UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Ingeniería en Sistemas de Información

L i s s a n d r a

El mundo comenzó y terminará en el File System.

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

- 1C2019 -
Versión 1.5

Versión de Cambios

v1.0 (28/03/2019)

- Publicación del documento original

v1.1 (4/4/2019)

- Agregado de comando RUN <path> en Kernel para correr archivos LQL (antes no existía)
- Ahora definimos que las Keys serán de tipo *u_int16*
- Definimos qué campos de los archivos de configuración pueden ser actualizadas en runtime *(todas las de tiempo/delays)*

v1.2 (11/4/2019)

- Agregado de comando “METRICS” en Kernel para mostrar las métricas por pantalla (antes no existía)
- Agregado de diagrama de atención de pedidos en LFS
- Eliminamos que las memorias mantengan metadata y estructuras administrativas sobre las tablas que manejan
- Aclaramos que los archivos metadata de las tablas no son por partición, sino una para toda la tabla

v1.3 (25/4/2019)

- Incluimos ejemplos del funcionamiento del File System en donde se ve la utilización de los bloques
- Modificamos que el Kernel solo falle ante un INSERT/SELECT/DROP si dentro de la metadata no figura esa tabla
- Aclaramos que el link incluido para calcular timestamps devuelve este dato en microsegundos
- Aclaramos que al inicial la Memoria solo se debe reservar la memoria para la tabla de páginas y la memoria principal mientras que la memoria para la tabla de segmentos se irá reservando dinámicamente

v1.4 (5/5/2019)

- Aclaramos que cada segmento en la Memoria tiene su correspondiente tabla de páginas
- Modificamos que la memoria para las tablas de páginas será alocada en tiempo de ejecución

v1.5 (30/5/2019)

- Eliminamos que dentro de las operaciones que se deben realizar al momento de realizar un INSERT en LFS se deba obtener la metadata de cada tabla
- Eliminamos que dentro de las acciones que se deben hacer cuando se realiza journaling se desactiven todos los flags de modificado de las distintas páginas
- Agregamos que cada vez que el Kernel realice un DESCRIBE contra Memoria actualice sus estructuras administrativas con los datos provistos por la misma

Índice

Objetivos y Normas de resolución

Objetivos del Trabajo Práctico

Mediante la realización de este trabajo se espera que el alumno:

- Adquiera conceptos prácticos del uso de las distintas herramientas de programación e interfaces (APIs) que brindan los sistemas operativos.
- Entienda aspectos del diseño de un sistema operativo.
- Afirme diversos conceptos teóricos de la materia mediante la implementación práctica de algunos de ellos.
- Se familiarice con técnicas de programación de sistemas, como el empleo de makefiles, archivos de configuración y archivos de log.
- Conozca con grado de detalle la operatoria de Linux mediante la utilización de un lenguaje de programación de relativamente bajo nivel como C.

Características

- Modalidad: grupal (5 integrantes +- 0) y obligatorio
- Tiempo estimado para su desarrollo: 90 días
- Fecha de comienzo: 30 de Marzo
- Fecha de primera entrega: 13 de Julio
- Fecha de segunda entrega: 20 de Julio
- Fecha de tercera entrega: 03 de Agosto
- Lugar de corrección: Laboratorio de Medrano

Evaluación del Trabajo Práctico

El trabajo práctico consta de una evaluación en 2 etapas.

La primera etapa consistirá en las pruebas de los programas desarrollados en el laboratorio. Las pruebas del trabajo práctico se subirán oportunamente y con suficiente tiempo para que los alumnos puedan evaluarlas con antelación. Queda aclarado que para que un trabajo práctico sea considerado evaluable, el mismo debe proporcionar registros de su funcionamiento de la forma más clara posible.

La segunda etapa se dará en caso de aprobada la primera y constará de un coloquio, con el objetivo de afianzar los conocimientos adquiridos durante el desarrollo del trabajo práctico y terminar de definir la nota de cada uno de los integrantes del grupo, por lo que se recomienda que la carga de trabajo se distribuya de la manera más equitativa posible.

Cabe aclarar que el trabajo equitativo no asegura la aprobación de la totalidad de los integrantes, sino que cada uno tendrá que defender y explicar tanto teórica como prácticamente lo desarrollado y aprendido a lo largo de la cursada.

La defensa del trabajo práctico (o coloquio) consta de la relación de lo visto durante la teoría con lo implementado. De esta manera, una implementación que contradiga a lo visto en clase ***es motivo de desaprobación del trabajo práctico***.

Deployment y Testing del Trabajo Práctico

Al tratarse de una plataforma distribuida, los procesos involucrados podrán ser ejecutados en diversas computadoras. La cantidad de computadoras involucradas y la distribución de los diversos procesos en estas será definida en cada uno de los tests de la evaluación y **es posible cambiar la misma en el momento de la evaluación**. Es responsabilidad del grupo automatizar el despliegue de los diversos procesos con sus correspondientes archivos de configuración para cada uno de los diversos tests a evaluar.

Todo esto estará detallado en el documento de pruebas que se publicará cercano a la fecha de Entrega Final. Archivos y programas de ejemplo se pueden encontrar en el repositorio de la cátedra.

Finalmente, recordar la existencia de las Normas del Trabajo Práctico donde se especifican todos los lineamientos de cómo se desarrollará la materia durante el cuatrimestre.

Aclaraciones

Debido al fin académico del trabajo práctico, los conceptos reflejados son, en general, versiones simplificadas o alteradas de los componentes reales de hardware y de sistemas operativos modernos, a fin de resaltar aspectos de diseño.

Invitamos a los alumnos a leer las notas y comentarios al respecto que haya en el enunciado, reflexionar y discutir con sus compañeros, ayudantes y docentes al respecto.

Abstract

El objetivo del trabajo práctico consiste en desarrollar una solución que permita la simulación de ciertos aspectos de una base de datos distribuida, es decir donde los distintos recursos del sistema puedan realizar una carga de trabajo ejecutando desde distintos puntos.

Dichos componentes incluidos dentro de la arquitectura del sistema deberán trabajar en conjunto para la planificación y ejecución de distintos scripts. Estos **scripts estarán conformados por múltiples Requests** (operaciones como leer valores, escribirlos, crear tablas, etc), los cuales se asemejan a las principales operaciones de una base de datos.

Los componentes del sistema serán:

- un kernel
- un pool de memorias
- un file system.

Los datos con los que se trabajará estarán organizados en estructuras de **tablas**, siendo los registros de estas del tipo **Key - Value**[1].

Los Request principales que le podremos solicitar al sistema son:

SELECT: Permite la obtención del valor de una key dentro de una tabla.

INSERT: Permite la creación y/o actualización de una key dentro de una tabla.

CREATE: Permite la creación de una nueva tabla dentro del file system.

DESCRIBE: Permite obtener la Metadata de una tabla en particular o de todas las tablas que el file system tenga.

DROP: Permite la eliminación de una tabla del file system.

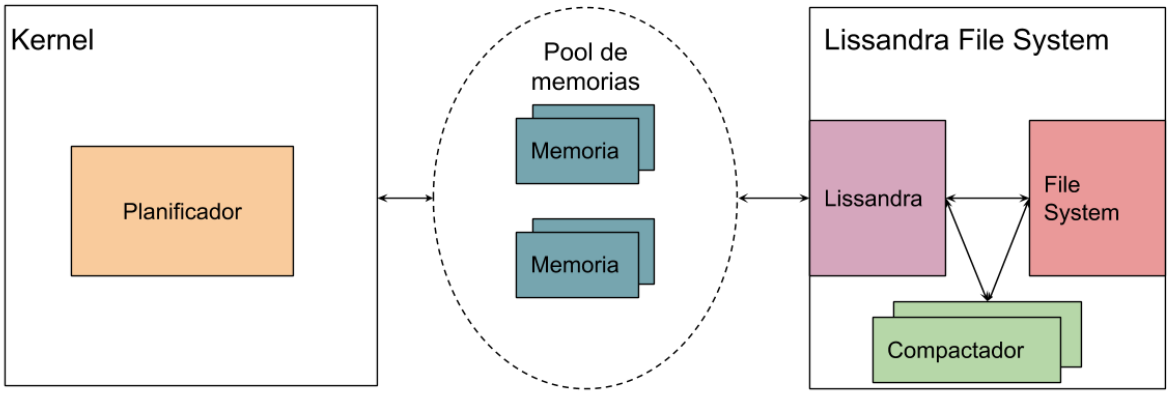
El sistema se evaluará no solo por su buen funcionamiento, sino también por la perfomance que logre distribuyendo la carga en los distintos módulos y el/los algoritmos que se aplique para manejar la consistencia y compactación de los datos.

Cuando hablamos de consistencia en el trabajo práctico nos referimos a la consistencia de los datos en general, es decir, asegurar que los datos se encuentran completos y se mantenido. De esta manera, un dato ingresado es consistente si es preciso, exacto y válido en el transcurso del tiempo.

[1] https://en.wikipedia.org/wiki/Key-value_database

Arquitectura del Sistema

Como se comentó en el punto anterior el sistema consta de tres grupos de procesos independientes: el **Kernel**, la **Memoria** y el **File System**. La interacción entre los mismos se da siguiendo el siguiente diagrama:



El Kernel, punto de partida de nuestro sistema, tendrá la función de recibir requests y distribuir la carga en las distintas memorias que él administre.

Dentro del sistema, existirá un pool de memorias que se comunicarán internamente entre ellas utilizando **gossiping** (ver Anexo III). Cada memoria será asignada dinámicamente en tiempo de ejecución a un criterio de elección (el sistema puede estar funcionando y nuevas memorias pueden unirse). La función de éstas será almacenar temporalmente los datos de los request que se le soliciten, manejando para esto una memoria principal.

El File System será el encargado de guardar físicamente los archivos del sistema. Cada uno de estos archivos estará asociados a una **Tabla** que poseerá distintos registros en forma de filas.

Para la prueba de cada módulo se dispondrá de una Consola/API. Dicha API ***deberá ser exactamente como indica este documento, no pudiendo alterarla***.

Como cátedra sugerimos que la carga de trabajo se distribuya equitativamente entre todos los integrantes del equipo. Adjuntamos la distribución que creemos óptima para la realización del trabajo práctico:

- Proceso Kernel: 1 Integrante
- Proceso Memoria: 2 Integrantes
- Proceso File System: 2 Integrantes

Esta distribución ***no asegura la aprobación***, sino que la misma está dada por la ***defensa del trabajo de cada integrante sobre lo realizado***.

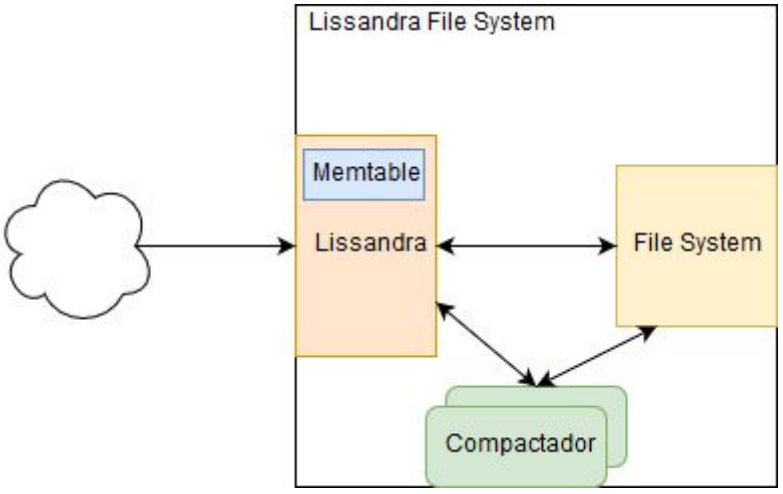
API

Una API (Application Programming Interface) se describe como el conjunto de funcionalidades y datos que cierto componente expone, es decir, el punto de comunicación entre el consumidor/cliente y el componente. Por ejemplo, la API que Google expone para que hagamos una búsqueda es el "pedido" que le realiza el navegador a su servidor, conteniendo su criterio de búsqueda.

La API de cada componente estará compuesta por los mensajes/pedidos que le podemos solicitar al mismo y que éste sabe cómo respondernos. Dentro del contexto del trabajo práctico la API de cada componente estará conformada por los request previamente mencionadas: SELECT, INSERT, CREATE, DESCRIBE y DROP.

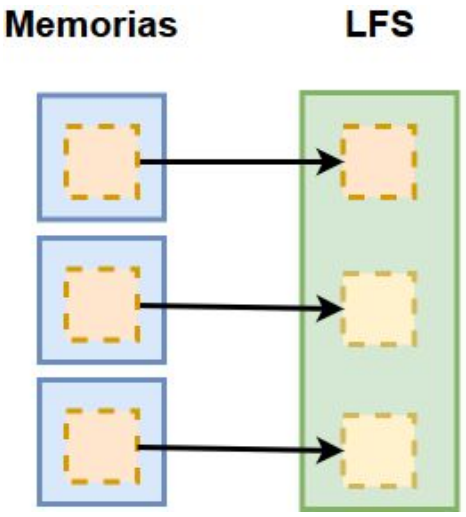
Proceso Lissandra File System (LFS)

El proceso Lissandra File System será el encargado de persistir y administrar las distintas tablas del sistema. Para esto, está conformado por distintos submódulos los cuales se representan en el siguiente gráfico de arquitectura:



El sub-módulo Lissandra será el punto de entrada al LFS, administrará la conexión con los demás módulos y entenderá como resolver cada uno de los distintos pedidos que se le soliciten.

Lissandra permitirá las conexiones de múltiples memorias al mismo tiempo y su forma de atenderlas será en paralelo, como se muestra en el siguiente gráfico:



El sub-módulo File System, será el encargado de resolver la persistencia de los archivos de las tablas en disco. Para ello, se dispondrá la especificación de un File System desarrollado especialmente para este trabajo práctico.

Por último, el sub-módulo Compactador será el encargado de realizar la compactación de las distintas tablas cuando el sistema lo requiera (este tema se explicará en el Anexo I).

El proceso LFS deberá poder atender múltiples peticiones en simultáneo. Bajo ningún punto podrá un request bloquear una tabla para el uso de otro request como tampoco el interbloqueo entre las mismas.

Dentro del sistema las tablas **únicamente** pueden quedar bloqueadas mientras se realiza el proceso de compactación (Ver Anexo 1).

Cualquier otra solución que no esté regulada bajo este punto, es ***motivo de desaprobación directa***.

Lissandra

Este sub-módulo se encargará de administrar las tablas físicas del sistema. Sabrá entender el contenido físico del FileSystem (los archivos de las tablas) y podrá transformar dicha información en la información que estas contienen. Por ejemplo, ante un SELECT sabrá cómo leer el/los archivos de una tabla y encontrar el valor de la Key solicitada.

Administración de Tablas

Debido a que trabajaremos con grandes volúmenes de datos y buscamos maximizar la disponibilidad y performance a la hora de leer y escribir los registros de una tabla, se dividirá la misma en distintas **particiones,** siendo cada una de estas un archivo en el FileSystem. La cantidad de particiones estará definida en la Metadata de cada tabla.

Cada tabla estará representada mediante un directorio, dentro del cual se encontrarán todos los archivos .bin que correspondan a cada una de sus particiones, como también su archivo de Metadata.

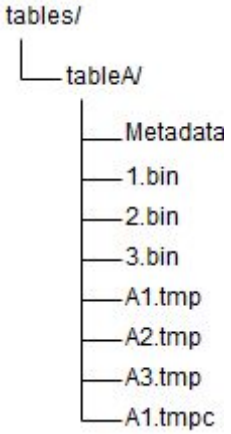
Para mantener la disponibilidad de una tabla, esta podría, dependiendo el momento, tener hasta dos tipos de archivos temporales:

1. Uno en el cual se escriban los datos que se van modificando para ser actualizados en los archivos originales en una próxima compactación.
2. Otro en el que se escriban los datos que se van modificando para ser actualizados en los archivos originales en una próxima compactación cuando la tabla ya está siendo actualizada mediante una compactación.

Para entender la diferencia entre estos dos archivos, recomendamos ver el flujo de compactación que se encuentra en el Anexo I.

Cabe aclarar que en estos archivos deberán también respetar la convención para almacenar datos en el FileSystem, no pudiendo guardar datos planos en los mismos (ver Aquí). Los datos ***siempre se guardarán dentro de los archivos de bloques*** (ver Aquí).

De esta manera, un ejemplo para el árbol de directorios/archivos del sistema en cierto momento podría resultar de esta forma:



Aclaración: La nomenclatura de las particiones y de los archivos temporales queda a definición de cada grupo, respetando siempre las extensiones (.tmp,.tmpc,.bin) y pudiendo diferenciarlos entre sí.

Por ej, los binarios podrían ser tanto “1.bin” como “particion_1.bin” como “part1.bin”.

Metadata

El archivo Metadata contendrá la información administrativa de la tabla. Esto implica que cada tabla estará configurada de una manera independiente y se administra internamente de distinta manera. Dicho archivo contiene la siguiente información:

- Tipo de consistencia: Este campo permite definir el nivel de criticidad de la tabla para el usuario final y el correspondiente nivel de exigencia para que se mantenga una consistencia rígida a la hora de obtener y grabar sus datos.

  Los tipos de consistencia pueden ser:

  1. Strong Consistency (SC)
  2. Strong Hash Consistency (SHC)
  3. Eventual Consistency (EC)

La explicación sobre el funcionamiento de estos valores se explicará tanto en el módulo de Planificación como en el Anexo II.

- Número de particiones: Número de particiones en la cual la tabla dividirá los datos. Las mismas se registrarán en los archivos .bin
- Tiempo entre compactaciones: Tiempo en milisegundos que debe pasar para validar si se requiere hacer una compactación.

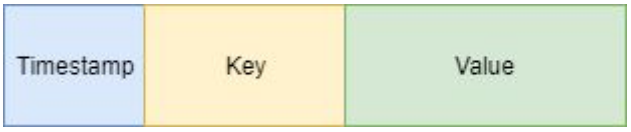
Ej:

```
CONSISTENCY=SC
PARTITIONS=4
COMPACTION_TIME=60000
```

Registro

Los datos de la tabla estarán formados por registros, donde cada uno de ellos estará compuesto por 3 partes:

1. Timestamp: Tiempo en milisegundos en el cual se insertó el registro.[2]
2. Key: Clave numérica que identifica al registro unívocamente.
3. Value: Valor o dato del registro.



- El Timestamp nos permite diferenciar inequívocamente cuál es el valor que tiene una Key. En el sistema pueden coexistir varios valores para una misma Key, y el Timestamp nos sirve para desambiguar dicho valor. Si se realizan varios INSERT sobre la misma Key se podrá determinar cuál es el último valor obteniendo el registro cuyo Timestamp sea el mayor.
- El Key será un campo numérico y no se proveerá una variante a esto.[3]
- Para la distribución de las distintas Key dentro de dicha tabla se aplicará una función módulo [4]
- El Value tendrá un valor máximo en bytes que estará indicado en el archivo de configuración del FileSystem. La aplicación tiene que ser responsable y administrar el error de un posible insert de un valor mayor al tamaño máximo.

Archivos temporales/Dumps[5]/Área de "memtable"

Dado un sistema distribuido, al momento de realizar una actualización de los datos, en el caso de que se modifiquen los datos originales y que los mismos se bloqueen esperando que el proceso de actualización finalice (por ej: no se podrá realizar una consulta sobre los mismos datos que están siendo actualizados), se producirá una pérdida de disponibilidad de los mismos, resultando en un sistema que no es de alta disponibilidad.

[2] Al igual que en el LFS, para calcular el TIMESTAMP usaremos el de Unix EPOCH.
[3] Para poder limitar el tamaño de las Key estaremos usando el tipo de dato uint16 para las mismas
[4] Se dividirá la key por la cantidad de particiones y el resto de la operación será la partición a utilizar
[5] Llamaremos dump a la acción de "descargar" datos que se encuentran en memoria a un archivo en el FS

Para evitar esto, la solución que vamos a utilizar dentro del trabajo práctico para garantizar la alta disponibilidad es el manejo de archivos temporales, en los cuales se guardarán estos datos que vayan modificándose para luego, en algún momento, consolidarlos (o compactarlos) con los originales.

Dispondremos dos jerarquías de datos temporales:

1. Datos en el área de "memtable"[6], es decir datos que aún no fueron dumpeados a un archivo temporal. Esta parte es una etapa previa a tener los datos en los archivos temporales que nos da velocidad a la hora de insertar un dato nuevo ya que se realizará en memoria y no directamente en un archivo del FileSystem.
2. Datos en archivos temporales que están pendientes de compactación.

De esta manera, cada vez que se desee realizar una actualización sobre un dato, el mismo se ***guardará en la memtable*** y cada X cantidad de tiempo (configurado por archivo de configuración) estos datos serán ***bajados a un archivo temporal por el proceso dump***. Este dumpeo consistirá ***en bajar toda la memtable*** (de todas las tablas) y copiar dichos datos a los distintos archivos temporales (uno por tabla). Una vez realizado este proceso, se limpiará la memtable para que pueda volver a llenarse con nuevos datos.

Así, podremos disponer de N archivos temporales en una tabla siendo N la cantidad de dumpeos realizados por el filesystem para dicha tabla.

Aquellos datos que aún no están dumpeados, y que se encuentren en la memtable, tendrán la cualidad de ser volátiles frente a una baja del proceso file system. Este punto es el único caso que se permite que los datos se pierdan.

Aclaración: En caso que se realice un nuevo Insert sobre una Key que ya se encuentra en la memtable, el Value no se actualizará, sino que se agregará el registro nuevamente teniendo "Keys duplicadas".

API

La API ***podrá ser ejecutada tanto desde consola como por sockets***. El objetivo de la API es permitir homogéneamente realizar verificaciones sobre este módulo. Por esta razón, ***la misma no puede ser modificada ni ampliada***. La misma atenderá las siguientes solicitudes:

- SELECT
- INSERT
- CREATE
- DESCRIBE
- DROP

Cuando la operación se realiza por consola, los valores que normalmente este módulo retorna por sockets deberá imprimirlos por pantalla.

[6] El área de "memtable" deberá estar dividido por tablas

SELECT

La operación Select permite la obtención del valor de una key dentro de una tabla. Para esto, se utiliza la siguiente nomenclatura:

```
SELECT [NOMBRE_TABLA] [KEY]
```

Ej:

```
SELECT TABLA1 3
```

Esta operación incluye los siguientes pasos:

1. Verificar que la tabla exista en el file system.
2. Obtener la metadata asociada a dicha tabla.
3. Calcular cual es la partición que contiene dicho KEY.
4. Escanear la partición objetivo, todos los archivos temporales y la memoria temporal de dicha tabla (si existe) buscando la key deseada.
5. Encontradas las entradas para dicha Key, se retorna el valor con el Timestamp más grande.

INSERT

La operación Insert permite la creación y/o actualización del valor de una key dentro de una tabla. Para esto, se utiliza la siguiente nomenclatura:

```
INSERT [NOMBRE_TABLA] [KEY] "[VALUE]" [Timestamp][7]
```

Ejemplos:

```
INSERT TABLA1 3 "Mi nombre es Lissandra" 1548421507
```

Ó

```
INSERT TABLA1 3 "Mi nombre es Lissandra" 1548421507
```

Esta operación incluye los siguientes pasos:

1. Verificar que la tabla exista en el file system. En caso que no exista, informa el error y continúa su ejecución.
2. Verificar si existe en memoria una lista de datos a dumpear. De no existir, alocar dicha memoria.
3. El parámetro Timestamp es **opcional**. En caso que un request no lo provea (por ejemplo insertando un valor desde la consola), se usará el valor actual del Epoch UNIX.
4. Insertar en la memoria temporal del punto anterior una nueva entrada que contenga los datos enviados en la request.

En los request solo se utilizarán las comillas ("") para enmascarar el Value que se envíe. No se proveerán request con comillas en otros puntos.

[7] Utilizaremos Unix EPOCH para calcular el Timestamp actual (y que sea una referencia única en todo el sistema). Para ver como calcularlo y obtenerlo, recomendamos leer:
https://www.systutorials.com/241697/how-to-get-the-epoch-timestamp-in-c/
Aclaración: la función que se muestra en el link devuelve el timestamp en microsegundos, no en milisegundos

De esta manera, cada insert se realizará siempre sobre la porción de memoria temporal asignada a dicha tabla sin importarle si dentro de la misma ya existe la key. Esto es así, ya que al momento de obtener la misma se retornará el que tenga un Timestamp más reciente mientras que el proceso de Compactación (explicado en el Anexo I), posterior al proceso de dump, será el que se encargue de unificar dichas Keys dentro del archivo original.

Todo dato dentro de un archivo será persistido con el formato:

```
[TIMESTAMP];[KEY];[VALUE]
```

No se dispondrá de keys ni Values con el carácter ";".

CREATE

La operación Create permite la creación de una nueva tabla dentro del file system. Para esto, se utiliza la siguiente nomenclatura:

```
CREATE [NOMBRE_TABLA] [TIPO_CONSISTENCIA] [NUMERO_PARTICIONES] [COMPACTION_TIME]
```

Ej:

```
CREATE TABLA1 SC 4 60000
```

Esta operación incluye los siguientes pasos:

1. Verificar que la tabla no exista en el file system. Por convención, una tabla existe si ya hay otra con el mismo nombre. Para dichos nombres de las tablas siempre tomaremos sus valores en UPPERCASE (mayúsculas). En caso que exista, se guardará el resultado en un archivo .log y se retorna un error indicando dicho resultado.
2. Crear el directorio para dicha tabla.
3. Crear el archivo Metadata asociado al mismo.
4. Grabar en dicho archivo los parámetros pasados por el request.
5. Crear los archivos binarios asociados a cada partición de la tabla y asignar a cada uno un bloque

DESCRIBE

La operación Describe permite obtener la Metadata de una tabla en particular o de todas las tablas que el File System tenga. Para esto, se utiliza la siguiente nomenclatura:

```
DESCRIBE [NOMBRE_TABLA]
```

Ej:

```
DESCRIBE

DESCRIBE TABLA1
```

Para el primer caso la operación incluye los siguientes pasos:

1. Recorrer el directorio de árboles de tablas y descubrir cuales son las tablas que dispone el sistema.
2. Leer los archivos Metadata de cada tabla.

3. Retornar el contenido de dichos archivos Metadata.

Para el segundo caso la operación incluye los siguientes pasos:

1. Verificar que la tabla exista en el file system.
2. Leer el archivo Metadata de dicha tabla.
3. Retornar el contenido del archivo.

DROP

La operación Drop permite la eliminación de una tabla del file system. Para esto, se utiliza la siguiente nomenclatura:

```
DROP [NOMBRE_TABLA]
```

Ej:

```
DROP TABLA1
```

Esta operación incluye los siguientes pasos:

1. Verificar que la tabla exista en el file system.
2. Eliminar directorio y todos los archivos de dicha tabla.

File System

El FileSystem es un componente creado con propósitos académicos para que el alumno comprenda el funcionamiento básico de la gestión de archivos en un sistema operativo.

La estructura básica del mismo se basa en una estructura de árbol de directorios para representar la información administrativa y los datos de las tablas en formato de archivos. El árbol de directorios tomará su punto de partida del punto de montaje del archivo de configuración.

Durante las pruebas no se proveerán archivos que tengan estados inconsistentes respecto del trabajo práctico, por lo que no es necesario tomar en cuenta dichos casos.

Metadata

Este archivo tendrá la información correspondiente a la cantidad de bloques y al tamaño de los mismos dentro del File System.

Dentro de cada archivo se encontrarán los siguiente campos:

- **Block_size**: Indica el tamaño en bytes de cada bloque
- **Blocks**: Indica la cantidad de bloques del File System
- **Magic_Number**: Un string fijo con el valor “LISSANDRA”

Ej:

```
BLOCK_SIZE=64
BLOCKS=5192
MAGIC_NUMBER=LISSANDRA
```

Dicho archivo deberá encontrarse en la ruta `[Punto_Montaje]/Metadata/Metadata.bin`

Bitmap

Este será un archivo de tipo binario donde solamente existirá un bitmap[8], el cual representará el estado de los bloques dentro del FS, siendo un 1 que el bloque está ocupado y un 0 que el bloque está libre.

La ruta del archivo de bitmap es: `[Punto_Montaje]/Metadata/Bitmap.bin`

Table Metadata

Las tablas dentro del FS se encontrarán en un path compuesto de la siguiente manera:

`[Punto_Montaje]/Tables/[Nombre_Tabla]`

Donde el path del archivo incluye tanto el archivo Metadata como las distintas particiones de cada tabla y los archivos temporales.

Ej:

```
/mnt/FS_LISSANDRA/Tables/Tabla1/Metadata

/mnt/FS_LISSANDRA/Tables/Tabla1/1.bin

/mnt/FS_LISSANDRA/Tables/Tabla1/A.tmp
```

Dentro de cada archivo (que no es Metadata) se encontrarán los siguientes campos:

- **Size**: indica el tamaño real del archivo en bytes.
- **Blocks**: es un array de números que contiene el orden de los archivos en donde se encuentran los datos propiamente dichos de ese archivo

Ej:

```
SIZE=250
BLOCKS=[40,21,82,3]
```

Datos

Los datos estarán repartidos en archivos de texto nombrados con un número, el cual representará el número de bloque. (Por ej 1.bin, 2.bin, 3.bin),

Dichos archivos se encontraran dentro de la ruta:

`[Punto_Montaje]/Bloques/[nroBloque].bin`

Ej:

```
/mnt/FS_LISSANDRA/Bloques/1.bin

/mnt/FS_LISSANDRA/Bloques/2.bin
```

Archivo de Configuración y Logs

El proceso deberá poseer un archivo de configuración en una ubicación conocida donde se deberán especificar, al menos, los siguientes parámetros:

Campo	Tipo	Descripción

[8] Se recomienda investigar sobre el manejo de los bitarray de las commons library.

PUERTO_ESCUCHA	[numérico]	Puerto TCP utilizado para recibir las conexiones de CPU y I/O
PUNTO_MONTAJE	[alfanumérico]	Valor del punto de montaje inicial de File System, a partir de este punto se creará el árbol para administrar el FS.
RETARDO	[numérico - milisegundos]	Valor del Retardo de cada operación realizada.
TAMAÑO_VALUE	[numérico]	Máximo tamaño de un value en bytes
TIEMPO_DUMP	[numérico - milisegundos]	Tiempo cada cuanto se realiza el proceso de dump

Queda a decisión del grupo el agregado de más parámetros al mismo. Además, ***el proceso deberá registrar toda su actividad mediante un archivo de log***. **Los campos "RETARDO" y "TIEMPO_DUMP" deberán poder ser modificados en tiempo de ejecución**, actualizando la operatoria del sistema.[9]

Ejemplo de Archivo de Configuración

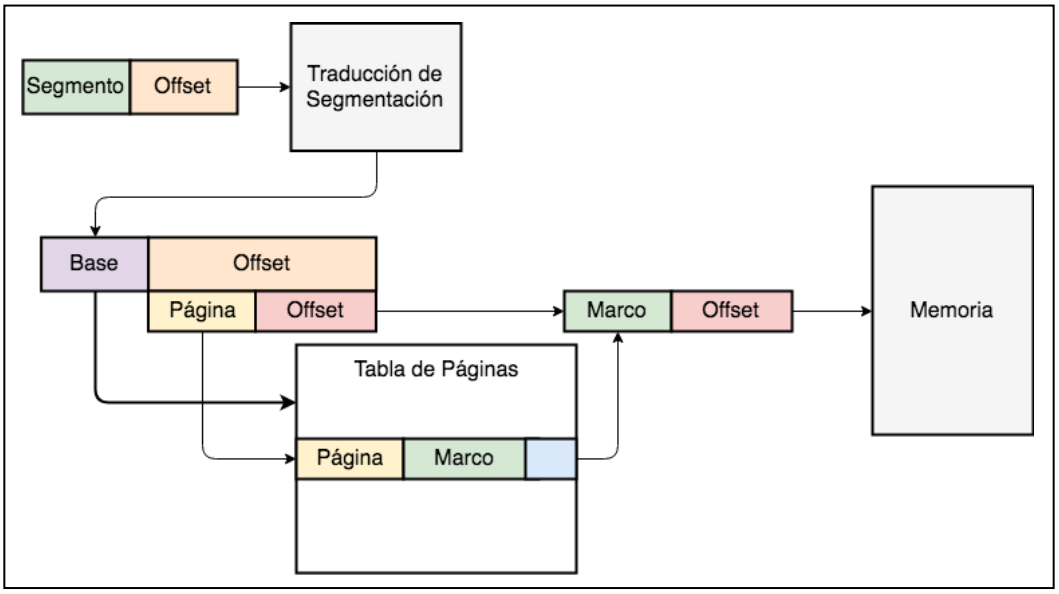
```
PUERTO_ESCUCHA=5003
PUNTO_MONTAJE="/mnt/LISSANDRA_FS/"
RETARDO=500
TAMAÑO_VALUE=4
TIEMPO_DUMP=5000
```

[9] Investigar inotify()

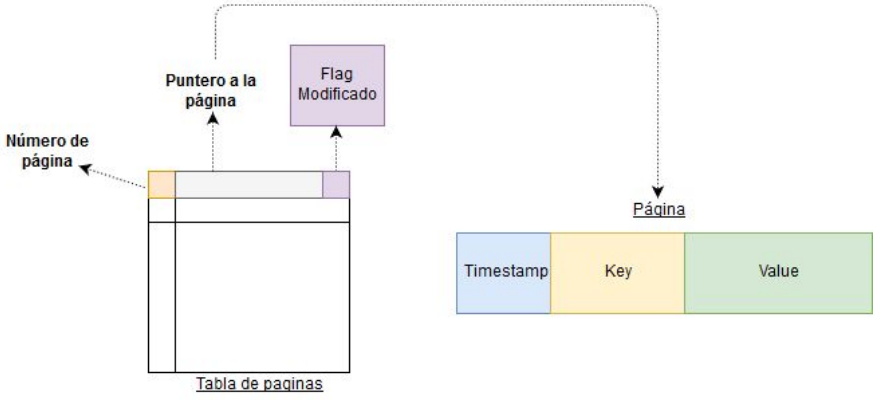
Proceso Memoria

El proceso memoria cumplirá la función de disponibilizar los datos requeridos para la obtención y actualización que los distintos request requieran de manera concurrente. Todas aquellas solicitudes que le podíamos pedir al LFS también se las podremos solicitar a la memoria, pero con una ventaja: los request en memoria serán más rápidos que aquellos que tengan que pasar por el file system.

Cada proceso memoria asumirá el concepto de Localidad Temporal[10]. Para administrar el contenido de los datos se utilizará el concepto de Segmentación Paginada (el cual será visto en la parte teórica de la materia). Cada segmento estará identificado por el path de la tabla a la cual pertenece.



De esta manera, se define que cada segmento es una tabla y que estas últimas están paginadas. La paginación se administra de manera que cada página es un registro de dicha tabla (sabiendo que el value tiene un maximo tamaño configurado en el File System y que la memoria debe conocer al momento de inicializarse).



[10] El concepto de localidad temporal infiere que los datos que se acceden en la memoria recientemente tienen una alta probabilidad de volver a ser accedidas en el futuro cercano. De esta manera, almacenar estos datos en una memoria ultrarrápida permite mejorar los tiempos de respuesta del sistema.

Dado que un segmento corresponde a una tabla del FS y dentro de las request posibles tenemos el “INSERT”, las tablas pueden ser modificadas dentro de la memoria. Para identificar aquellas páginas que fueron modificadas utilizaremos el concepto “Flag de Modificado”. Se sabrá entonces que aquellas páginas cuyo Flag de Modificado esté activado tienen un estado más actualizado que la tabla del FS.

Los cambios realizados sobre las tablas que se encuentren en memoria, por los distintos request, luego deberán verse impactados en el LFS, mediante el proceso al que llamamos Journaling.

Start Up

Al momento de iniciar el proceso memoria se deben realizar los siguientes pasos:

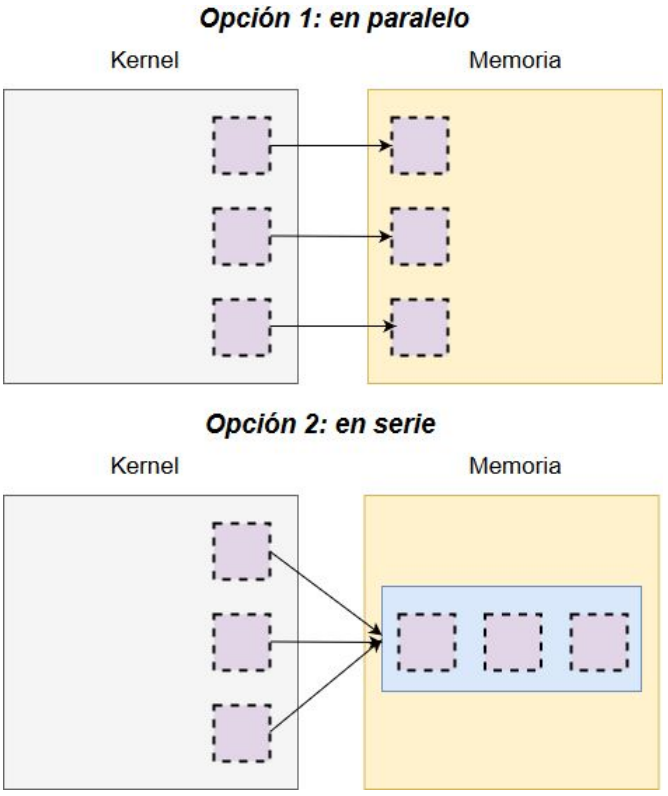
1. Conectarse al proceso File System y realizar handshake necesario para obtener los datos requeridos. Esto incluye el tamaño máximo del Value configurado para la administración de las páginas.
2. Inicializar la memoria principal (que se explican en los siguientes apartados).
3. Iniciar el proceso de Gossiping (explicado en profundidad en el Anexo III) que consiste en la comunicación de cada proceso memoria con otros procesos memorias, o seeds, para intercambiar y descubrir otros procesos memorias que se encuentren dentro del pool (conjunto de memorias).

En caso que no pueda realizar alguna de las dos primeras operaciones, se deberá abortar el proceso memoria informando cuál fue el problema.

Memoria Principal

La memoria principal será la responsable de almacenar temporalmente aquellos datos que son solicitados y/o modificados por los distintos request del sistema. Podrá recibir múltiples conexiones o solicitudes del Kernel, y además, deberá responder pedidos desde la terminal del sistema. También se conectará con otras memorias para intercambiar información.

Podremos utilizar tanto múltiples threads (opción 1) como multiplexación -*select*- (opción 2) para poder recibir múltiples pedidos de sus clientes. **La elección del método utilizado será evaluada en el coloquio.**



Para funcionar tendrá los siguientes componentes:

- Tabla de segmentos: Estructura administrativa que contiene el listado de los segmentos que contiene la memoria principal.
- Tabla de páginas: Estructura administrativa que contiene el listado de páginas para cada segmento. Cada segmento tendrá su correspondiente tabla de páginas.
- Memoria principal: Memoria contigua donde se alojan las distintas páginas de los segmentos utilizados. Cualquier otra solución que no sea memoria contigua es ***motivo de desaprobación directa***.

Cabe aclarar que todos los componentes deben ser inicializados ***por única vez al iniciar el proceso, reservando en este momento la memoria necesaria para la memoria principal. La memoria que se necesite para la tabla de segmentos y las correspondientes tablas de páginas de cada uno de los segmentos se irán reservando dinámicamente en tiempo de ejecución.***

Para administrar el reemplazo de las distintas páginas en una memoria principal se utilizará el algoritmo LRU (Least Recently Used, el cual será visto en la parte teórica de la materia). Se debe tener en cuenta que la página a reemplazar no debe tener el Flag de Modificado activado. Esto es, no debe contener datos que aún no han sido impactados en el file system.

En base a lo indicado en el punto anterior puede darse la ***situación en que una memoria requiere asignar una nueva página a un segmento sin disponer libres y a su vez posea todos sus segmentos actuales con todas sus páginas en estado Modificado***. Cuando esto suceda se dice que una memoria está ***FULL (completa)*** y deberá comenzar su proceso de Journal. Una memoria en este estado no responderá ningún pedido que implique una operación que requiera el reemplazo de uno

de los segmentos actuales. Una vez finalizado el proceso de Journal, la memoria volverá a aceptar los pedidos que se le realicen.

El proceso de Journal (explicado en profundidad en el Anexo IV) es el encargado de informar al File System todos los cambios realizados en la memoria principal para luego vaciar completamente la misma. Este proceso tiene dos modos de ejecución: uno por medio de la API y otro automático que se ejecuta cada X unidades de tiempo (este parámetro se obtiene desde el archivo de configuración).

API

La API responderá a los mismos comandos utilizados en el File System más uno propio:

- SELECT
- INSERT
- CREATE
- DESCRIBE
- DROP
- JOURNAL

SELECT

La operación Select permite la obtención del valor de una key dentro de una tabla. Para esto, se utiliza la siguiente nomenclatura:

```
SELECT [NOMBRE_TABLA] [KEY]
```

Ej:

```
SELECT TABLA1 3
```

Esta operación incluye los siguientes pasos:

1. Verifica si existe el segmento de la tabla solicitada y busca en las páginas del mismo si contiene key solicitada. Si la contiene, devuelve su valor y finaliza el proceso.
2. Si no la contiene, envía la solicitud a FileSystem para obtener el valor solicitado y almacenarlo.
3. Una vez obtenido se debe solicitar una nueva página libre para almacenar el mismo. En caso de no disponer de una página libre, se debe ejecutar el algoritmo de reemplazo y, en caso de no poder efectuarlo por estar la memoria full, ejecutar el Journal de la memoria.

INSERT

La operación Insert permite la creación y/o actualización de una key dentro de una tabla. Para esto, se utiliza la siguiente nomenclatura:

```
INSERT [NOMBRE_TABLA] [KEY] “[VALUE]”
```

Ej:

```
INSERT TABLA1 3 “Mi nombre es Lissandra”
```

Esta operación incluye los siguientes pasos:

1. Verifica si existe el segmento de la tabla en la memoria principal. De existir, busca en sus páginas si contiene la key solicitada y de contenerla actualiza el valor insertando el Timestamp actual. En caso que no contenga la Key, se solicita una nueva página para almacenar la misma. Se deberá tener en cuenta que si no se disponen de páginas libres aplicar el algoritmo de reemplazo y en caso de que la memoria se encuentre full iniciar el proceso Journal.
2. En caso que no se encuentre el segmento en memoria principal, se creará y se agregará la nueva Key con el Timestamp actual, junto con el nombre de la tabla en el segmento. Para esto se debe generar el nuevo segmento y solicitar una nueva página (aplicando para este último la misma lógica que el punto anterior).

Cabe destacar que **la memoria no verifica la existencia de las tablas al momento de insertar nuevos valores**. De esta forma, no tiene la necesidad de guardar la metadata del sistema en alguna estructura administrativa. En el caso que al momento de realizar el Journaling una tabla no exista, deberá informar por archivo de log esta situación, pero el proceso deberá actualizar correctamente las tablas que *sí* existen.

Cabe aclarar que esta operatoria en ningún momento hace una solicitud directa al FileSystem, la misma deberá manejarse siempre dentro de la memoria principal.

CREATE

La operación Create permite la creación de una nueva tabla dentro del file system. Para esto, se utiliza la siguiente nomenclatura:

```
CREATE [TABLA] [TIPO_CONSISTENCIA] [NUMERO_PARTICIONES] [COMPACTION_TIME]
```

Ej:

```
CREATE TABLA1 SC 4 60000
```

Esta operación incluye los siguientes pasos:

1. Se envía al FileSystem la operación para crear la tabla.
2. Tanto si el FileSystem indica que la operación se realizó de forma exitosa o en caso de falla por tabla ya existente, continúa su ejecución normalmente.

DESCRIBE

La operación Describe permite obtener la Metadata de una tabla en particular o de todas las tablas que el file system tenga. Para esto, se utiliza la siguiente nomenclatura:

```
DESCRIBE [NOMBRE_TABLA]
```

Ej:

```
DESCRIBE
```

```
DESCRIBE TABLA1
```

Esta operación consulta al FileSystem por la metadata de las tablas. Sirve principalmente para poder responder las solicitudes del Kernel.

DROP

La operación Drop permite la eliminación de una tabla del file system. Para esto, se utiliza la siguiente nomenclatura:

```
DROP [NOMBRE_TABLA]
```

Ej:

```
DROP TABLA1
```

Esta operación incluye los siguientes pasos:

1. Verifica si existe un segmento de dicha tabla en la memoria principal y de haberlo libera dicho espacio.
2. Informa al FileSystem dicha operación para que este último realice la operación adecuada.

JOURNAL

La operación Journal permite el envío manual de todos los datos de la memoria al FileSystem. Para esto, se utiliza la siguiente nomenclatura:

```
JOURNAL
```

Para más información de cómo funciona el Journal se recomienda la lectura del Anexo IV. Cuando una memoria se encuentra procesando o realizando un Journal a Disco no se podrá operar ningún request que le llegue a la memoria, debiendo esperar a que el primero finalice.

Como se mencionó anteriormente, cabe aclarar que cada memoria tendrá dos formas de realizar un Journal:

1. Manualmente por el comando recién explicado, que puede ser tanto ejecutado por consola como a pedido del Kernel cuando se cumple la condición de que la memoria se encuentra en estado Full (el mismo se explicará en detalle en la descripción del Kernel)
2. Automáticamente configurado cada X cantidad de tiempo (donde X es el tiempo configurado por archivo de configuración en milisegundos).

Hasta que no se realice uno de estos dos procesos, el FileSystem no tendrá conocimiento ni actualizará los datos modificados o agregados que cada memoria contenga.

Pool de Memorias

Una vez explicado el funcionamiento de una memoria independiente, nos encontramos con el problema de que una sola de ellas no es suficiente para soportar una alta demanda de pedidos y por lo tanto tampoco hará que nuestro sistema funcione lo suficientemente rápido.

Por esta razón, se tendrán múltiples instancias de los procesos Memorias funcionando en simultáneo donde cada una funcionará independiente de otra. Dicho esto, se utilizará la técnica de Gossiping para el descubrimiento de memorias entre ellas (explicado en profundidad en el Anexo III). Esta técnica se ejecutará temporalmente cada X unidades de tiempo configuradas por archivo de configuración. A su vez, para su implementación, toda memoria conocerá por lo menos a una más (que estará configurada por archivo de configuración) a las cuales llamaremos ***seeds* (o *semillas)***.

Ahora, si cada una funciona independientemente de otra y dos memorias pueden trabajar sobre la misma tabla, esto genera una posible condición de carrera (concepto a ver en la parte teórica de la materia) sobre los datos modificados y obtenidos. Esto se traduce en poder llegar a obtener un valor inconsistente al consultarlo a una memoria en particular cuando se actualizó recientemente en otra memoria. ***Esto es algo que puede pasar y estamos dispuestos a tomar como riesgo, ya que al enviar los datos al LFS siempre terminará impactándose el último valor actualizado, obtenido a partir de la comparación por Timestamp***.

Para administrar este último punto es que cada tabla tendrá distintos tipos de consistencia para poder verificar y validar los distintos casos. Para comprender el funcionamiento de los distintos tipos de consistencia se aconseja leer el proceso de Kernel y el Anexo II.

Por último, el sistema en general debe soportar la desconexión de una memoria. En caso que suceda, el proceso de gossiping deberá identificar ésto y al enterarse el kernel quitar dicha memoria de los criterios que la utilizaban.

Archivo de Configuración y Logs

El proceso deberá poseer un archivo de configuración en una ubicación conocida donde se deberán especificar, al menos, los siguientes parámetros:

Campo	Tipo	Ejemplo
Puerto de Escucha	[numérico]	`8000`
IP del File System	[cadena]	`“192.168.1.2”`
Puerto del File System	[numérico]	`8001`
IP de Seeds	[array - cadena]	`[“192.168.1.3”, “192.168.1.4”]`
Puertos de Seeds	[array - numérico]	`[8000,8001]`
Retardo de acceso a Memoria Principal	[numérico - milisegundos]	`600`
Retardo de acceso a File System	[numérico - milisegundos]	`600`
Tamaño de la Memoria	[numérico]	`2048`
Tiempo de Journal	[numérico - milisegundos]	`60000`
Tiempo de Gossiping	[numérico - milisegundos]	`30000`
Número de memoria	[numérico]	`1`

Queda a decisión del grupo el agregado de más parámetros al mismo. Además, el proceso deberá registrar toda su actividad mediante un archivo de log. **Los campos de retardo deberán poder ser**

modificados en tiempo de ejecución, actualizando la operatoria del sistema (excepto los relacionados a conexiones).[11]

Ejemplo de Archivo de Configuración

```
PUERTO=8000
IP_FS="192.168.1.2"
PUERTO_FS=8001
IP_SEEDS=["192.168.1.3","192.168.1.4"]
PUERTO_SEEDS=[8000,8001]
RETARDO_MEM=600
RETARDO_FS=600
TAM_MEM=2048
RETARDO_JOURNAL=60000
RETARDO_GOSSIPING=30000
MEMORY_NUMBER=1
```

[11] Investigar inotify()

Proceso Kernel

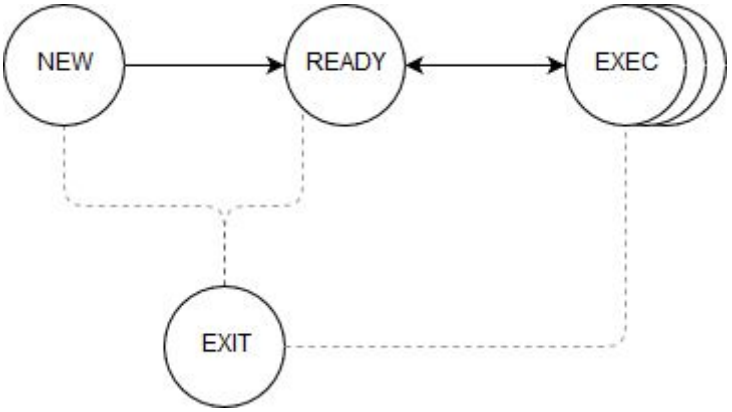
El proceso Kernel, como fue descrito ya anteriormente dentro de este trabajo práctico, será el punto de entrada a nuestro sistema. Él tendrá la responsabilidad de planificar las distintas request que se quieran efectuar, así como también la administración del pool de memorias y distribución de los pedidos entre las mismas.

Las request pueden llegar a este componente de dos formas posibles:

1. Archivos LQL (Lissandra Query Language)
2. Request unitarias realizadas mediante la API de este componente, las cuales serán consideradas como archivos LQL de una única línea

Planificación

Para la planificación de los distintos scripts/requests utilizaremos un diagrama de estados similar al visto en la parte teórica de la materia.



Al tener la capacidad de trabajar con varios scripts al mismo tiempo en sus memorias, el sistema permitirá que haya **multiprocesamiento, es decir, múltiples scripts al mismo tiempo en estado "Exec"**. El grado de multiprocesamiento estará dado por una entrada en el archivo de configuración, que será independiente de las memorias que tenga conectadas.

Al ingresar un nuevo script al sistema el mismo se ubicará en la cola de "New", se crearán las estructuras administrativas necesarias para su ejecución y pasará al estado "Ready".

La planificación de los scripts será realizada bajo el algoritmo Round Robin con un Quantum configurable desde su archivo de configuración. El script elegido para ejecutarse podrá ir a cualquiera de los N estados de "Exec" disponibles, volviendo a la cola de Ready una vez que su quantum haya finalizado.

Aclaración: Dentro de nuestro trabajo práctico consideraremos que cada request (una línea del script) ocupará una sola unidad de ejecución (quantum). Por ejemplo: si el RR es de Q=4 cada script en el estado Exec podrá ejecutar 4 líneas para luego volver a la cola de Ready nuevamente.

Los scripts pasarán a estado "Exit" una vez que fueron ejecutadas todas sus request o que haya fallado la ejecución en una de ellas, sin ejecutar las posteriores en caso de haberlas. Es decir: Si

consideramos un script de 10 request donde la request 3 falla el mismo entrará en estado “Exit” sin ejecutar las 7 líneas restantes.

Es importante destacar que al intentar ejecutar una request la **memoria elegida puede estar en estado FULL o realizando Journaling**. En este caso, se deberá esperar a que la memoria **termine de vaciarse**, para luego resolver el request (si se encuentra FULL, se deberá forzar un JOURNAL).

Este sistema no contempla el bloqueo de los scripts, por lo que no contamos con el estado “Bloqueado”.

Ciclo de ejecución

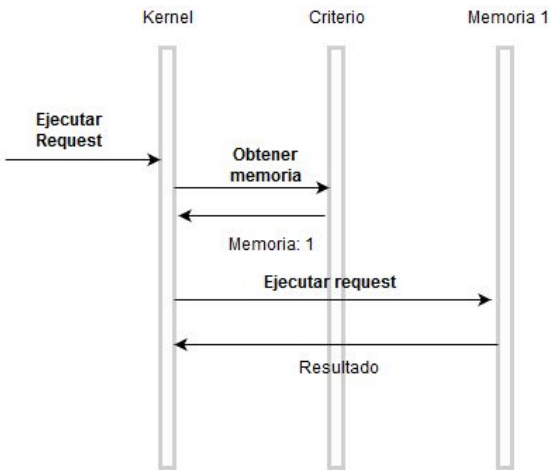
Una vez que un script ingresa a un estado de Exec el Kernel se encargará entonces de leer y parsear la siguiente línea y mandarla a ejecutar en alguna de las memorias del pool.

Para saber a qué memoria deberá enviar cada request el Kernel utilizará un **criterio** distinto dependiendo del tipo de consistencia de la tabla que se requiera. Cada criterio tendrá asociada una o varias memorias dentro de las cuales deberá escoger la más apropiada.

Al momento de inicializarse el Kernel, solicitará a una memoria (configurada por archivo de configuración) ***cuáles son las memorias que componen el pool*** (cada memoria conocerá al pool por medio de la técnica de gossiping, Anexo III).

Una vez realizado ésto, el kernel conocerá todas las memorias activas dentro del pool y utilizará el comando *ADD* para asociar dichas memorias a los distintos criterios. Esto quiere decir que las memorias del pool serán utilizables para la ejecución de las requests si y sólo si fueron asociadas a algún criterio.

Para mantener la metadata de las tablas actualizada el kernel llevará registro de cuáles son las tablas creadas en el sistema en una estructura administrativa. Esta estructura se cargará luego de haberse conectado a alguna memoria haciendo un DESCRIBE global a la misma. Luego, esta estructura se actualizará cada un tiempo obtenido por archivo de configuración realizando nuevos DESCRIBE globales a cualquier memoria que tenga conectada.



Por último, al finalizar el ciclo de ejecución se deberá contemplar un tiempo retardo indicado desde archivo de configuración. Este retardo será utilizado a fines prácticos para la evaluación del TP.

Aclaración: Una memoria ***puede ser asignada a múltiples criterios.***

Criterios

Como dijimos anteriormente el Kernel tendrá distintos criterios para la elección de la memoria a utilizar dependiendo el nivel de consistencia que cada tabla requiera. Ésto permitirá poder realizar correctamente los pedidos tanto de tablas cuya consistencia es fuerte como también de las cuales posean una consistencia eventual (concepto explicado en el Anexo II).

Para esto, se dispondrán tres criterios:

- Criterio Strong Consistency
- Criterio Strong-Hash Consistency
- Criterio Eventual Consistency

Criterio Strong Consistency

El criterio Strong Consistency será aquel que garantice que sobre todo pedido realizado se obtenga el valor real que contiene la Key en el FS (ya sea que aun el dato se encuentre en memoria o que ya se haya actualizado en disco).

Para esto, dicho criterio requerirá para su funcionamiento y **solo dispondrá de una memoria asignada a él.** (El por qué se explicará en profundidad en el Anexo II)

En resumen: al tener una sola memoria, la memoria elegida para resolver todas las request será siempre la misma.

Criterio Strong-Hash Consistency

Este criterio funcionará con el mismo concepto de garantizar que todo pedido realizado se obtenga el valor real que terminará impactado en el disco pero con la diferencia que el mismo **podrá utilizar múltiples memorias**.

Al llegar un pedido a dicho criterio aplicará una función de Hash sobre la key que definirá a qué memoria deberá ir dicho pedido. De ésta manera, cada vez que se realice un pedido sobre dicha key

(independientemente de la tabla) siempre se redireccionará el pedido a una memoria en particular y garantizará que el pedido sea consistente.

Dicha función de hash deberá garantizar la correcta distribución en las distintas memorias que se dispongan.

Cada vez que una memoria se agregue a dicho criterio se deberá realizar un Journal sobre todas las memorias asociadas al mismo para garantizar que las keys se mantengan en las memorias correctas.

Sobre la función de hash, cabe aclarar que la cátedra orientará sobre cómo debe realizarse, pero la misma deberá ser ***única***, ***exclusiva*** y de ***desarrollo propio***.

Criterio Eventual Consistency

En éste criterio el valor devuelto podría no ser exactamente el que tenga el Timestamp más alto registrado en el sistema. Esto sucede porque al insertar un valor en una memoria y consultarlo en otra, esta última podría no haber actualizado aún sus valores (este proceso se explicará en profundidad en el Anexo II).

En la actualidad este tipo de consistencia se utiliza para modelar situaciones en donde no siempre es importante tener el último valor disponible, o en cargas con mucha cantidad de INSERTS y pocos SELECTS. Por ejemplo, *Twitter podría vivir sin mostrarte el último Tweet de Adro hace menos de un segundo. Pero* ***eventualmente*** *(seguro luego de unos segundos/minutos) va a llegar a tu feed.*

Para utilizar este criterio **el planificador podrá elegir cualquier memoria que se encuentre asociada al criterio**. El alumno podrá utilizar algún algoritmo como Round Robin, utilizar la memoria que hace más tiempo se usó, la que menos pedidos realizó hasta ahora o simplemente alguna función de random[12] para elegir qué memoria utilizar. La única condición a aplicar es que no se envíe siempre el request a la misma memoria (en el caso de haber muchas) y que todas las memorias asociadas a este criterio participen.

Métricas

Por cada solicitud a una memoria se debe mantener una estadística interna ***de cada criterio*** que indique:

- **Read Latency / 30s:** El tiempo promedio que tarda un SELECT en ejecutarse en los últimos 30 segundos.
- **Write Latency / 30s:** El tiempo promedio que tarda un INSERT en ejecutarse en los últimos 30 segundos.
- **Reads / 30s:** Cantidad de SELECT ejecutados en los últimos 30 segundos.
- **Writes / 30s:** Cantidad de INSERT ejecutados en los últimos 30 segundos.
- **Memory Load (por cada memoria):** Cantidad de INSERT / SELECT que se ejecutaron en esa memoria respecto de las operaciones totales.

Estas métricas deben poder ser consultadas por consola y logueadas cada 30 segundos.

API

La API responderá a los mismos comandos utilizados en la Memoria más dos propio:

[12] https://www.programmingsimplified.com/c-program-generate-random-numbers

- SELECT
- INSERT
- CREATE
- DESCRIBE
- DROP
- JOURNAL
- ADD
- RUN
- METRICS

Los primeros cinco comandos cuando sean solicitados al Kernel serán directamente redireccionados a la memoria elegida por el criterio de la tabla en cuestión.

En caso que se intente realizar un INSERT/SELECT/DROP sobre una tabla de la cual el Kernel no tiene conocimiento dentro de la metadata que conoce, entonces se deberá informar el error por archivo de log y finalizar la ejecución del script.

Al realizar un DESCRIBE el Kernel deberá actualizar sus estructuras administrativas que contengan la metadata de las tablas con lo informado por la Memoria.

JOURNAL

En el caso de recibir un JOURNAL, enviará un JOURNAL a **cada una de las memorias que tiene asociadas a sus criterios.**

RUN

El comando RUN <path> nos permitirá ejecutar un archivo LQL, siendo <path> la ruta del mismo. **Este archivo se encontrará en el FileSystem local de la máquina del Kernel y NO en el proceso LFS del TP.**

METRICS

El comando METRICS informará las métricas actuales por consola.

ADD

El comando ADD será el encargado de asignar una memoria a un criterio en concreto. Cabe aclarar que varios criterios pueden utilizar la misma memoria sin inconvenientes. Esta asignación es una asignación lógica/virtual para indicar que dicho criterio puede utilizar la memoria correspondiente para elegir a la hora de redireccionar un request.

La sintaxis del mismo es la siguiente:

```
ADD MEMORY [NÚMERO] TO [CRITERIO]
```

Ej:

```
ADD MEMORY 3 TO SC
```

Cabe destacar que la nomenclatura utilizada para el parámetro [Criterio] es decisión de la implementación de cada grupo por lo que no se solicitará valores específicos.

Archivo de Configuración y Logs

El proceso deberá poseer un archivo de configuración en una ubicación conocida donde se deberán especificar, al menos, los siguientes parámetros:

Campo	Tipo	Ejemplo
IP de memoria	[cadena]	“192.168.1.2”
Puerto de la memoria	[numérico]	8001
Quantum	[numérico]	4
Multiprocesamiento	[numérico]	3
Refresh Metadata	[numérico]	10000
Retardo ciclo de ejecución	[numérico - milisegundos]	5000

Queda a decisión del grupo el agregado de más parámetros al mismo. Además, el proceso deberá registrar toda su actividad mediante un archivo de log. **Los campos de retardo (SLEEP, METADATA_REFRESH) y QUANTUM deberán poder ser modificados en tiempo de ejecución**, actualizando la operatoria del sistema (excepto los relacionados a conexiones).[13]

Ejemplo

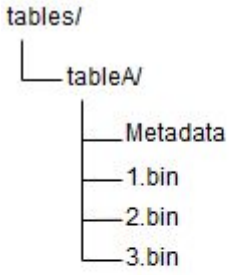
```
IP_MEMORIA="192.168.1.2"
PUERTO_MEMORIA=8001
QUANTUM=4

MULTIPROCESAMIENTO=3
METADATA_REFRESH=10000
SLEEP_EJECUCION=5000
```

[13] Investigar inotify()

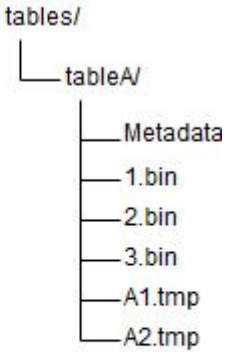
Anexo I - Compactación

El proceso de compactación será el encargado de unificar los distintos archivos temporales y binarios de una tabla para **maximizar la eficiencia de los SELECT.** Se realizará dinámicamente a nivel de tabla donde cada una de éstas indicará dentro de su metadata cada cuánto tiempo se debe verificar si hay un archivo temporal que requiere compactar.

De esta manera cuando se analice si se debe realizar una compactación de una tabla se nos presentarán los siguientes casos:



Caso A



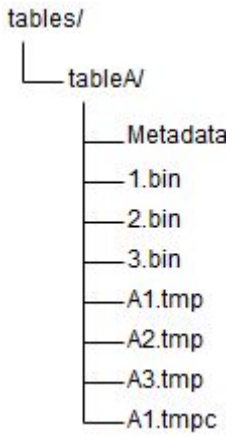
Caso B

En el Caso A se ve como cada partición de la tabla solo tiene un archivo ".bin" que representan los archivos originales e indica que no se realizó ninguna operación INSERT sobre la misma (o estos aún se encuentran en la memtable).

Por otro lado, vemos en el Caso B que existen dos archivos ".tmp" que indica que se realizaron dos dump's a disco de cambios sobre la tabla y éstos ***deben ser compactados***. A partir de este momento seguiremos el caso B de análisis como descripción para el funcionamiento de la compactación.

El proceso de compactación tiene como objetivo permitir que la tabla permanezca la mayor cantidad de tiempo disponible para realizar posibles SELECT bloqueando la misma solo en el caso que se indica en este anexo.

Una vez detectado el Caso B debemos proceder a definir el alcance de nuestra compactación, el cual quedará definido por los archivos ".tmp" actuales. Para impedir que durante nuestra compactación se ingrese un nuevo dump y que dicho archivo obtenga el nombre "A3.tmp", procederemos a renombrar todos los archivos temporales a ".tmpc" que indicará que están dentro del proceso de compactación. Nuestro nuevo directorio quedará de la siguiente manera:



Esto permitirá:

1. Trabajar en el proceso de compactación sin preocuparse de si se realiza o no un nuevo dump sobre dicha tabla.
2. Poder manejar nuevos dump sobre dicha tabla, creando un nuevo archivo “A1.tmp” que quedarán por fuera del alcance de la compactación actual.
3. Seguir realizando SELECT como si la compactación no existiese durante su ejecución. En este caso, se debe buscar una Key no solo en los archivos temporales sino también en los distintos archivos temporales en compactación. Por ejemplo: “1.bin”, “A1.tmp”, “A2.tmp”, “A3.tmp” y “A1.tmpc”.

El siguiente paso de la compactación será analizar los archivos “.tmpc” registro a registro y compararlos contra los existentes en nuestro “.bin”. Por cada registro que se lea, al no estar diferenciados por partición dentro de los archivos temporales, se deberá calcular a qué partición corresponde. Aquí se pueden presentar los siguientes casos:

- La Key X existente en el archivo “.tmpc” no existe en el archivo “.bin”: En este caso se debe agregar dicha Key, con su valor y Timestamp al final del archivo “.bin”.
- La Key X existente en el archivo “.tmpc” también existe en el archivo “.bin”: Se debe validar los distintos valores de Timestamp y quedarse con el más reciente. En caso que dicho valor sea el del archivo “.bin”, se debe proceder sin realizar operación alguna. Caso contrario, se debe actualizar el valor y el Timestamp de nuestro archivo “.bin”.

Una vez obtenida la diferencia entre los registros de los distintos archivos se deben realizar las siguientes operaciones:

1. Bloquear la tabla para cualquier operación sobre la misma.
2. Liberar los bloques que contengan el archivo “.tmpc”
3. Liberar los bloques que contengan el archivo “.bin”
4. Solicitar los bloques necesarios para el nuevo archivo “.bin”
5. Grabar los datos en el nuevo archivo “.bin”
6. Desbloquear la tabla y dejar un registro de cuánto tiempo estuvo bloqueada la tabla para realizar esta operatoria.

Cabe destacar que cualquier bloqueo de la tabla fuera del enunciado se dará como funcionalidad incorrecta y es consecuencia de desaprobación del trabajo práctico.

Anexo II - Tipos de Consistencias

En este apartado nos encargaremos de explicar teóricamente los dos tipos de consistencia que existen en el trabajo práctico. El objetivo del mismo es comprender cuales son los alcances de cada una de ellas. [14]

Strong Consistency

Trabajar con una consistencia fuerte tiene como objetivo mantener siempre la consistencia de los datos que se consultan, ingresan y actualizan. Ésto quiere decir que no importa quién ni cuándo realice esa operación, siempre el resultado va a ser el más actualizado, se encuentre éste tanto en memoria o en el FS.

Un ejemplo claro donde ésta consistencia debe darse de ésta forma es en el ámbito financiero donde si dos personas operan sobre una misma cuenta, ambos deben ver la misma información ya que es de gran importancia y crítica dentro del contexto.

Como corolario: si siempre se necesita que un dato esté actualizado y sea el mismo, éste debe ser extraído de un único punto de origen, hacerlo de varios implica que uno de los dos puede estar desactualizado en un tiempo determinado.

Si esto lo extrapolamos a nuestro trabajo práctico, al criterio al cual le lleguen request de aquellas tablas cuya consistencia es “Strong Consistency” deberá saber asociar una key con una determinada memoria (por ejemplo, usando una única instancia o utilizando la función de hash sobre la misma), estando sujetos a los tiempos y las request que esta pueda responder.

Eventual Consistency

¿Qué pasa cuando no es requerido que nuestros datos estén actualizados justo luego de haberlos insertado? ¿Cuándo en pequeños periodos de tiempo no importa si uno devuelve un dato previo o “eventualmente inconsistente”? En este caso es cuando surge el concepto de Eventual Consistency.

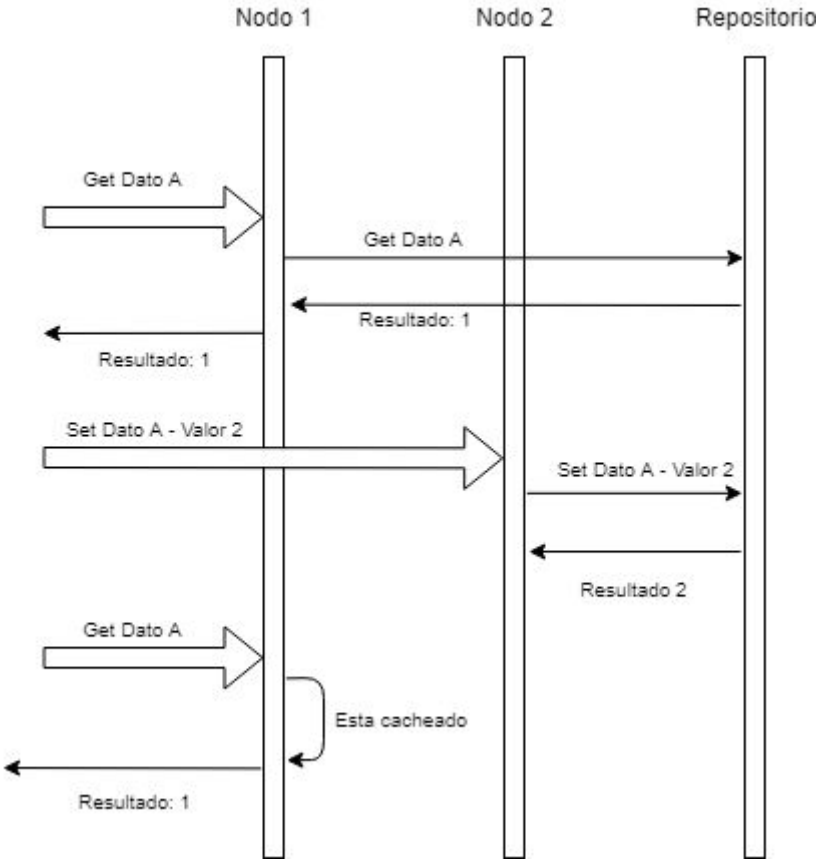
Este tipo de consistencia permite romper la limitación anterior (con respecto al único punto de origen) a costas de que no siempre un dato sea el actual.

Para ésto imaginemos un ambiente distribuido en donde disponemos distintos nodos los cuales consultan/actualizan datos contra un repositorio (filesystem, base de datos, etc). En un momento dado, un Nodo A puede consultar el dato 1, obteniendo que su contenido es la palabra “Sisop” y mantenerlo en una memoria interna. Inmediatamente después el Nodo B modifica el repositorio actualizando el dato 1 al valor “Sistemas Operativos”. Si consultamos nuevamente el dato 1 al Nodo A obtendremos el valor “Sisop” que es el que se mantenía dentro de la memoria, mientras que el mismo en el repositorio si es distinto, fue actualizado.

Este es un caso de consistencia eventual: el valor devuelto por el Nodo A ya no es consistente con el que se encuentra en el repositorio, pero en algún momento lo fue.

[14] Si se desea mas información y de otra fuente recomendamos leer https://hackernoon.com/eventual-vs-strong-consistency-in-distributed-databases-282fdad37cf7

Para ilustrar el caso explicado anteriormente adjuntamos el siguiente diagrama de secuencia.



Si esto lo extrapolamos a nuestro trabajo práctico, al criterio al cual le lleguen request de aquellas tablas cuya consistencia es “Eventual Consistency” podrá trabajar y estar asociado a múltiples memorias.

Anexo III - Gossiping

Hasta este momento, sabemos que tendremos un Kernel que conocerá por su archivo de configuración a una ***única memoria*** y tendremos múltiples memorias que conocerán también una o múltiples memorias. Dicho esto, nos preguntamos cómo el Kernel conocerá al pool de memorias para poder agregarlo a los distintos criterios. Para esto, utilizaremos el concepto de ***Gossiping***.

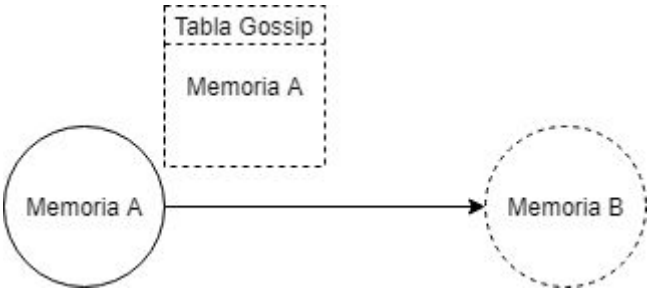
El Gossiping es una técnica de comunicación interna para que los nodos, en este caso las memorias, de un cluster/pool se comuniquen entre sí. Es un protocolo de transmisión inter-nodal eficiente, ligero y confiable para difundir y transmitir datos. Es descentralizado, "epidémico", tolerante a fallas y un protocolo de comunicación de igual a igual. Se suele utilizar para el descubrimiento de pares y la propagación de metadatos.

El proceso se ejecuta temporalmente en cada nodo e intercambia mensajes de estado con otro de su par en el clúster. Dado que todo el proceso está descentralizado, no hay nada ni nadie que coordine cada nodo para comenzar su gossip. Cada nodo de forma independiente siempre seleccionará un conjunto de nodos para realizar dicho proceso.

Como se puede observar, el proceso al ejecutarse temporalmente y de forma descentralizada, toda la información del sistema se mantendrá “eventualmente” actualizada. El sistema irá conviviendo con los cambios que se ocasionen en el mismo.

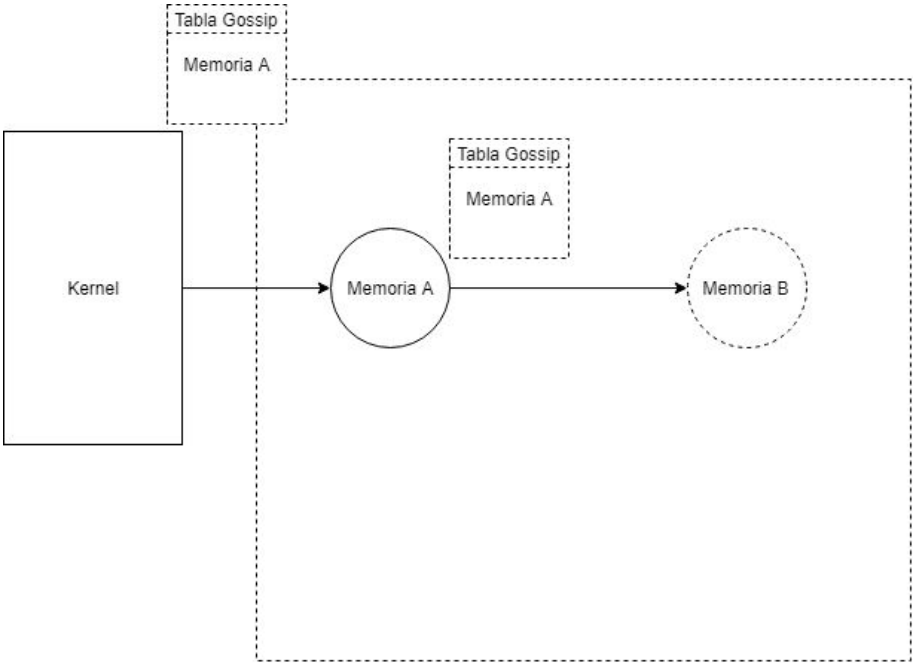
En nuestro trabajo práctico utilizaremos una aproximación de esta implementación para la difusión y propagación del estado de los nodos.

En una primera instancia pensaremos en un sistema base nuestro donde cada memoria tendrá solamente a una memoria como seed (este es el caso óptimo que buscaremos llegar en la entrega final). Para esto, Iniciaremos la ***Memoria A*** en el Tiempo 0 (***T0***) donde tendrá como memoria seed a la ***Memoria B*** que aún no fue ingresada en el sistema (ver imagen). Para mantener una lógica en la explicación de nuestro ejemplo diremos que *el proceso de gossip se ejecutará cada 4 unidades de tiempo* en cada uno de los nodos.



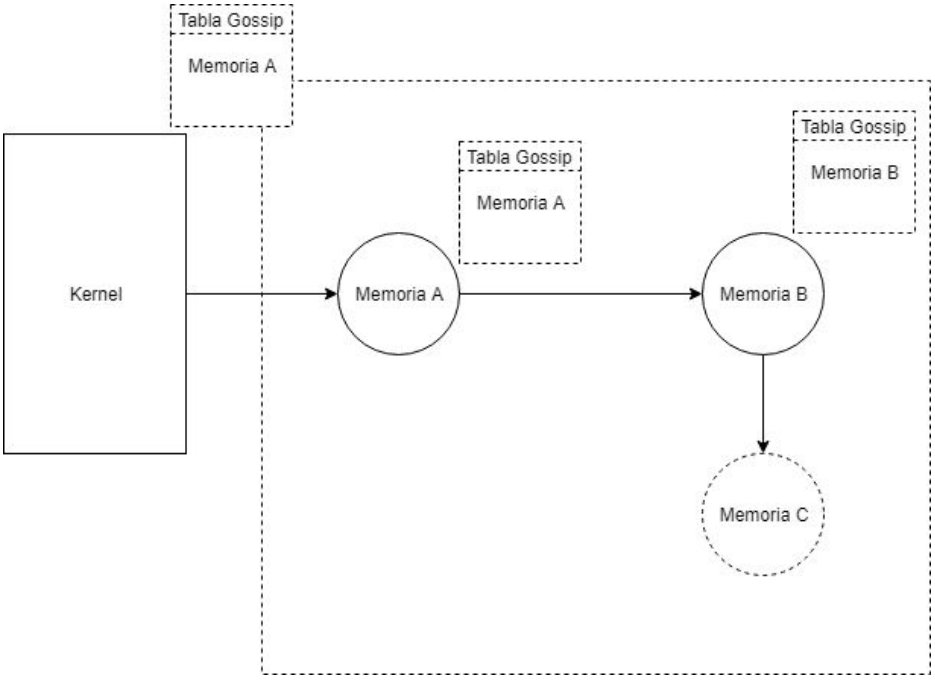
Al momento de ingresar en el sistema la ***Memoria A*** comenzará su mecanismo de gossip para descubrir y conocer a los demás pares en su cluster/pool. Este proceso lo que producirá es consultarle a su memoria seed (***Memoria B***) cuál es el estado de su Tabla Gossip. Como la ***Memoria B*** no se encuentra conectada, dicho proceso de Gossip fallará y la Memoria A indicará en su tabla de gossip que solo existe ella misma en el cluster.

A continuación (***T1***) ingresará el ***Kernel*** que conocerá a la ***Memoria A*** y realizará el proceso de descubrimiento del pool, consultandole a la ***Memoria A*** quienes son los pertenecientes en pool.



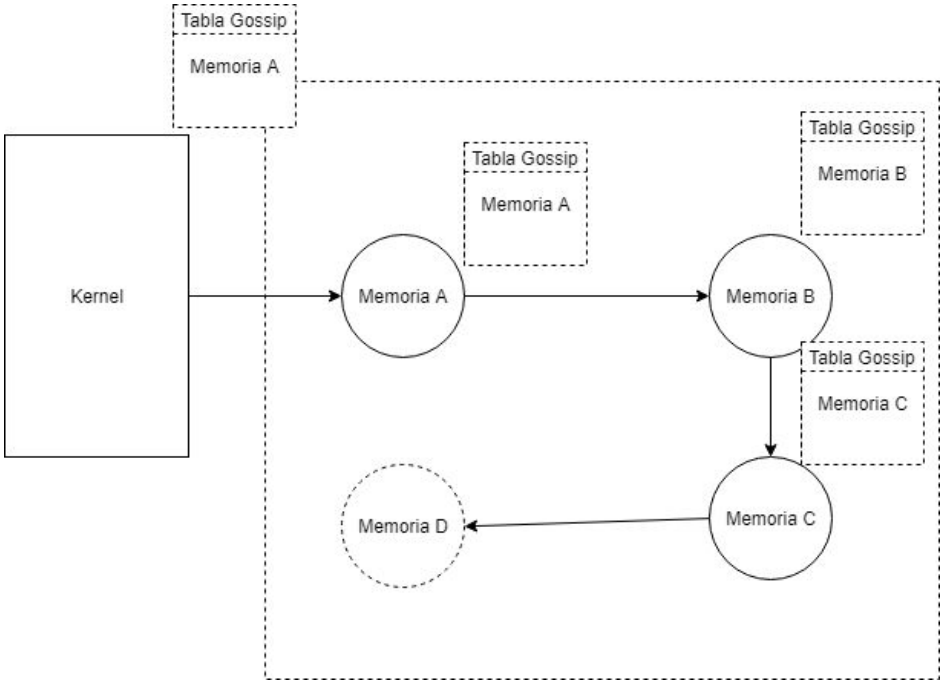
Cuando realice este descubrimiento le preguntará a la memoria que él conoce quiénes se encuentran dentro del clúster actual. La ***Memoria A*** le transmitirá su Tabla de Gossiping y de esta manera el ***Kernel*** conocerá a dicha memoria y podrá asignarla a sus criterios. Hasta este punto, vemos que el sistema se mantiene estable y todos los procesos se “conocen” entre sí.

Para empezar a comprender cómo funciona este proceso ahora incluiremos en ***T2*** a la ***Memoria B*** al sistema, donde conocerá a la ***Memoria C*** como seed que aún no se ingresó en el sistema.



Como realizo la ***Memoria A*** en su momento, la ***Memoria B*** comenzará su proceso de Gossip y se intentará comunicar con la ***Memoria C***. Este proceso, al no encontrarse aún en el sistema la ***Memoria C,*** fallará y en su tabla de gossip quedará que solo existe el. Observen como la ***Memoria B*** solo se conecta con un nodo y si dicho nodo no responde dentro de su tabla de ***gossip pondrá que sólo él existe***. Así tendremos un sistema de dos memorias donde ninguna de las dos conocerá la existencia de la otra. En este momento empezamos a ver que el ***sistema inconsistente***.

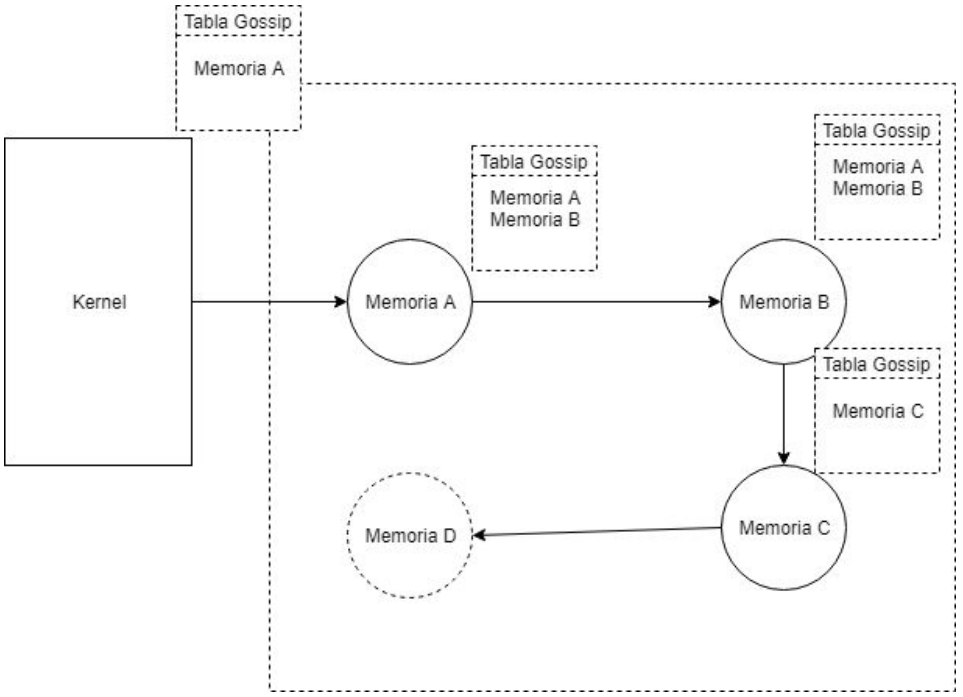
Para continuar con el ejemplo, en ***T3*** agregaremos una nueva ***Memoria C*** que conoce como memoria seed a una ***Memoria D*** que aún no existe en el sistema.



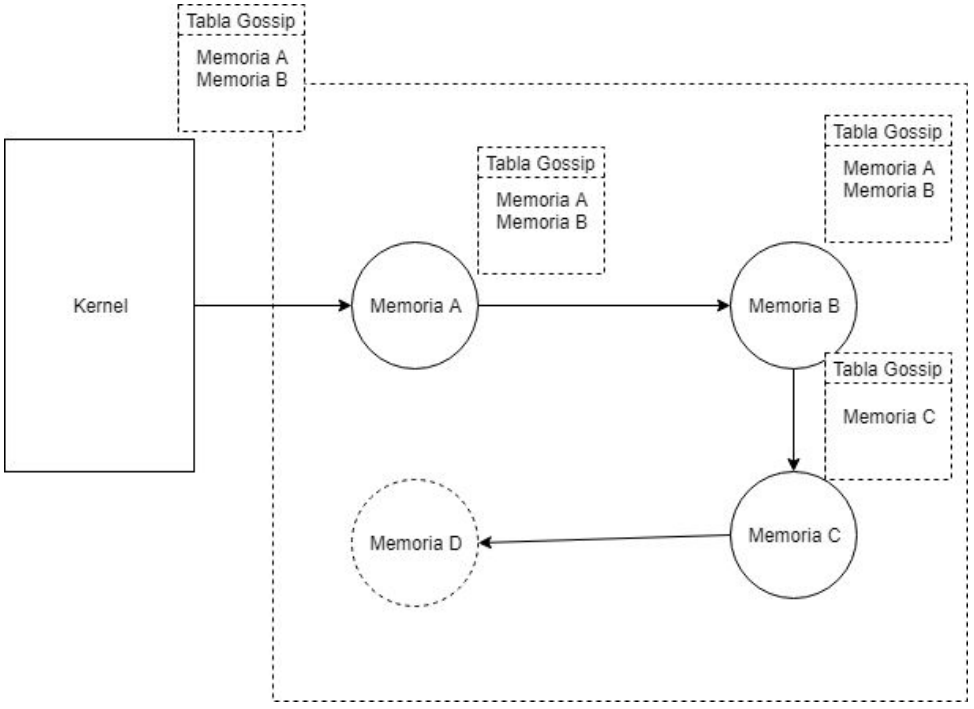
La ***Memoria C*** al igual que en el caso anterior comenzará su proceso de Gossip y fallará al no encontrar la ***Memoria D*** conectada y definirá su tabla de gossip como ella sola.

En el siguiente punto, Tiempo 4 (***T4***). ***Memoria A*** volverá a comenzar su proceso de gossip ya que definimos inicialmente que el gossip se realizará siempre cada 4 unidades de tiempo. En este momento la ***Memoria A*** volverá a intentar conectarse a la ***Memoria B*** y en este punto intercambiarán su tabla de gossip agregando los nodos faltantes.

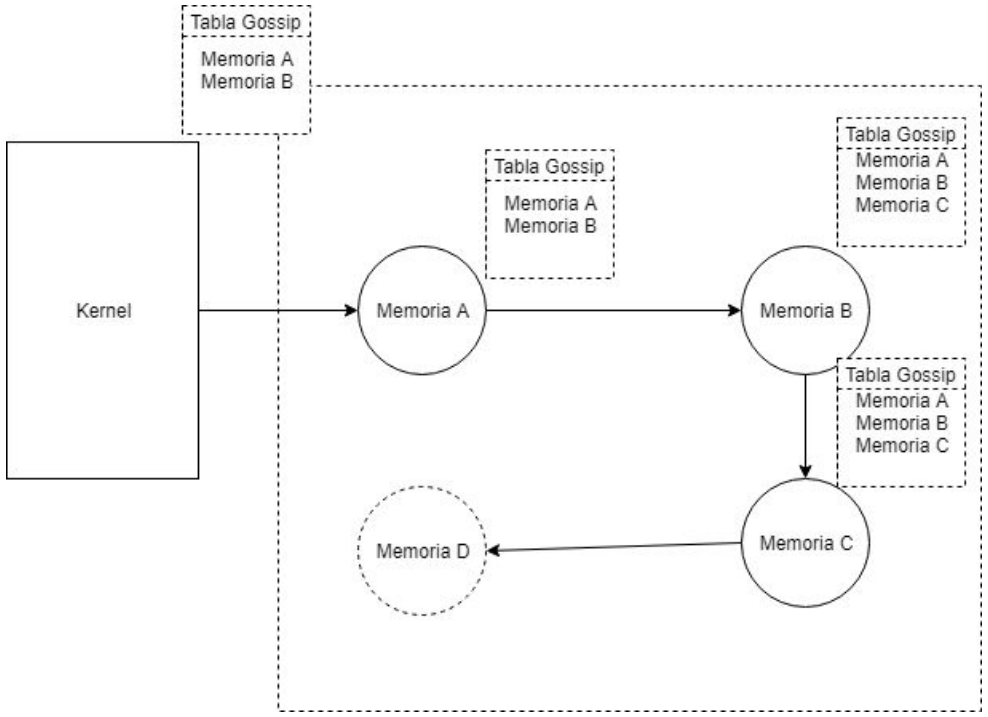
Vemos que la ***Memoria A*** conocerá a la ***Memoria B*** y viceversa (debido al intercambio de las tablas de cada una). Observen como la ***Memoria C*** no participa en esta comunicación ya que la comunicación solo se realiza en un sentido y la realiza A hacia B.



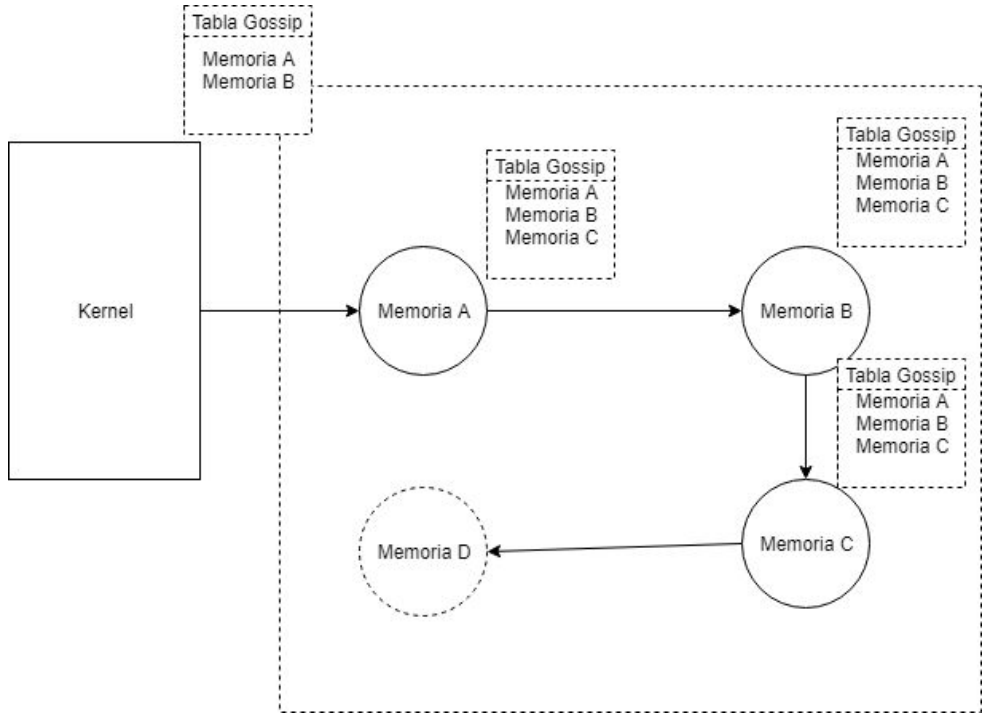
En ***T5***, El Kernel volverá a consultar a ***Memoria A*** cuáles son los nodos pertenecientes al sistema y al realizarlo descubrirá que también existe la ***Memoria B***.



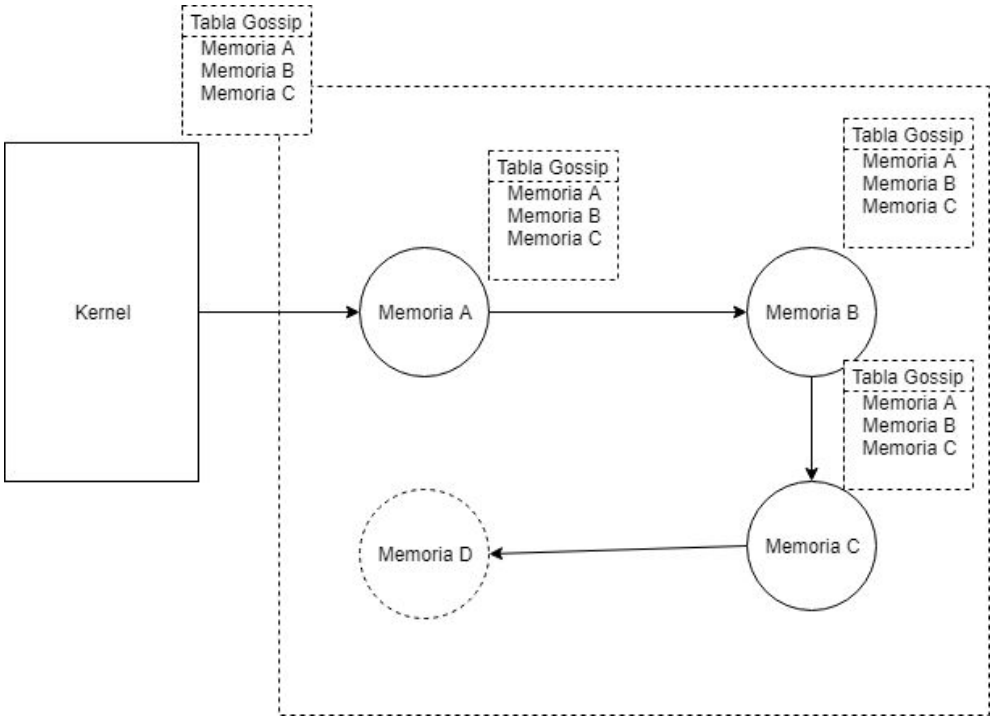
Cuando se ejecute ***T6***, la ***Memoria B*** volverá a realizar su proceso de gossip comunicándose con la ***Memoria C*** e intercambiarán su conocimiento de los nodos actuales. En este punto, vemos tanto la ***Memoria B*** como ***C***, conocen a la totalidad del pool mientras que la ***Memoria A*** aún no (hasta que vuelva a realizar su proceso de gossip).



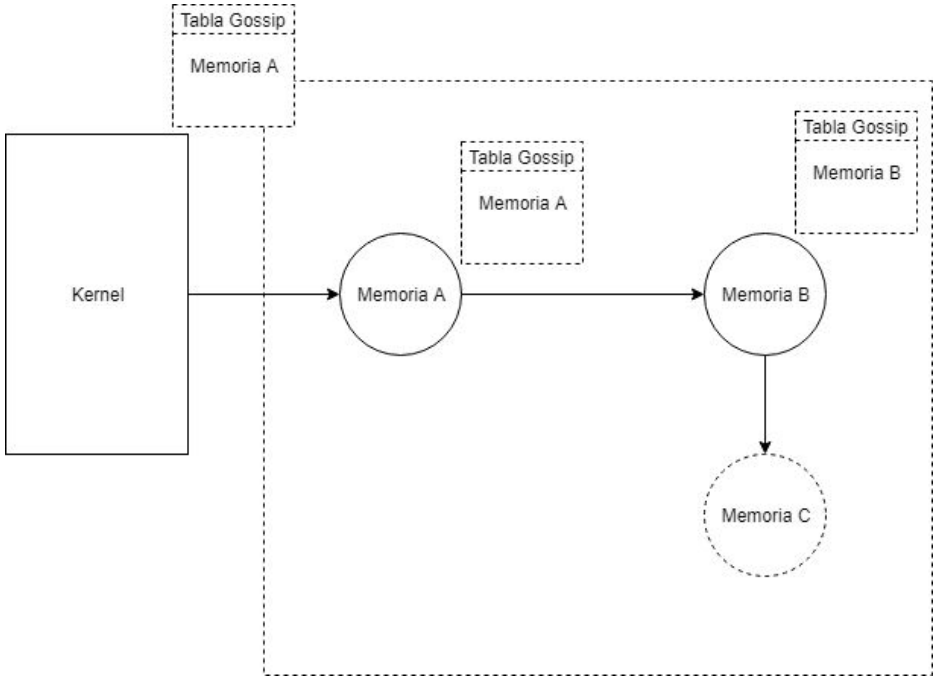
En ***T7*** la ***Memoria C*** comenzará su proceso de gossip y volverá a fallar por lo que no se altera el sistema. En cambio, cuando se llegue ***T8*** la ***Memoria A*** volverá a comenzar su proceso de gossip y en este caso se comunicará con la ***Memoria B*** y al intercambiar la metadata descubrirá a la ***Memoria C***.



Finalmente, cuando lleguemos a ***T9***, el ***Kernel*** volverá a consultar a ***Memoria A*** cuál es el estado del pool y el ***Kernel*** conocerá al pool en su totalidad.

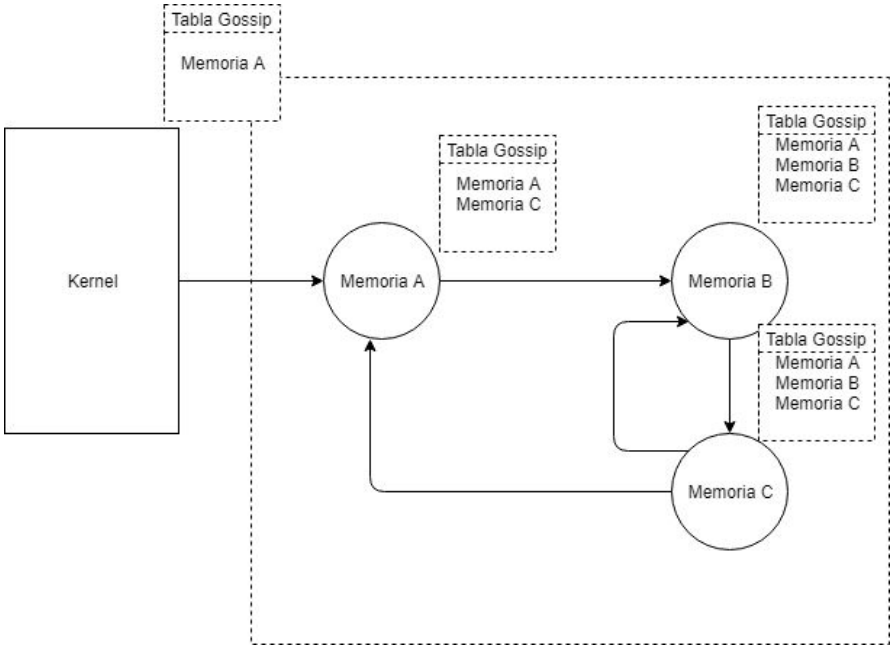


Dejaremos este caso en este punto e iremos a otro caso en el cual una de las memorias tenga varias seeds para ver cómo impacta esto en nuestro sistema. Inicialmente conectaremos al igual que el ejemplo anterior conectaremos primero una ***Memoria A***, el ***Kernel*** y la ***Memoria B*** (partiremos desde el ***T3***).



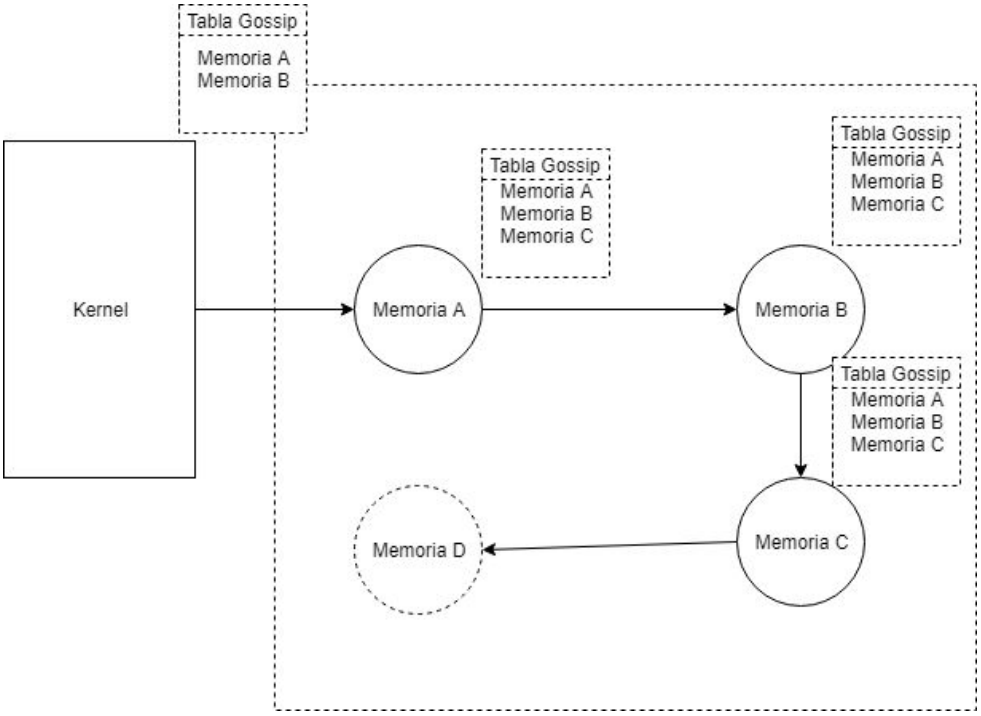
Aquí modificaremos y conectaremos una ***Memoria C*** que tenga como seeds tanto a la ***Memoria A*** como a la ***Memoria B*** (configuradas en ese orden). Inicialmente al conectarse la ***Memoria C*** se intentará comunicar con la ***Memoria A*** e intercambiar su tablas actuales. En este punto la ***Memoria C*** conocerá a la ***Memoria A*** y viceversa.

A continuación, ***Memoria C*** seguirá intercambiando información con su siguiente seed y se comunicará con ***Memoria B***. En este punto, intercambiarán su tabla de gossip y como ***Memoria C*** ya conoce a ***Memoria A***, la ***Memoria B*** pasará también a conocerla.

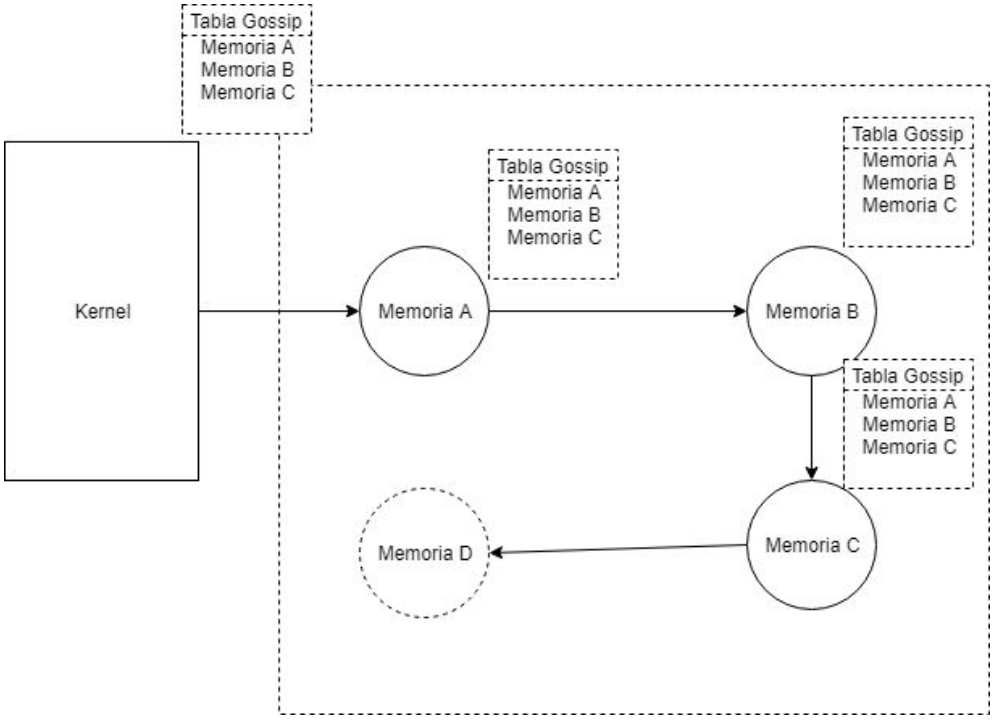


Observen como en este punto, la ***Memoria B*** y la ***Memoria C*** tienen conocimiento de todo el Pool pero la ***Memoria A*** aún no y esto se debe al orden en el cual se ordenan los seeds de la ***Memoria C***.

En ***T4***, la ***Memoria A*** volverá a realizar su gossip y al intercambiar con la ***Memoria B*** la totalidad del pool de memorias se conocerá entre ellas.



Por último, en ***T5***, el kernel se comunicará con la ***Memoria A*** y obtendrá toda la metadata del pool de memorias.



Una vez explicado estos casos haremos un par de conclusiones:

1. El sistema es eventualmente consistente ya que recién pasadas 9 unidades de tiempo el sistema logró quedar estable y que todos los nodos se conozcan.
2. El proceso es completamente descentralizado, cada memoria ejecuta su pedido cuando deba hacerlo cada X unidades de tiempo y solo conocerá la sumatoria de sus nodos más los de su memoria seed.
3. El proceso es tolerante a fallas, si un nodo intenta conectarse a otro y este falla, solo esperará a volverse a ejecutar para ver si esto cambió.
4. El Kernel eventualmente podrá ir agregando las memorias a los distintos criterios en tanto se vaya realizando el proceso de gossip.
5. Que una memoria posea varios seeds hace que el proceso de gossip pueda ser consistente más rápidamente.

Anexo IV - Journaling

Recordamos que dentro de nuestro trabajo práctico planteamos la opción de realizar las request sobre memoria, debido a que nos ahorra consultar y/o modificar directamente contra la parte principal del FileSystem, permitiendo mejorar la performance del sistema.

El problema surge cuando podríamos llegar a realizar un INSERT sobre alguna tabla en memoria, escritura de la cual el FS no tendrá conocimiento.

El proceso de Journaling surge, entonces, para poder notificar al File System todas aquellas escrituras que fueron realizadas en memoria. Finalmente: es un mecanismo que realiza un seguimiento de los distintos cambios dentro de cada memoria que aún no fueron impactados en el FileSystem, para luego poder impactarlos sobre la parte principal.

Dentro de nuestro trabajo práctico identificamos todas aquellas escrituras a ser notificadas al FS con el “Flag de Modificado” con el que cuenta cada una de las páginas. Es decir: todas aquellas páginas con el flag activado son las que contienen las Key que deben ser actualizadas en el FS. Las páginas cuyo flag esté desactivado implican que el dato en memoria es consistente (o eventualmente consistente) con el que está en el FS.

La desventaja de utilizar esta técnica es que, en caso de una caída global del sistema, los Flags de Modificado se perderán, pudiendo estos cambios nunca ser impactados finalmente en el FS. Este es un riesgo que estamos dispuestos a tomar.

Los cambios serán informados al File System a partir de la ocurrencia de alguno de los siguientes disparadores:

1. El Journal automático realizado cada X unidades de tiempo (configurado por archivo de configuración).
2. El Journal manual por medio de la API que dispone la Memoria (JOURNAL).
3. El Journal forzoso debido a que la Memoria entró en un estado FULL y requiere nuevas páginas para asignar (realizado por pedido del Kernel)
4. El Journal manual por medio de la API que dispone el Kernel.

Cuando ocurra alguno de estos disparadores diremos que comienza, en nuestro trabajo práctico, el proceso de Journaling de una memoria.

Este proceso consiste en encontrar todas aquellas páginas cuyas Key deben ser actualizadas dentro del FS y enviar por cada una de estas una petición de insert al FileSystem indicando los datos adecuados.

Una vez efectuados estos envíos se procederá a eliminar los segmentos actuales.

Anexo V - Ejemplos

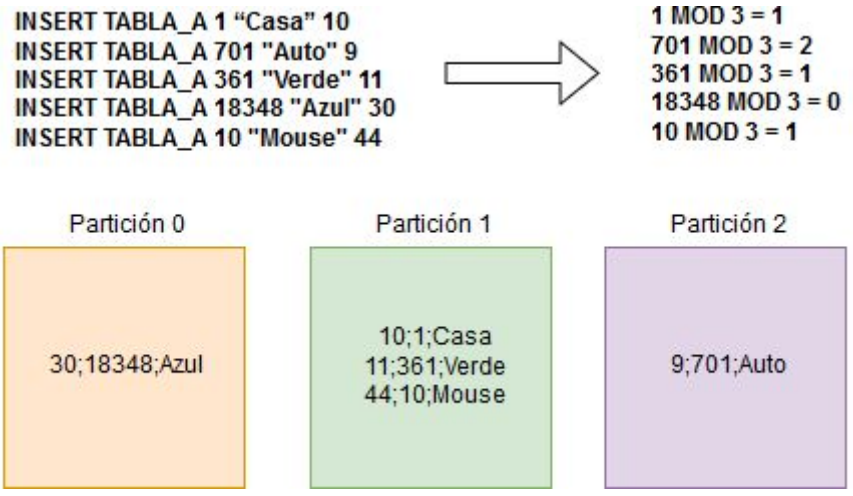
Para los próximos ejemplos:

Nomenclatura INSERT utilizada: INSERT [NOMBRE_TABLA] [KEY] “[VALUE]” [Timestamp]

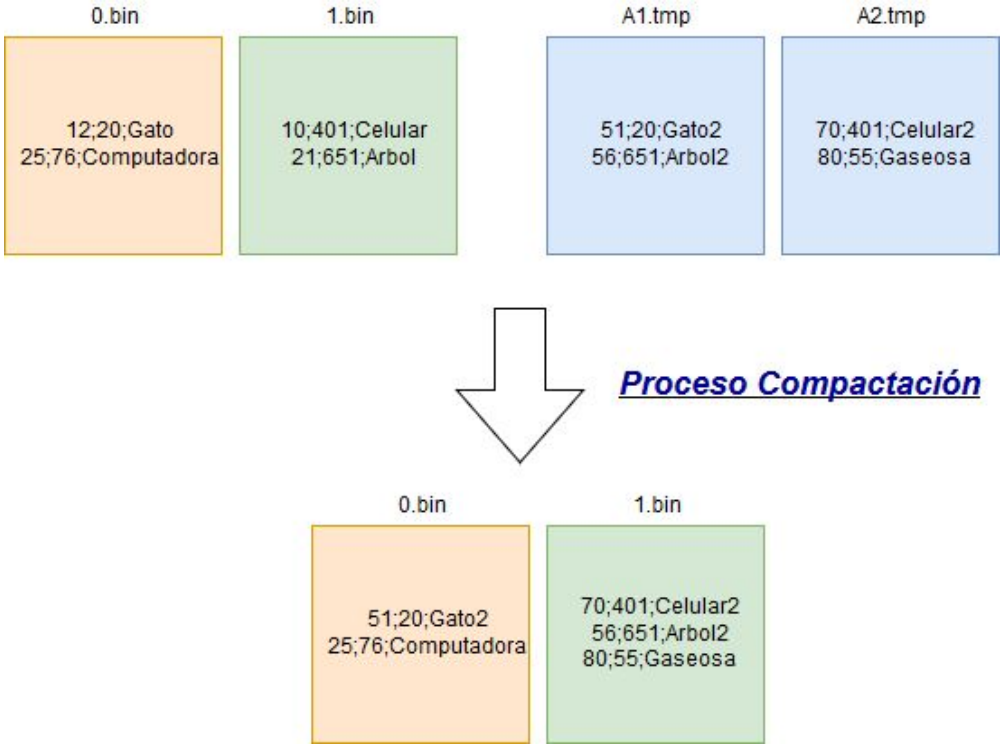
Tupla utilizada: Timestamp;Key;Value

Ejemplo de particionamiento

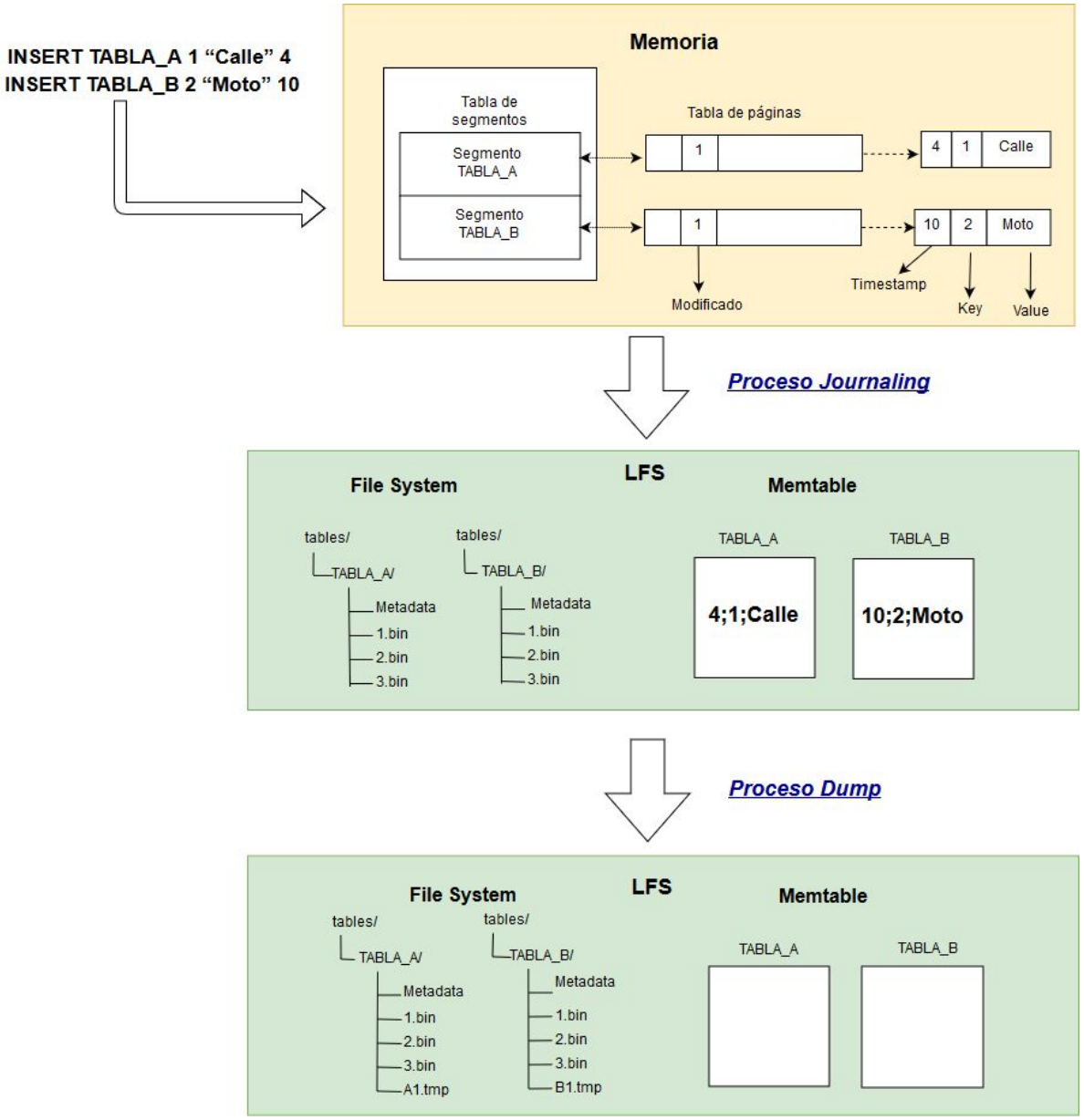
Considerando que la TABLA_A tiene 3 particiones.



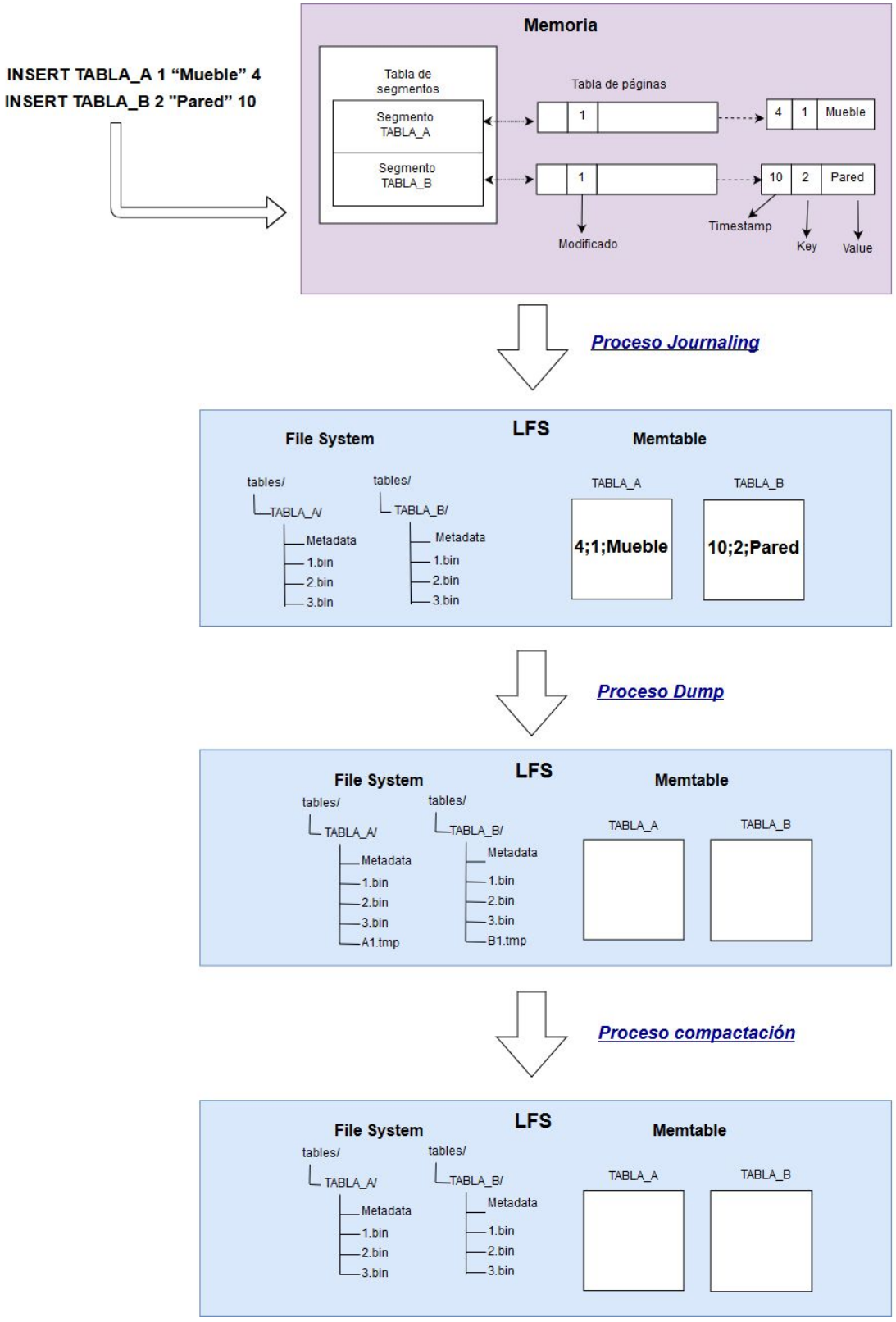
Ejemplo de compactación



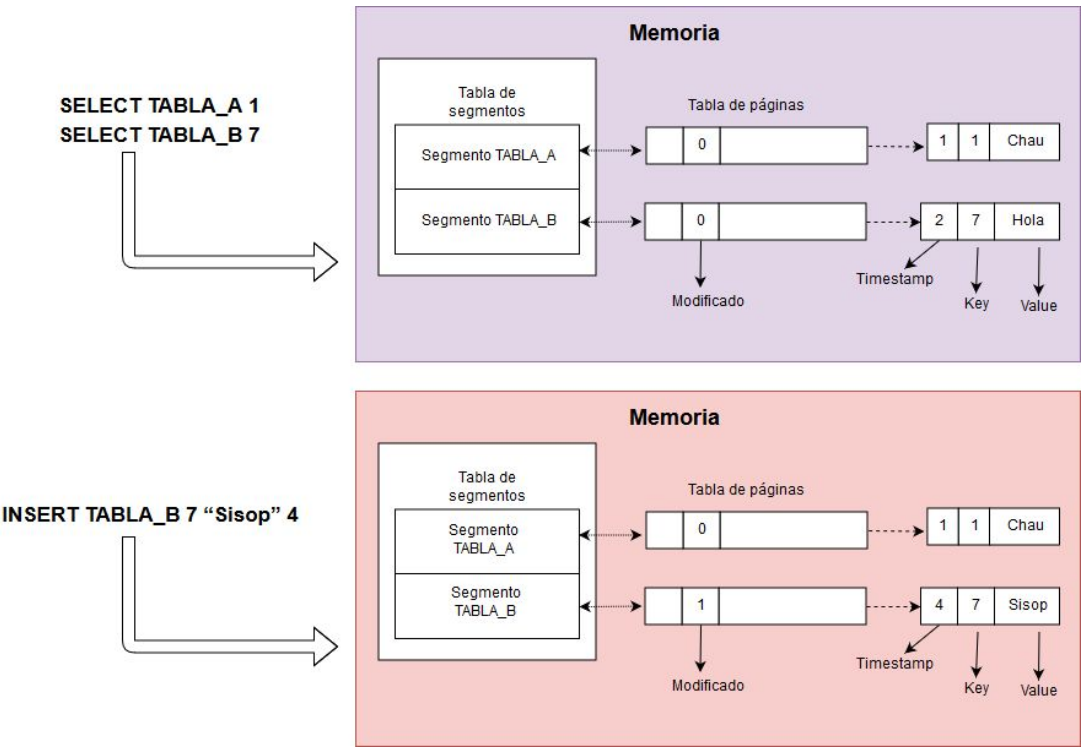
Ejemplo de dump



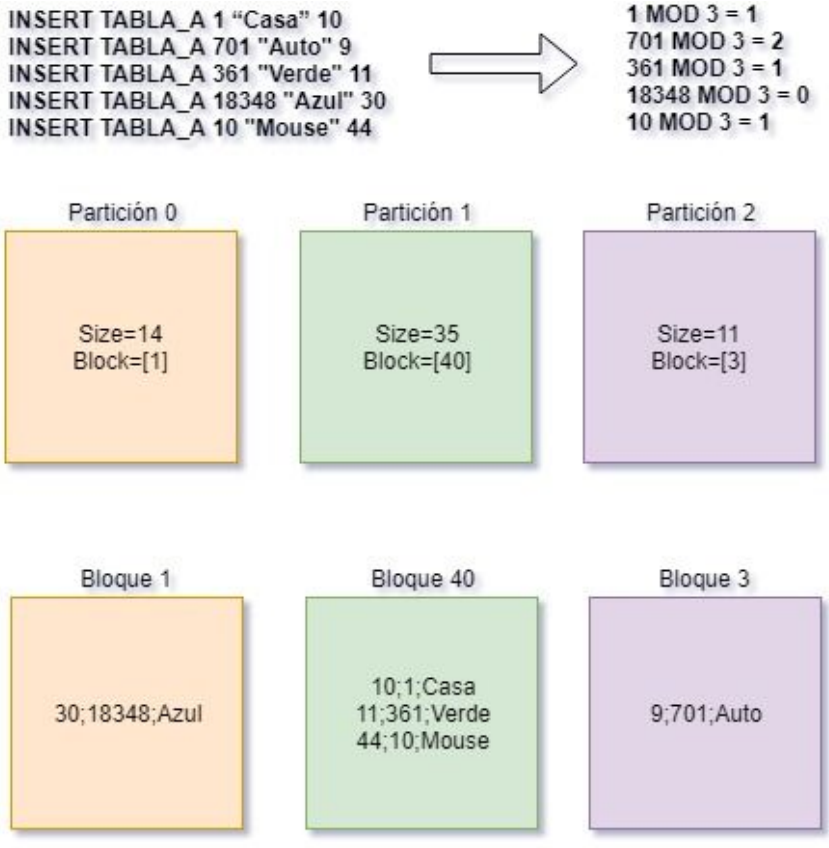
Flujo completo de los datos

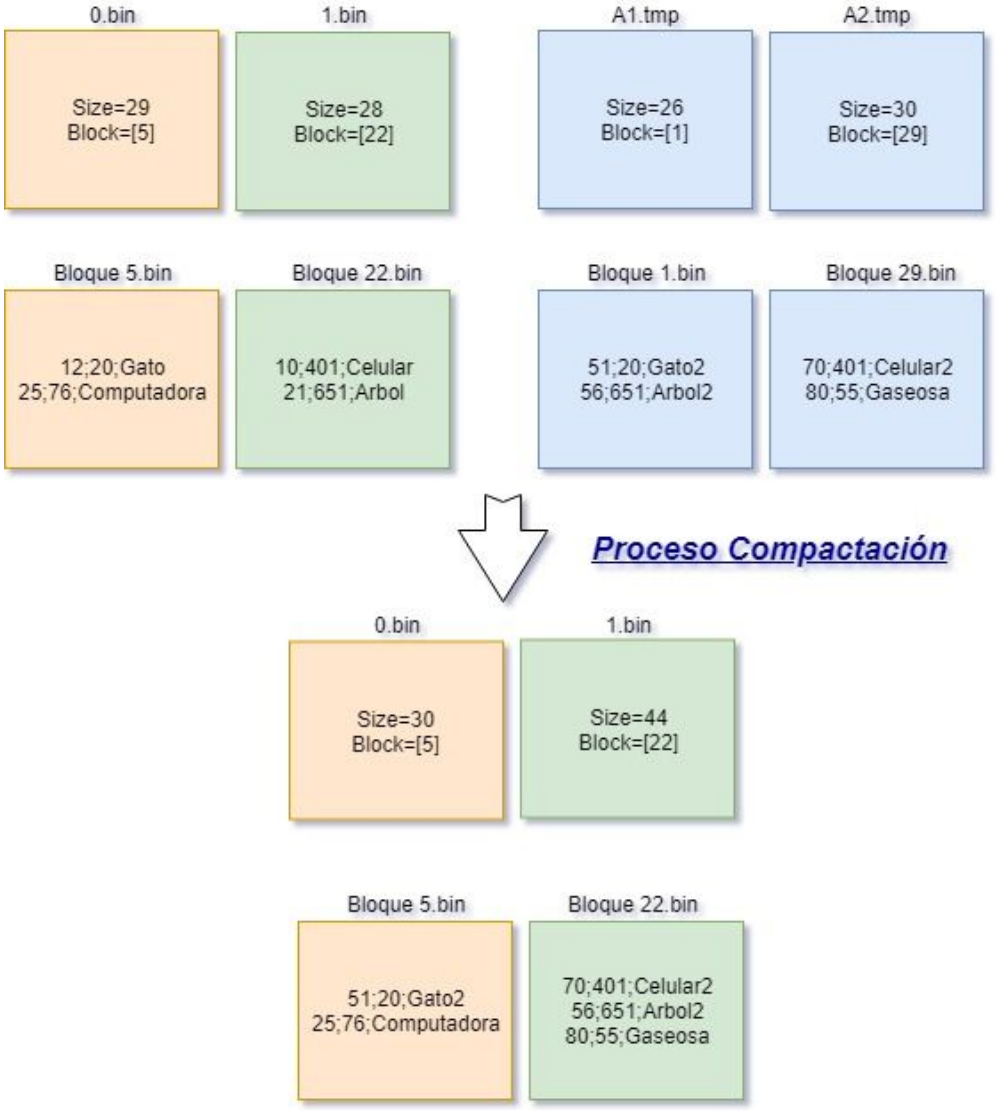


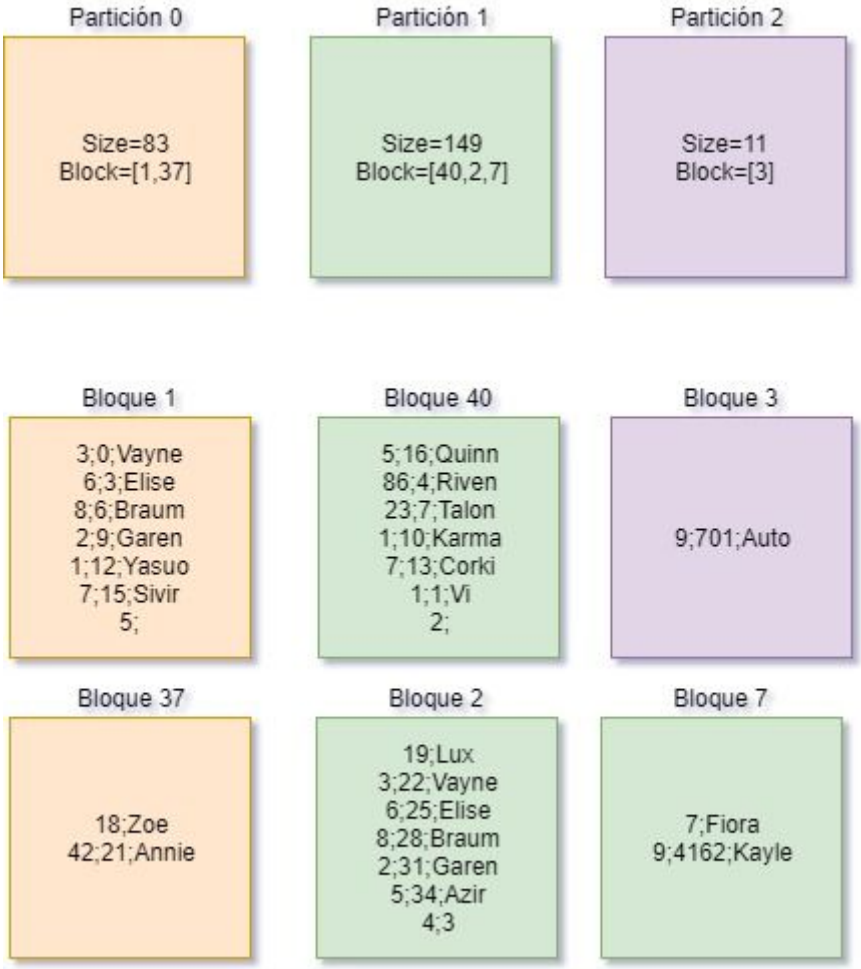
Ejemplo de Flag de Modificado



Ejemplos bloques en File System




0.bin
Size=29
Block=[5]
1.bin
Size=28
Block=[22]
A1.tmp
Size=26
Block=[1]
A2.tmp
Size=30
Block=[29]
Bloque 5.bin
12;20;Gato
25;76;Computadora
Bloque 22.bin
10;401;Celular
21;651;Arbol
Bloque 1.bin
51;20;Gato2
56;651;Arbol2
Bloque 29.bin
70;401;Celular2
80;55;Gaseosa
Proceso Compactación
0.bin
Size=30
Block=[5]
1.bin
Size=44
Block=[22]
Bloque 5.bin
51;20;Gato2
25;76;Computadora
Bloque 22.bin
70;401;Celular2
56;651;Arbol2
80;55;Gaseosa



Descripción de las entregas

Hito 1: Conexión Inicial

Fecha: 20 de Abril

Objetivos:

- ★ Familiarizarse con Linux y su consola, el entorno de desarrollo y el repositorio
- ★ Aplicar las Commons Libraries, principalmente las funciones para listas, archivos de conf y logs
- ★ Definir el Protocolo de Comunicación

Implementación mínima:

- ★ Creación de todos los procesos que intervienen
- ★ Desarrollar una comunicación simple entre los procesos que permita propagar un mensaje por cada conexión necesaria
- ★ Implementar las APIs de los distintos componentes sin funcionalidades

Lectura recomendada:

- http://faq.utnso.com/arrancar
- Beej Guide to Network Programming - link
- Linux POSIX Threads - link
- SO UTN FRBA Commons Libraries - link
- Sistemas Operativos, Silberschatz, Galvin - Capítulo 3: Procesos
- Sistemas Operativos, Silberschatz, Galvin - Capítulo 4: Hilos

Hito 2: Avance del Grupo

Fecha: 18 de Mayo

Objetivos:

- ★ Familiarizarse con el desarrollo de procesos servidor
- ★ Comprender el manejo de memoria dinámica y administrarla correctamente
- ★ Comprender el manejo de archivos e interpretar su contenido

Implementación mínima:

- ★ El LFS debe ser capaz de responder las request SELECT e INSERT utilizando un archivo del directorio propio de Linux
- ★ La memoria debe ser capaz de responder las request SELECT e INSERT
- ★ La memoria sabe manejar un único segmento con una única página
- ★ El kernel debe ser capaz de recibir archivos LQL y parsearlos
- ★ El criterio SC debe poder saber elegir una memoria entre las que tiene asignadas[15]

Lectura recomendada:

- Sistemas Operativos, Silberschatz, Galvin - Capítulo 4: Hilos
- Sistemas Operativos, Silberschatz, Galvin - Capítulo 5: Planificación
- Sistemas Operativos, Silberschatz, Galvin - Capítulo 8: Memoria

Hito 3: Checkpoint Presencial en el Laboratorio

Fecha: 08 de Junio

Objetivos:

- ★ Comprender el funcionamiento interno de distribución de memoria y creación de segmentos.
- ★ Administración de archivos (crearlos y editarlos)

Implementación mínima:

- ★ El LFS debe ser capaz de responder las request CREATE, SELECT e INSERT utilizando el submódulo propio de FS del trabajo práctico

[15] Para esta iteración simplemente se le asignará la memoria configurada por archivo de configuración y sin necesidad de utilizar el comando ADD

- ★ El kernel debe ser capaz de agregarle memorias a los criterios mediante el comando ADD
- ★ El kernel utiliza un esquema de múltiples estados para la ejecución de un archivo LQL con un único script en estado “Exec”
- ★ La memoria debe manejar distintos segmentos con distintas páginas cada uno
- ★ La memoria debe poder realizar CREATE

Lectura recomendada:

- Sistemas Operativos, Silberschatz, Galvin - Capítulo 8: Memoria
- Sistemas Operativos, Silberschatz, Galvin - Capítulo 10 y 11: File System

Hito 4: Avance del Grupo

Fechas: 29 de Junio

Objetivos:

- ★ Comprender el funcionamiento de los algoritmos de reemplazo
- ★ Familiarizarse con el concepto de Journaling
- ★ Comprender el funcionamiento de una compactación

Implementación mínima:

- ★ El LFS debe ser capaz de realizar el proceso de compactación
- ★ Criterio EC terminado
- ★ El kernel utiliza un esquema de múltiples estados para la ejecución de un archivo LQL con múltiples scripts en estado “Exec”
- ★ La memoria debe implementar el algoritmo de reemplazo
- ★ La memoria debe saber responder si esta FULL
- ★ La memoria debe realizar journaling

Lectura recomendada:

- Sistemas Operativos, Silberschatz, Galvin - Capítulo 8: Memoria
- Sistemas Operativos, Silberschatz, Galvin - Capítulo 10 y 11: File System

Hito 5: Entregas Finales

Fechas: 13 de Julio - 20 de Julio - 03 de Agosto

Objetivos:

- ★ Probar el TP en un entorno distribuido
- ★ Realizar pruebas intensivas
- ★ Finalizar el desarrollo de todos los procesos
- ★ Todos los componentes del TP ejecutan los requerimientos de forma integral, bajo escenarios de stress.

Implementación mínima:

- ★ Todos los componentes deben saber responder todas las request
- ★ Criterio SHC terminado
- ★ Cada criterio sabe responder las métricas sobre la/s memoria/s que maneja
- ★ Las memorias deben ser capaces de conocerse entre sí a partir del concepto de gossiping

Lectura recomendada:

- Guías de Debugging del Blog utnso.com - link
- MarioBash: Tutorial para aprender a usar la consola - link