Insert` Hacen uso de compresión de datos.
- Una instrucción `insert` podría hacer uso del espacio liberado por otra instrucción bajo las siguientes condiciones:
  - La instrucción que libera espacio se ejecuta primero y ambas instrucciones pertenecen a la misma transacción.
  - La instrucción que libera espacio se ejecuta primero, pertenece a una transacción diferente a la instrucción `insert`. En este caso la instrucción `insert` podrá hacer uso del espacio hasta haber realizado `commit`.



### 6.2.3.2. Optimización al defragmentar el bloque.

- El espacio que se libera en un bloque puede no ser contiguo llamado **espacio fragmentado**.
- La BD realizará una operación de defragmentación bajo las siguientes condiciones:
  - El espacio libre total del bloque fragmentado permite actualizar un registro existente o agregar uno nuevo
  - El nivel de fragmentación del bloque impide que el registro se guarde de forma contigua.

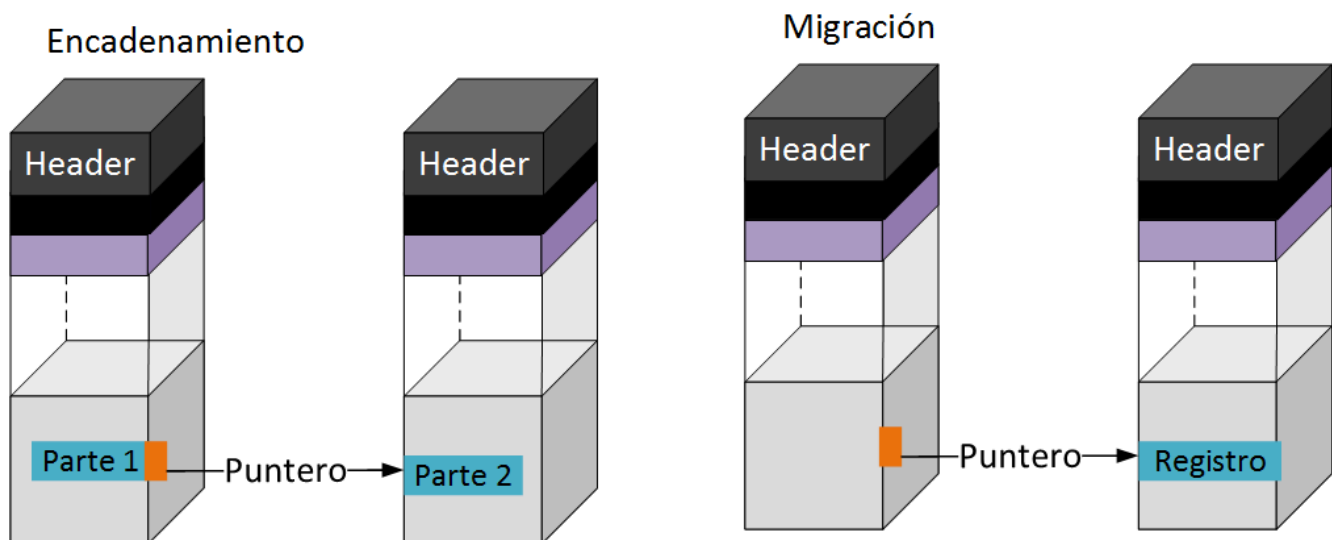


### 6.2.3.3. Encadenamiento de registros.

- Ocurre en situaciones donde el tamaño del registro es demasiado grande para ser almacenado en un solo bloque, por ejemplo, en registros con una gran cantidad de columnas, o en registros que contengan columnas con tipos de datos LONG, o LONG RAW.
- En esta situación El registro se guarda en una cadena formada por 2 o más bloques asignados al mismo segmento.
- Otro escenario de encadenamiento ocurre cuando el registro originalmente se encuentra en un solo bloque pero después es actualizado, crece de tamaño y ya no existe suficiente espacio para guardar la nueva versión (de aquí el beneficio de emplear la cláusula `pctfree`).

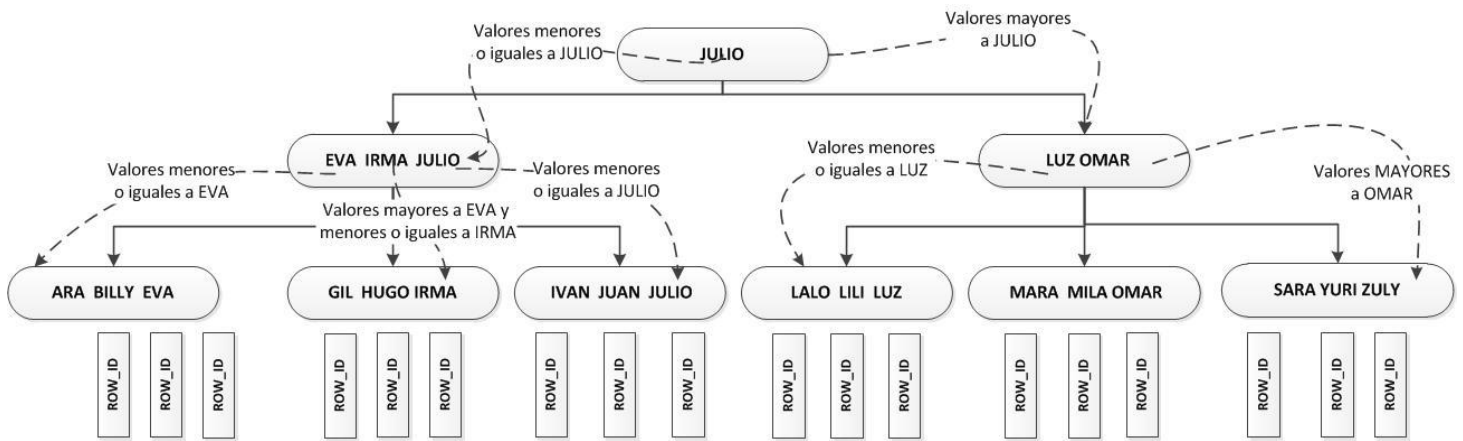
#### 6.2.3.4. Migración de registros.

- En este caso, el registro es migrado a un nuevo bloque donde se tenga el espacio suficiente.
- En el bloque original se guarda un puntero (*forwarding address*) que hace referencia al segundo bloque.
- El ROWID del registro migrado no cambia.
- Adicional a los eventos mencionados en encadenamiento, una migración ocurre cuando el registro tiene más de 255 columnas.



#### 6.2.4. Index Blocks.

- Existe otro tipo de bloques empleados específicamente para administrar el almacenamiento de los índices.
- Su forma de almacenamiento difiere de los bloques convencionales.
- Debido a la naturaleza de los índices los cuales representan una estructura de árbol B, se consideran 3 tipos de bloques para índices:
  - Bloques para el nodo raíz (Root block)
  - Bloques para nodos intermedios (Branch block)
  - Bloques para las hojas del árbol (Leaf Block).

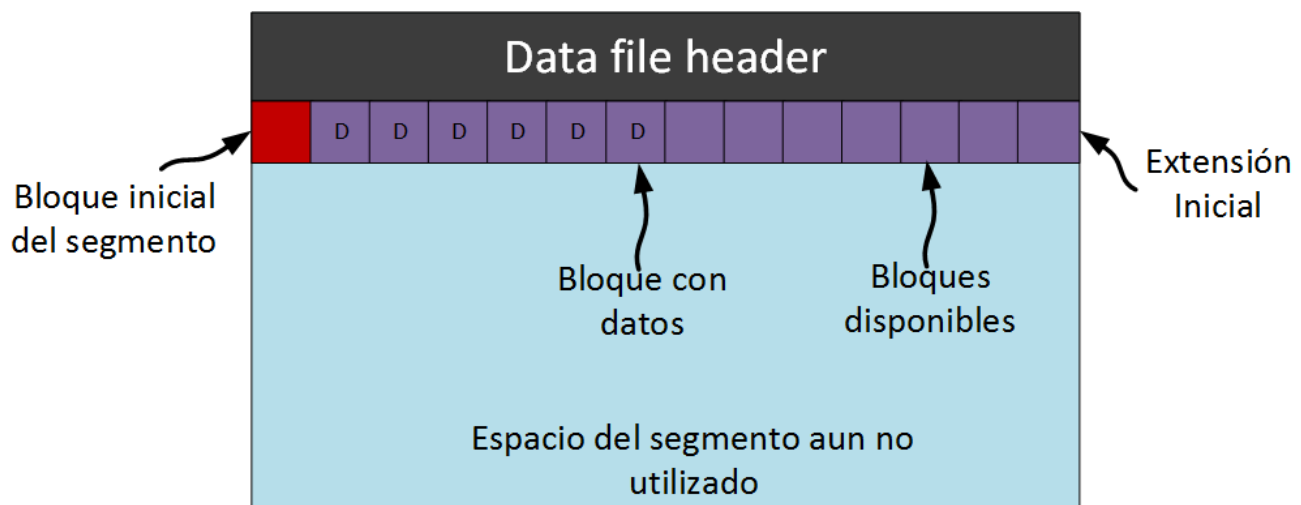


- A diferencia de los datos de una tabla, los datos o “keys” de los índices se almacenan de forma ordenada en los Index blocks.
- La defragmentación de este tipo de bloques no es automática, se debe ejecutar `alter index <nombre_indice> rebuild o coalesce`.
- Esta operación permite que las hojas del árbol B se reduzcan.



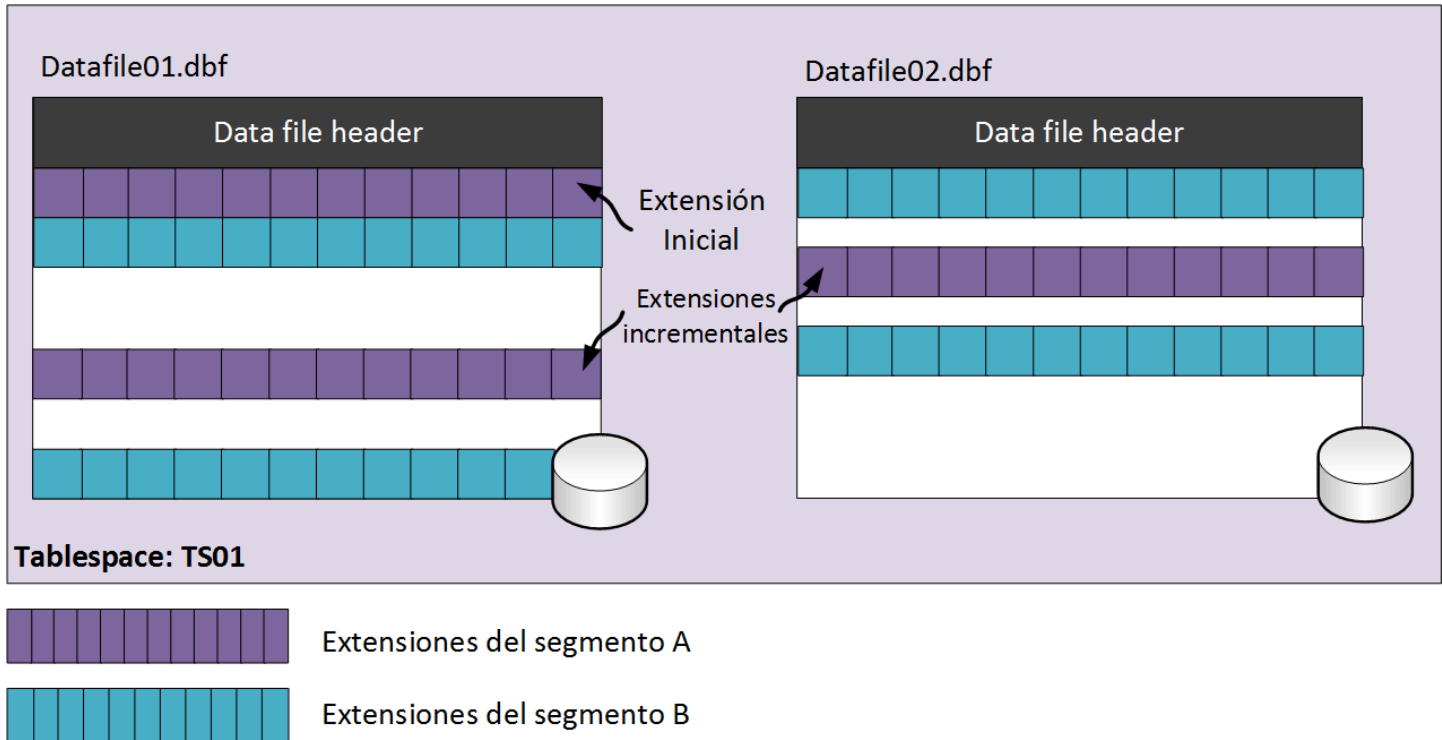
### 6.3. EXTENSIONES.

- Una extensión representa una unidad de almacenamiento formada por un conjunto de bloques contiguos.
- Por default la base de datos crea una sola extensión cuando se crea un segmento. Inicialmente, al crear un segmento, se le asigna una sola extensión sin importar si el objeto contiene datos.
- Como se verá más adelante, un segmento se crea al momento de crear un objeto, por ejemplo, una tabla. La relación entre el objeto y el segmento es 1:1
- El primer bloque de datos del segmento representa a un directorio que lista todas las extensiones que forman al segmento.





- Si la extensión inicial se llena y si se requiere más espacio, la BD automáticamente reserva una nueva extensión llamada *extensión incremental*.
- Si el tablespace que contiene al segmento está configurado como *locally managed* la BD realiza una consulta en el bitmap de algún data file para determinar si existen suficientes bloques contiguos para generar una nueva extensión. En caso de no existir, se revisa en otros data files.
- Todas las extensiones de un segmento deben pertenecer al mismo tablespace, pero pueden estar en diferentes data files.



- En la imagen anterior, se muestran 2 data files que contienen extensiones de diferentes segmentos (segmento A y segmento B), todos ellos, pertenecen al mismo tablespace TS01 integrado por 2 data files.
- No necesariamente las extensiones deben ubicarse en el mismo data file, pero si en el mismo tablespace.

### 6.3.1. Liberación de extensiones.

- Las extensiones se liberan para ser reutilizadas cuando ocurren eventos como los siguientes:
  - Al ejecutar la sentencia `drop` sobre el objeto
  - Al ejecutar la sentencia `truncate` sobre la tabla.
  - Notar que al eliminar todos los registros de una tabla con la instrucción `delete`, las extensiones no son liberadas.
  - Es posible liberar extensiones de forma manual. Existe un *advisor* (*segment advisor*) que permite conocer o detectar los objetos que tienen espacio suficiente para ser recuperado basado entre otras cosas, en el nivel de fragmentación.
  - Es posible invocar una operación de defragmentación de un segmento. Esto permite adicional a la recuperación de espacio, contar con tablas con datos contiguos, mejorando así las lecturas en especial para operaciones de escaneo completo: *table Access full*.

### 6.3.2. Parámetros de almacenamiento para extensiones.


Cada segmento es definido en términos de parámetros de almacenamiento expresados en forma de extensiones. Los valores de estos parámetros permiten realizar el control para realizar la asignación de espacio libre a un segmento.

Los valores de estos parámetros son definidos en el siguiente orden de precedencia:

- Empleando la cláusula `segment storage`.
- Empleando la cláusula `tablespace storage`
- Empleando los valores por default a nivel de la base de datos.

Un tablespace configurado como “Locally managed” puede contar con extensiones de tamaño fijo o variable.

#### 6.4. SEGMENTOS.



Ejercicio práctico 01

- Conjunto de extensiones que contienen todos los datos de un objeto ubicado dentro de un tablespace. Dependiendo el tipo de objeto, los segmentos se clasifican en:
  - *User segments*
  - *Temporary segments*
  - *Undo segments*.

##### 6.4.1. User segments.

- Empleados en objetos para almacenar los datos de los usuarios:
  - Tablas, tablas particionadas, cluster de tablas.
  - Particiones LOB
  - Índices, índices particionados.
- Los objetos que no están particionados se asocian a un solo segmento.
- Si una tabla está particionada en 5 particiones, esta contará con 5 segmentos.
- Por default, la creación del segmento se realiza hasta que el usuario comienza a almacenar datos. A esta estrategia se le conoce como *Deferred Segment Creation*.
- Por ejemplo, el segmento no se crea al ejecutar la sentencia `create table`.
- Este comportamiento se controla con el parámetro `deferred_segment_creation`. Su valor por default es `true`.
- A nivel de tabla esta configuración puede sobrescribirse. Para ello se hace uso de la cláusula `segment creation deferred` o `segment creation immediate`

#### Ejemplo:

```
create table my_table (  
    id number,  
    constraint my_table_pk primary key (id)  
)  
segment creation deferred;
```

- La razón del comportamiento anterior se debe a que la creación de un solo objeto puede requerir la creación de varios segmentos los cuales no necesariamente van a comenzar a utilizarse de forma inmediata.
- Dependiendo el tipo de objeto y sus características, el objeto puede provocar la creación de varios segmentos, no siempre se crea uno solo.

### Ejemplo:

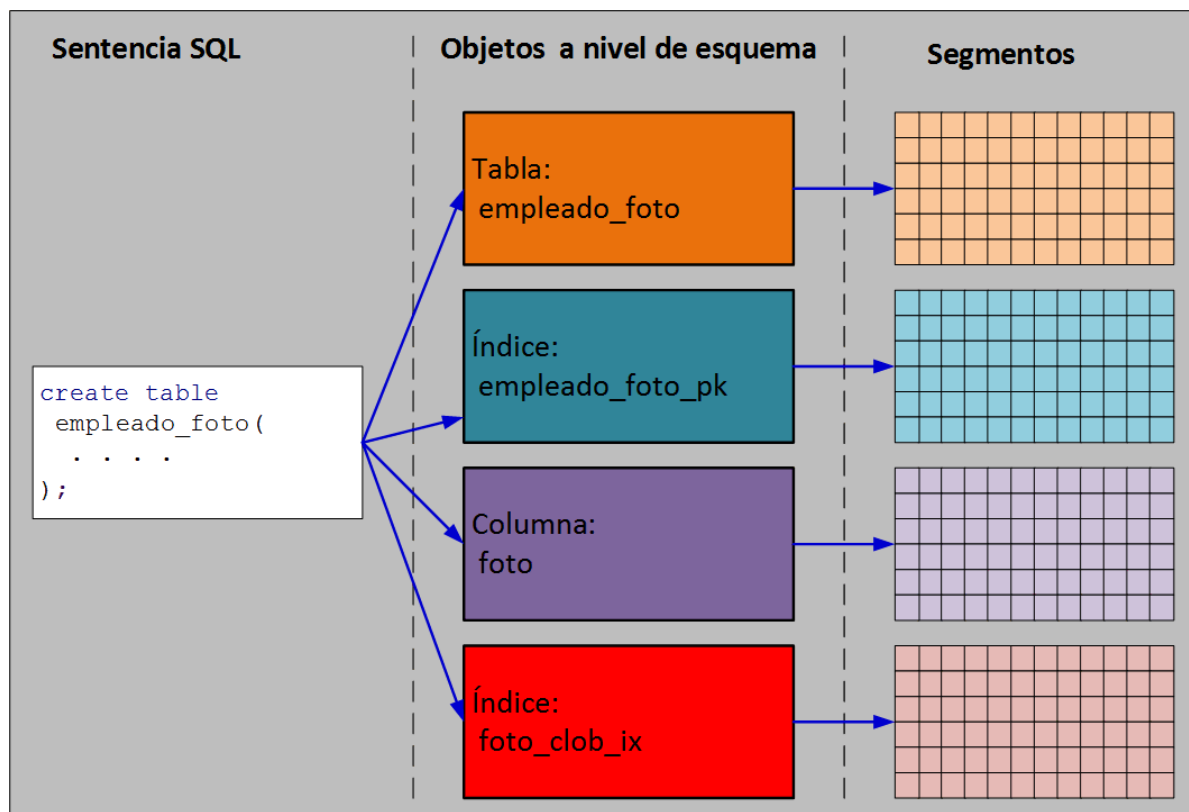
Identificar los segmentos que se crearán cuando comience a usarse la siguiente tabla:

```
create table empleado_foto(
  empleado_id number constraint empleado_foto_fk primary key,
  foto blob
);
```

Al agregar un registro a la tabla se crearán los siguientes segmentos:

- Implícitamente el manejador genera un índice tipo unique para implementar la PK. Los datos del índice se guardarán en un **segmento**.
- Para el caso de la columna con tipo de dato clob, sus datos se almacenan en un **segmento** independiente.
- Para cada columna CLOB/BLOB se crea un índice (clob/blob index) el cual es almacenado en otro **segmento**.
- El resto de las columnas de la tabla se almacenarán en el **segmento** principal de la tabla.

De lo anterior, se crearán 4 segmentos. El momento en el que se crearán dependerá de la configuración que tome la tabla, es decir, todos estos tipos de segmentos heredan la configuración del objeto que los contiene.



#### 6.4.2. Temporal segments.

- Cuando una sentencia SQL se ejecuta, la BD puede requerir de cierto espacio para almacenar temporalmente el resultado de cada una de las fases o etapas del procesamiento.
- Operaciones típicas que requieren de este espacio son ordenamiento, construcción de tablas hash, etc. Cuando un índice se crea, sus segmentos se almacenan como temporales durante el proceso de creación y al concluir se vuelven permanentes.
- A nivel general, si una operación se puede realizar en memoria, no se hace uso de segmentos temporales. Solo las operaciones que no pueden realizarse en memoria hacen uso de este tipo de segmentos.
- Otro uso de los segmentos temporales es durante la ejecución de consultas. En algunos casos puede ser necesario la creación de un segmento temporal para procesar la consulta y al terminar, el segmento se elimina.
- Otro uso de este tipo de segmentos son las **tablas temporales** y sus **índices**. Debido a que estos datos desaparecen al terminar la sesión, se prefiere utilizar segmentos temporales. Las extensiones de estos segmentos solo pueden ser accedidas por la sesión (usuario) que los genera.
- Debido a su frecuente uso, se considera buena práctica crear un tablespace dedicado para almacenar segmentos temporales al momento de crear la base de datos (tal cual como se realizó en el tema 2).
- A cada usuario se le asocia un tablespace temporal, el mismo tablespace puede ser empleado por múltiples usuarios.

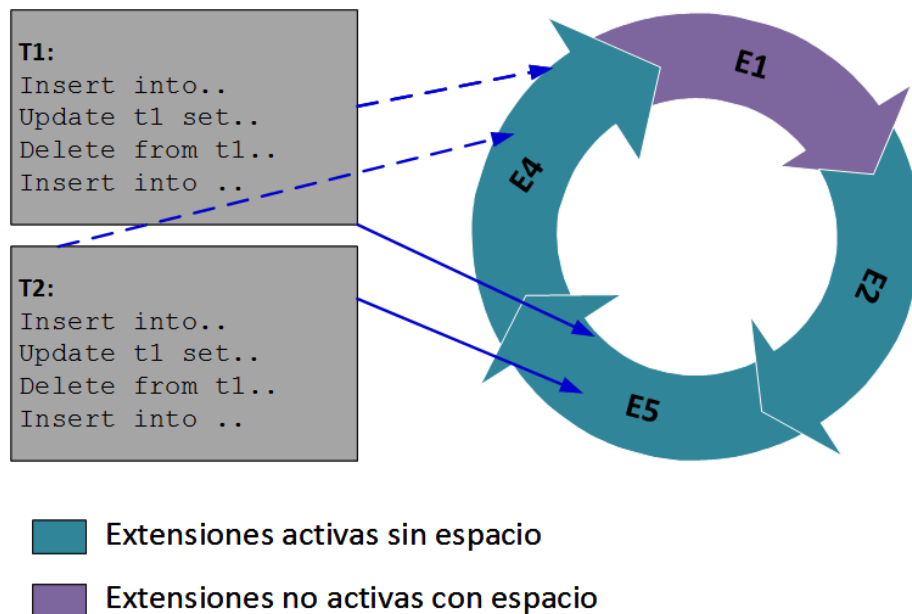
#### 6.4.3. Segmentos UNDO

- Como parte de la operación normal de la base de datos, y en especial durante la ejecución de una transacción, la base de datos almacena cierta información y así contar con la capacidad para realizar las siguientes acciones:
  - Hacer rollback de una transacción.
  - Hacer la recuperación de una transacción
  - Implementar la propiedad ACID Read Consistency
  - Realizar algunas operaciones relacionadas con la funcionalidad flashback.
- A este tipo de información se le conoce como **undo data**.
- A diferencia de los datos REDO que se guardan en Online Redo Logs, Undo data se almacena en bloques similar a los datos en el db buffer cache.
- Los datos undo generados por objetos permanentes son almacenados en un tablespace especial llamado **undo tablespace**.

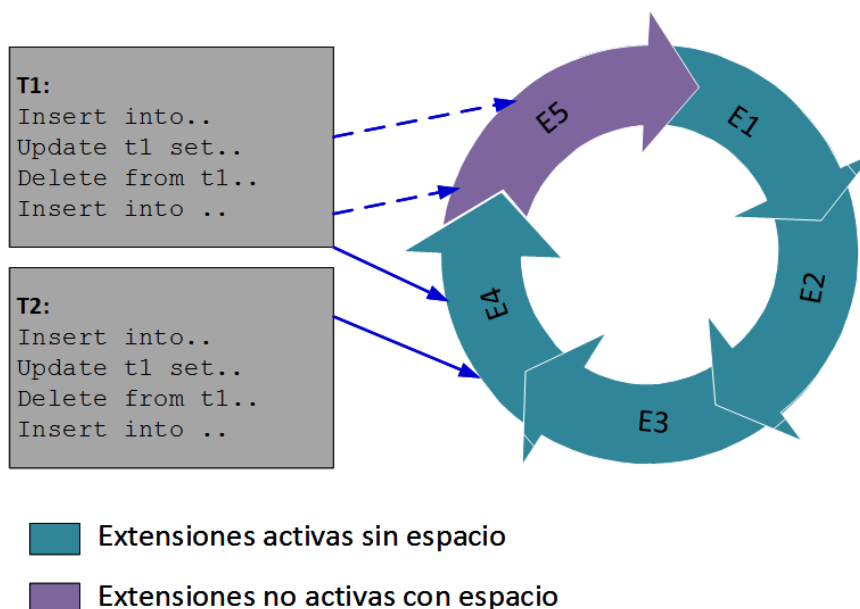
##### 6.4.3.1. Segmentos Undo y transacciones.

- Cuando una transacción se inicia, se le asigna un segmento undo a dicha transacción.
- Múltiples transacciones pueden escribir de forma concurrente al mismo o a diferentes segmentos.
- De forma conceptual las extensiones de un segmento undo forman un **anillo**.
- Cada transacción escribe a una extensión del segmento undo. Si el espacio de la extensión se termina, continua escribiendo en la siguiente.
- Varias transacciones pueden escribir a la misma extensión.

- Dentro de cada extensión se encuentran los bloques de datos. Un bloque de dato contiene datos de **una sola** transacción.
- En el siguiente ejemplo, las transacciones T1 y T2 comienzan a escribir en la extensión E4, y posteriormente, cuando esta se llena, continúan en la siguiente extensión E5, y así sucesivamente en un estilo circular.



- Lo anterior quiere decir que las extensiones se van sobrescribiendo una vez que los datos contenidas en ellas se vuelven **extensiones undo inactivas**.
- En caso de no poder sobrescribir datos debido a que todas las extensiones tienen datos **undo activos**, se crea una nueva extensión.
- En el siguiente ejemplo, se crea una nueva extensión E5 ya que las demás extensiones ya no cuentan con espacio y los **datos undo** que contienen aún siguen activos.



Ejercicio  
práctico 02

- De forma similar a los segmentos temporales, a cada usuario se le asigna un tablespace que almacena *datos tipo undo*. Esto con la finalidad de mejorar desempeño al separar los segmentos permanentes de los temporales y de los *datos undo*.

#### 6.5. ADMINISTRACIÓN DEL ESPACIO EMPLEANDO SEGMENTOS.

- Para realizar la administración del espacio, la base de datos realiza un seguimiento del estado de los bloques de datos en el segmento.
- El concepto de **High Water Mark (HWM)** se refiere al punto dentro de un segmento en el que posterior a él, los bloques nunca se han empleado, es decir, existen sin formato.
- Existen 2 técnicas para realizar la administración del espacio de un segmento:
  - Automatic Segment Space Management (**ASSM**)
  - Manual segment space management (**MSSM**).

##### 6.5.1.1. Manual segment space management (**MSSM**).

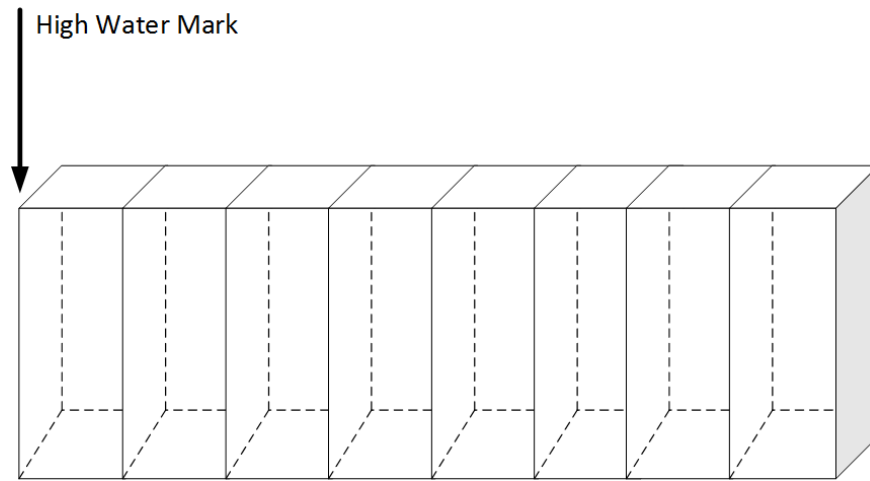
- Se hace uso de estructuras llamadas **free lists**.
- Cuando se realiza la inserción por primera vez de un registro a una tabla, se realiza una búsqueda en una lista para determinar si existen bloques de datos con espacio.
- En caso de no encontrar bloques dentro de la lista, se obtienen de algún data file, genera un grupo de bloques, los formatea y los inserta en la lista. Posteriormente comienza a insertar los datos.
- Esta estrategia ya no es empleada actualmente.

##### 6.5.1.2. Automatic Segment Space Management (**ASSM**)

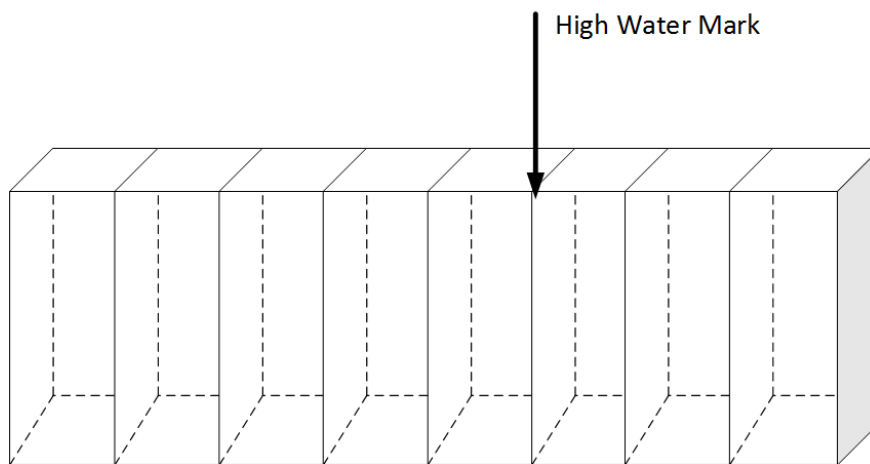
- Cuando se realiza la inserción por primera vez de un registro a una tabla, adicional a la creación del segmento, se le da formato a un bloque para formar un **bitmap block**.
- Cada segmento contiene su **bitmap block** y es empleado para conocer el **status** de espacio de cada bloque (qué tan lleno o vacío se encuentra cada uno).
- Cuando se encuentra un bloque disponible, este se formatea y posteriormente se comienza a escribir en él.
- Esta técnica es la que se emplea por default y la más recomendada. Entre sus beneficios:
  - Mejora el uso del espacio disponible.
  - Permite almacenar datos en múltiples bloques de forma simultánea mejorando desempeño.
- Para realizar la administración de los bloques de datos en el **bitmap** se consideran los siguientes estados:
  - *Bloque ubicado arriba de la marca de agua*: Significa que el bloque está libre, nunca se ha empleado, y no está formateado.
  - *Bloque ubicado abajo de la marca de agua*: Se divide a su vez en 3 posibles estados:
    - Se reservó para ser empleado pero aún no ha sido formateado ni usado.
    - Formateado y contiene datos.
    - Formateado y vacío debido a que sus datos fueron eliminados.

#### Ejemplos:

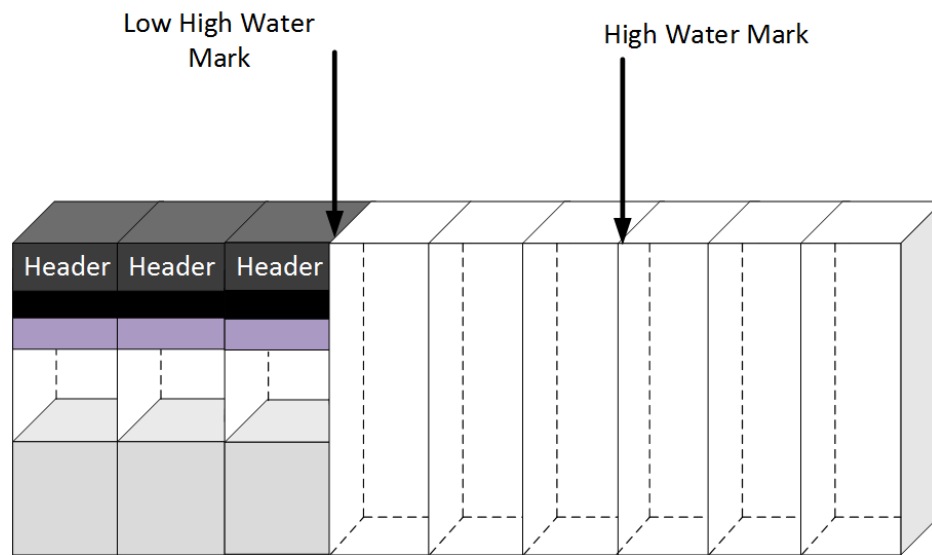
- Cuando la tabla se crea, la marca de agua se encuentra del lado izquierdo, al inicio del segmento.
- Notar que los bloques se representan en blanco ya que están vacíos y no tienen formato.



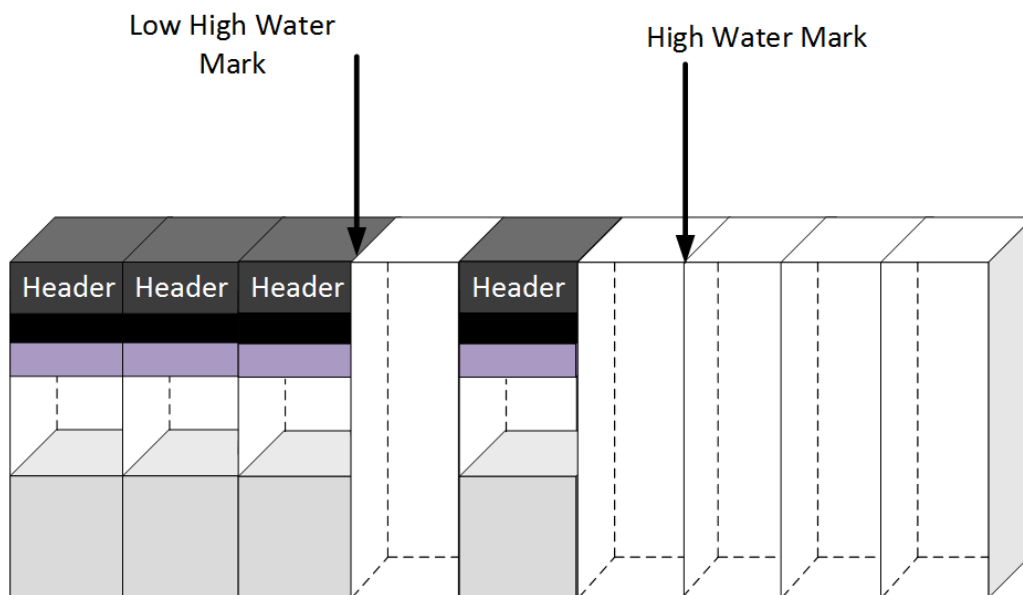
- Suponer que una transacción comienza a realizar inserciones. Esto significa que será necesario reservar un grupo de bloques para poder almacenar los registros.
- En este momento, se le aplica formato a un solo bloque, y se almacena el bitmap en el que se indica que los bloques han sido reservados. Todos los demás bloques del grupo permanecen sin formato.
- La marca de agua se establece justo después de este grupo de bloques.



- Al comenzar a realizar las inserciones, los bloques comienzan a formatearse y llenarse de datos.
- No necesariamente todos los bloques reservados serán utilizados. Por ejemplo, suponer que las inserciones que realiza la transacción solo ocuparon 3 bloques. El segmento se vería de la siguiente manera:



- Notar la existencia de una nueva marca llamada **low high water mark**. Indica el punto hasta el cual **todos** los bloques han sido formateados ya sea porque contienen datos o están vacíos con formato porque se realizó alguna operación de eliminación de datos.
- Los otros 3 bloques restantes fueron reservados en el bitmap pero aún no se han ocupado.
- Notar que la base de datos puede continuar insertando datos dentro del grupo reservado de bloques, y lo puede hacer en cualquiera de ellos o en algún bloque formateado con espacio.
- Esto puede generar ciertos huecos dentro del grupo reservado.
- Notar que **low High Water mark** permanece en la posición indicada en la figura a pesar de existir bloques con datos a la derecha. Recordando, todos los bloques a la izquierda de esta marca deben estar formateados.



- **Low High Water Mark** es muy importante en especial cuando se desea leer todo el contenido de la tabla (full table Access).
- En esta operación, se realiza la lectura de forma secuencial de todos los bloques hasta encontrar a la marca Low High Water Mark.



- La lectura hasta este punto se realiza de forma secuencial y rápida ya que se tiene la certeza que todos los bloques han sido formateados.
- Posteriormente se hace una lectura más detallada y cuidadosa para leer únicamente los bloques de datos que han sido formateados entre la marcas **Low High Water Mark** y **High Water Mark**.
- En el ejemplo, el bloque 4 y 6 se descartan y el 5 se lee.
- Cuando el espacio entre estas 2 marcas se llena, **low high Water Mark** toma el lugar de **High Water Mark** y **High Water Mark** se mueve hacia la derecha para reservar más bloques.

## 6.6. VISUALIZACIÓN DE INFORMACIÓN ACERCA DEL USO DEL ESPACIO DE ALMACENAMIENTO.

Existen 2 principales mecanismos

- Empleando paquetes PL/SQL
- Empleando vistas del diccionario de datos.

### 6.6.1. Visualización empleando paquetes PL/SQL

Nombre del paquete /función	Descripción
dbms_space.unused_space	Obtiene información acerca del espacio no utilizado en un objeto (índice, tabla, o cluster).
dbms_space.free_blocks	Obtiene información de los bloques de datos libres en un objeto con administración basada en listas (administración manual del espacio libre).
dbms_space.space_usage	Obtiene información de los bloques de datos en un objeto con administración automática del espacio libre.

#### Ejemplo:

```
--@Autor:          Jorge A. Rodriguez C
--@Fecha creación:  dd/mm/yyyy
--@Descripción:    Consulta el espacio libre empleando funciones PL/SQL
```

```
whenever sqlerror exit rollback
```

```
Prompt Conectando como Jorge para crear tabla con datos.
```

```
connect jorge/jorge
```

```
set serveroutput on
```

```
--Programa 1
```

```
declare
```

```
  v_query varchar2(1000);
```

```
  v_index number;
```

```
begin
```

```
  begin
```

```
    execute immediate 'drop table numeros';
```

```
  exception
```

```
    when others then
```

```
      if sqlcode = -942 then
```

```
        dbms_output.put_line('La tabla numeros no existe, se creará');
```

```
      else
```

```
        raise;
```

```
      end if;
```

```
  end;
```

```

execute immediate
  'create table numeros(id number constraint numeros_pk primary key,
    numero_aleatorio number)';

--inserta 10,000 registros
for v_index in 1..10000 loop
  execute immediate 'insert into numeros values (:id,:num_aleatorio)'
    using v_index, dbms_random.random;
end loop;

commit;

dbms_output.put_line('Ok, ' || v_index || ' insertados');

end;
/

--Programa 2
declare
  v_total_blocks number;
  v_total_bytes number;
  v_unused_blocks number;
  v_unused_blocks_bytes number;
  v_last_used_extent_file_id number;
  v_last_used_extent_block_id number;
  v_last_used_block number;
begin
  /**
  Parámetros del procedimiento
  1. Nombre del esquema
  2. Nombre del objeto
  3. Tipo de objeto. Por ejemplo, TABLE.
  4. (out) Número total de bloques existentes en el segmento
  5. (out) Número total de bloques existentes en el segmento en bytes
  6. (out) Número de bloques que no han sido empleados
  7. (out) Número de bloques que no han sido empleados en bytes
  8. (out) Identificador del data file que contiene la última extensión
    que contiene datos.
  9. (out) Identificador del primer bloque de datos que pertenece a la
    última extensión que contiene datos.
  10. (out) Identificador del último bloque de datos que contiene datos.

  11 y 12 no relevantes para el curso.
  */
  dbms_space.unused_space (
    segment_owner      => 'JORGE',
    segment_name       => 'NUMEROS',
    segment_type       => 'TABLE',
    total_blocks       => v_total_blocks,
    total_bytes        => v_total_bytes,
    unused_blocks      => v_unused_blocks,
    unused_bytes       => v_unused_blocks_bytes,
    last_used_extent_file_id => v_last_used_extent_file_id,
    last_used_extent_block_id => v_last_used_extent_block_id,
    last_used_block    => v_last_used_block,
    partition_name     => null);

```

```

dbms_output.put_line('v_total_blocks:      '||v_total_blocks);
dbms_output.put_line('v_total_bytes:      '||v_total_bytes);
dbms_output.put_line('v_unused_blocks:    '||v_unused_blocks);
dbms_output.put_line('v_unused_blocks_bytes:  '||v_unused_blocks_bytes);
dbms_output.put_line('v_last_used_extent_file_id:  '||v_last_used_extent_file_id);
dbms_output.put_line('v_last_used_extent_block_id: '||v_last_used_extent_block_id);
dbms_output.put_line('v_last_used_block:    '||v_last_used_block);

```

```
end;
```

```
/
```

### Salida:

```

v_total_blocks:      32
v_total_bytes:      262144
v_unused_blocks:    0
v_unused_blocks_bytes: 0
v_last_used_extent_file_id: 4
v_last_used_extent_block_id: 248
v_last_used_block: 8

```

### **6.6.2. Vistas del diccionario de datos asociadas con el uso de espacio de almacenamiento**

Vista	Descripción
dba_segments user_segments	Describe los datos de los segmentos reservados para los objetos de todos los usuarios o de un usuario actual.
dba_extents user_extents	Describe los datos de las extensiones reservadas para los objetos de todos los usuarios o de un usuario actual.
dba_free_space user_free_space	Contiene datos acerca de las extensiones libres en todos los tablespaces o en los tablespaces a los que un usuario se le ha asignado una cuota.

### Ejemplo 1:

```

select segment_name, tablespace_name, bytes, blocks, extents
from dba_segments
where segment_type = 'TABLE'
and owner='JORGE'
order by segment_name;

```

SEGMENT_NAME	TABLESPACE_NAME	BYTES	BLOCKS	EXT...	
BIN\$klhxymlU3PgUwEAAH+Z4w==\$0	USERS	262144	32	4	
NUMEROS	USERS	262144	32	4	
TEST	USERS	65536	8	1	

### Ejemplo 2:

```

select segment_name, segment_type, tablespace_name, extent_id,
       bytes, blocks
from dba_extents
where segment_type = 'TABLE'
and owner='JORGE'
order by segment_name;

```

SEGMENT_NAME	SEGMENT_TYPE	TABLESPACE_NAME	EXTENT_ID	BYTES	BLOCKS
NUMEROS	TABLE	USERS	1	65536	8
NUMEROS	TABLE	USERS	0	65536	8
NUMEROS	TABLE	USERS	3	65536	8
NUMEROS	TABLE	USERS	2	65536	8
TEST	TABLE	USERS	0	65536	8



**Ejercicio  
práctico 03**