Universidad de Valladolid

E. T. S. DE INGENIERÍA INFORMÁTICA

I.T. EN INFORMÁTICA DE SISTEMAS

Implementación en C-PARMACS del Benchmark TPC-C y del SGBD Concurrente

---

Alumno: Alfredo Javier Gonel Crespo

Tutor: Benjamín Sahelices Fernández

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Dentro del Departamento de Informática, en el grupo de investigación de Arquitectura y Tecnología de Computadores (conocido como ATC), se realizan simulaciones de aplicaciones paralelas (como las que se se pueden encontrar en Splash-2 [16]) con el simulador RSIM [9], para probar sus trabajos de protocolos de coherencia caché [12, 13, 6]. Esta investigación a su vez está integrada dentro del proyecto *Computación de Altas Prestaciones IV. Jerarquía de Memoria de Altas Prestaciones* de la Universidad de Zaragoza (Ref. TIN2004-00739-C02-02). En base a esto y para mejorar la fiabilidad de los resultados obtenidos se hace necesario ampliar y completar el repertorio de aplicaciones y núcleos disponibles en las simulaciones.

El propósito de este proyecto de fin de carrera es múltiple:

- Primero, colaborar con la aplicación que se ha implementado a lo largo del desarrollo del proyecto, y así aumentar la gama de aplicaciones disponibles para las simulaciones del grupo ATC.

- A la vez que se hace disponible al resto de la comunidad informática un programa capaz de cuantificar el rendimiento de un sistema paralelo, basado en un estándar y que funciona independientemente de otros sistemas software.

- Por último, completar la formación académica de la carrera de Ingeniería Técnica de Informática de Sistemas dando una aplicación a un gran número de conceptos adquiridos durante la misma.

La aplicación que aquí se detalla es un software de medida de rendimiento, comúnmente conocido por su nombre en inglés *benchmark*. Este benchmark pretende ser una aplicación ejecutable en entornos UNIX y similares, en los que es muy común encontrarse máquinas con múltiples procesadores. La poder ejecutar la aplicación en estos entornos UNIX, se ha aplicado la compatibilidad a través del código fuente, utilizando:

- El lenguaje de programación C, muy extendido en entornos UNIX, permite obtener la aplicación ejecutable en dichos entornos UNIX sin cambiar el código gracias a las especificaciones ANSI-C y POSIX [21].

- La biblioteca de macros PARMACS [8], para una programación paralela transportable entre sistemas paralelos con diferente arquitectura de memoria.

TPC-C

Este benchmark está basado en las especificaciones del TPC (*Transaction Processing Performance Council*) [23] conocidas como TPC-C; además está destinado a medir el rendimiento del procesamiento de transacciones en línea simulando diferentes tipos de transacciones y una compleja base de datos en un entorno empresarial común y genérico pero que se aproxima bien al uso actual de los sistemas de bases de datos existentes hoy en día.

En la industria de sistemas servidores dicho benchmark es un punto de referencia; si bien TPC-C son sólo unas especificaciones, que aplicadas correctamente dan lugar a un sistema de medida del rendimiento con unidades comparables entre distintos sistemas. En el caso del TPC-C, la unidad de medida es la transacción, y la capacidad de los sistemas se compara en transacciones por minuto (tpmC). En cuanto a los sistemas paralelos, dada la clasificación de las transacciones en 5 tipos, en base a unas reglas de unicidad (ACID), y gracias a la concurrencia de peticiones , el benchmark TPC-C se convierte en un buen candidato para medir el rendimiento de estos sistemas paralelos.

Las especificaciones TPC-C dan lugar a un benchmark, que normalmente es implementado por empresas propietarias ya sea para medir el rendimiento de los sistemas que ponen a la venta o para venderlo a las empresas que lo necesiten. Esto implica que hay muy pocas implementaciones de uso libre y/o con código abierto para poder investigar, por lo que es interesante ampliar el repertorio tanto con un diseño como con una implementación libre.

Notas del diseño

Uno de los objetivos es proporcionar al grupo ATC un sistema más para sus simulaciones usando RSIM. Uno de los principales problemas de RSIM es el reducido número de bibliotecas que soporta, luego simular programas complejos que utilizan un gran número de funciones externas resulta algo inviable, y se hace necesario que los programas no dependan más que de la biblioteca estándar de funciones.

Aunque el diseño sólo se ha orientado a PARMACS, si se quiere tener la posibilidad de trasladar la aplicación de manera sencilla a RSIM, hay que tener en cuenta que aplicarlo en el diseño implica que:

- Hay que analizar más allá de las especificaciones, buscando todos los subsistemas necesarios para la implementación, ya que no podemos depender de bibliotecas externas.
- Hay que buscar una arquitectura que abarque todos los subsistemas: analizar y diseñar el sistema desde 0.
- Dado que hemos hablado de una *compleja base de datos* habrá que diseñarla directamente y no sobre un sistema gestor de bases de datos ya existente.

Aspectos de la implementación

Las implicaciones que supone el diseño de una aplicación desde cero, además del diseño de un sistema de almacenamiento, se reflejan fielmente en la implementación. En cuanto al sistema de almacenamiento, supone la tarea de codificar un gestor de bases de datos, que aun partiendo de un buen diseño, es una traba a superar.

Al estar toda la aplicación implementada en C y usando bibliotecas estándar, este proyecto se convierte en un buen candidato tanto para ser adaptado a RSIM y poderlo utilizar en las simulaciones; como para la medición de rendimiento, de manera independiente, de sistemas mono y multiprocesador.

No hay que olvidar que el lenguaje C, sin ser un lenguaje de tan bajo nivel como el ensamblador, no implementa ni facilita ciertos paradigmas de la programación que en este caso podrían ser de mucha ayuda: orientación a objetos, manejo automático de memoria, etc. Por lo que es necesario una implementación cuidadosa e ingeniosa en algunos momentos para resolver estos problemas. Aunque de la misma manera que hace falta diseñar soluciones ingeniosas, existe el problema de encontrar fallos en momentos finales del desarrollo.

Requerimientos

Por último, comentar algunos de los conocimientos más importantes que han hecho falta para solucionar todos los problemas encontrados.

- Ingeniería del software: necesaria para analizar los requisitos del estándar TPC-C y poder diseñar una arquitectura sostenible para el sistema. Se ha hecho un esfuerzo especial en dejar claros los requisitos iniciales y en formar el diseño arquitectónico que define los subsistemas.
- Programación estructurada: para dar forma a las necesidades del diseño. Dado que no se va a utilizar un lenguaje orientado a objetos, hace falta dar forma a las necesidades del análisis mediante las estructuras disponibles en la programación estructurada.
- Lenguaje de programación C: usado para reflejar el diseño en un programa compilable y ejecutable. Se ha usado este lenguaje sobre todo por la necesidad de utilizarlo más tarde en RSIM, pero también por la compatibilidad entre sistemas.
- Estructuras de datos: concretamente árboles de búsqueda y derivados de dichos árboles que sean utilizados actualmente en sistemas gestores de bases de datos reales.
- Sistemas operativos y programación paralela: se aplican a la hora de implementar un sistema de almacenamiento concurrente, que servirá de base sólida para el funcionamiento paralelo de la aplicación.

Capítulo 2

Fundamentos

2.1. Análisis de rendimiento

Las especificaciones del estándar TPC-C no son más que las especificaciones de un análisis de rendimiento que incluyen la realización de una herramienta de medida de rendimiento, pero también la preparación de una documentación adecuada así como una elección de parámetros y factores que analizar. Para entender mejor qué es y qué influye en un análisis de rendimiento se va a detallar en qué consiste y qué factores influyen de una manera general.

2.1.1. Objetivos del análisis de rendimiento

Los objetivos de cualquier análisis de rendimiento de un sistema informático o de uno de sus componentes dependen del la situación concreta y de la capacidad, habilidad e intereses de la persona encargada de realizar ese análisis (el analista). Aun con estas dependencias se pueden señalar varios objetivos [5] que suelen buscar en los análisis de rendimiento los cuales son útiles tanto para diseñadores de sistemas informáticos como para el usuario final.

- **Comparar alternativas.** Cuando se necesita adquirir un nuevo sistema informático, uno se enfrenta a diferentes sistemas con diferentes características entre los que elegir. Cada característica de un solo sistema puede afectar al rendimiento y al precio: memoria, número de procesadores, interfaz de red, número de discos, sistema operativo, etc. El objetivo de un análisis de rendimiento en este caso es proporcionar información cuantitativa sobre qué configuración es la más adecuada bajo unas determinadas condiciones.
- **Determinar el impacto de una característica concreta.** Cuando se diseña un nuevo sistema, o cuando se actualiza uno ya existente, se suele necesitar determinar el impacto causado al añadir o eliminar una característica concreta; por ejemplo, el diseñador de un nuevo procesador puede querer determinar cuando es útil añadir una unidad de punto flotante a la arquitectura, o si el tamaño de la caché incluida en el chip es el adecuado. Este es el análisis llamado *comparación antes y después*, ya que se cambia un componente bien definido del sistema.
- **Ajustar el sistema.** El objetivo de un análisis de rendimiento cuando se quieren optimizar los ajustes de un sistema, es encontrar el conjunto de parámetros con sus valores que producen en su totalidad el mejor rendimiento. En los sistemas operativos de tiempo compartido, por ejemplo, es posible controlar el número de procesos que pueden compartir el procesador; el impacto

sobre el rendimiento que perciben los usuarios del sistema puede verse muy afectado por este número y también por la cantidad de tiempo asignado a cada proceso. Muchos otros parámetros del sistema como los tamaños de los almacenes intermedios de red y disco pueden también alterar el rendimiento de un sistema, ya que los impactos en el rendimiento de estos parámetros pueden estar interconectados; el hecho de encontrar el mejor conjunto de parámetros con sus valores adecuados puede ser una tarea difícil.

- **Identificar el rendimiento relativo.** El rendimiento de un sistema informático sólo suele tener un significado concreto dentro del contexto de su rendimiento relativo a alguna otra *cosa*, como otro sistema u otra configuración del sistema; por eso el objetivo en esta situación puede ser cuantificar el cambio en el rendimiento relativo a las antiguas versiones/generaciones del sistema. Otro objetivo puede ser cuantificar el rendimiento relativo a las necesidades de un cliente o el de los sistemas de la competencia.

- **Búsqueda de fallos en el rendimiento.** Buscar fallos en un programa para que su funcionamiento sea lo más correcto posible es lo básico en cualquier aplicación, sin embargo, el análisis de rendimiento se convierte en un problema de búsqueda también, aunque en este caso de rendimiento. Un programa funciona correctamente pero lo hace más lentamente de lo deseado, por lo que el analista en este punto busca identificar las causas por las que el programa no cumple con los requisitos de rendimiento que se esperan de él; una vez los problemas de rendimiento se identifican, puede que exista alguna posibilidad de resolverlos.

- **Establecer unas expectativas.** Los usuarios de un sistema informático pueden hacerse una idea de la capacidad de, por ejemplo, la nueva generación de sistemas informáticos, gracias a que se ha realizado un análisis de rendimiento indicando lo que dichos sistemas son capaces de hacer.

En todas estas situaciones, el esfuerzo involucrado en el trabajo de un análisis de rendimiento debe ser proporcional al coste de escoger la decisión equivocada; por ejemplo, si se están comparando diferentes fabricantes de sistemas informáticos para conocer cual de ellos es mejor en una decisión de compra de muchos equipos, el coste monetario de escoger la empresa equivocada puede ser importante, tanto el coste del sistema en si como las áreas de la empresa que se vean afectadas por esa mala decisión. Se ve claramente que en este caso el análisis tiene que ser muy detallado. Aunque por ejemplo, si se está buscando el mejor PC para un uso personal, el coste de escoger la opción equivocada es mínimo, y el análisis de rendimiento puede limitarse a leer unos pocos artículos de una revista.

2.1.2. Técnicas generales

Cuando uno se enfrenta a un problema de análisis de rendimiento, hay tres técnicas fundamentales que se pueden usar para encontrar la solución adecuada. Estas son: *medidas* de los sistemas actuales, *simulación* y *modelado analítico*. Las medidas de los sistemas existentes normalmente proporcionan los mejores resultados ya que teniendo las herramientas de medida adecuadas, no hace falta hacer ninguna simplificación, sólo queda utilizarlas; lo que hace que los resultados basados en las medidas de un sistema existente sean más creíbles cuando se presentan a otras personas. Aun así las medidas de estos sistemas no suelen ser flexibles y sólo proporcionan información del sistema analizado. Un objetivo común del análisis de rendimiento es caracterizar cómo el rendimiento de un sistema cambia al variar ciertos parámetros; en un sistema existente puede ser difícil, si no imposible, cambiar alguno de estos parámetros, por lo que evaluar el impacto en el rendimiento de dicho parámetros es también

imposible. Medir ciertos aspectos del rendimiento en un sistema actual puede suponer una tarea costosa en tiempo y difícil, así que aunque las medidas de sistemas reales nos ofrezcan datos muy fiables, las dificultades y limitaciones existentes hacen que sean necesarias otras técnicas para buscar soluciones.

La simulación de un sistema informático es un programa escrito para modelar características importantes del sistema que está siendo analizado; ya que el simulador no es nada más que un programa, puede modificarse fácilmente para estudiar el impacto de cambios en casi cualquiera de los componentes que se simulan. El coste de una simulación incluye el tiempo requerido para escribir y depurar el programa de simulación así como el tiempo de cada simulación; dependiendo de la complejidad del sistema que se esta simulando, y el nivel de detalle del modelo, estos costes pueden ser relativamente bajos o moderados comparados con el coste de comprar una máquina real en la cual realizar los experimentos.

La principal limitación de un análisis de rendimiento basado en simulaciones es que es imposible modelar hasta el más mínimo detalle del sistema que se está estudiando, por lo que se necesita simplificar algunos conceptos y realizar algunas asunciones para que sea posible escribir un modelo que se pueda ejecutar en un tiempo razonable. Estas simplificaciones pueden limitar la exactitud de los resultados y por lo tanto reducir la confianza que se puede tener en el modelo comparado con un sistema real, sin embargo, la simulación disfruta de una gran popularidad para el análisis de sistemas informáticos (Véase RSIM por ejemplo) dado el alto grado de flexibilidad y su relativa facilidad de implementación.

La tercera técnica disponible para el analista es el modelado analítico; el modelo analítico es una descripción matemática del sistema, que comparada con una simulación o unas medidas de un sistema real, obtiene unos resultados menos creíbles y menos precisos. No obstante, un modelo analítico simple puede proveernos rápidamente de pequeños detalles dentro del conjunto general del sistema o de alguno de sus componentes; estos pequeños detalles internos pueden ser usados para ayudar a centrar unas medidas más detalladas o un modelo de simulación más concreto. Un modelo analítico puede ayudar a discernir si los resultados producidos por un simulador o los valores obtenidos de un sistema real son razonables o no.

Un ejemplo

El retraso que se observa en una aplicación cuando accede a memoria puede tener un gran impacto sobre el tiempo de ejecución total; medidas directas de dicho retraso en una máquina existente pueden ser difíciles de obtener, aunque dado que los pasos que se ejecutan para acceder a una sistema de memoria complejo no están accesibles normalmente desde una aplicación de usuario, un programador avanzado puede ser capaz de escribir un simple programa que ejecute porciones específicas de código en dicha arquitectura de memoria para poder obtener algunos parámetros importantes del sistema. Por ejemplo, el tiempo de ejecución de un programa sencillo que referencia una y otra vez la misma variable puede usarse para estimar el tiempo que se necesita para acceder al primer nivel de caché; de la misma manera, un programa que siempre fuerza fallos en la caché puede servir para medir indirectamente el tiempo de acceso a la memoria principal. Desafortunadamente, el impacto de esos parámetros del sistema en el tiempo de ejecución de una aplicación completa es muy dependiente de las referencias a memoria que realice dicha aplicación, y obtener un patrón de esas referencias puede ser difícil.

Por otro lado, la simulación es una potente técnica para estudiar el comportamiento de los sistemas de memoria, dado su alto grado de flexibilidad, cualquier parámetro de la memoria incluida la granularidad de la caché, los tiempos de acceso relativos, los tamaños de la caché y la memoria, etc, puede ser fácilmente alterado para estudiar su impacto en el rendimiento. Puede ser todo un reto

modelar completamente el solapamiento de los retrasos de memoria, y la ejecución de otras instrucciones en procesadores que tiene sistemas como la ejecución fuera de orden, predicción de saltos, acceso a caché no bloqueantes, etc. Aun con las simplificaciones y asunciones, los resultados de una simulación detallada proporcionan detalles internos útiles del efecto de la memoria en el rendimiento de un programa concreto.

Por último, se puede desarrollar un modelo analítico simple: Sea t_c el retardo observado al acceder a una dirección de memoria si la memoria a la que accedemos se encuentra en la caché, sea t_m el retardo al acceder a la memoria principal, sea h el ratio de aciertos de la caché (la cantidad de accesos a memoria que son resueltos por la caché sin acceder a la memoria principal), por lo que el ratio de fallos en la memoria será $h - 1$. Podemos concluir que el tiempo medio necesario para un acierto en la caché es $h * t_c$, mientras que el tiempo medio para los fallos es $t_m * (h - 1)$. El modelo simple del tiempo medio de acceso a memoria es:

$$t_avg = h * t_c + (1 - h) * t_m \tag{2.1}$$

Para aplicar este modelo simple a un programa específico se necesita conocer el ratio de aciertos (h) para ese programa; así como los valores de t_c y t_m para el sistema, que se pueden encontrar en las especificaciones del fabricante o pueden ser obtenidos mediante medidas como se describió anteriormente; por último el ratio de aciertos para un programa normalmente es más difícil de obtener. En definitiva, un modelo simple puede aportar información sobre los efectos de incrementar ciertos parámetros en el sistema.

Resumen

Vamos a ver un resumen de estas 3 técnicas así como de los parámetros que las diferencian (Cuadro 2.1).

Característica	Modelo analítico	Simulación	Medidas reales
Flexibilidad	Alta	Alta	Baja
Coste	Bajo	Medio	Alto
Credibilidad	Baja	Media	Alta
Exactitud	Baja	Media	Alta

Cuadro 2.1: Tabla resumen de las técnicas de análisis

La *flexibilidad* de una técnica indica cómo de fácil es cambiar la configuración del sistema que se está estudiando; el *coste* corresponde al tiempo y dinero necesario para realizar el experimento adecuado dependiendo de la técnica; la *credibilidad* de una técnica es alta si los resultados producidos usando esa técnica son lo más reales que se pueda. Es más fácil para alguien creer que el tiempo de ejecución de una aplicación estará dentro de un rango que uno puede demostrar en una máquina actual que el rango obtenido de una simulación; por la misma regla, son más reales los resultados obtenidos de una simulación que de un modelo analítico. Por último, la *exactitud* de una técnica indica cómo de cercanos son los resultados a los de un sistema real.

La elección de una solución específica depende del problema que se quiera resolver; por lo que una de las habilidades que debe desarrollar un analista del rendimiento de sistemas informáticos, es la de determinar la técnica más apropiada dada una situación concreta.

2.1.3. Errores comunes en los análisis de rendimiento

Para favorecer el uso de la metodología adecuada para evaluar el rendimiento, en esta sección se van a exponer una serie de errores que en los que se incurre de manera habitual en muchos proyectos de evaluación de rendimiento. La mayoría de errores listados no se cometen de manera intencionada, sino que ocurren por errores de concepción, suposiciones o falta de conocimiento de las técnicas de medida de rendimiento.

1. **No tener objetivos.** Tener alguna meta es la parte más importante de cualquier esfuerzo, y sin ella dicho esfuerzo esta avocado al fracaso; en este aspecto los proyectos de medida de rendimiento no son ninguna excepción. La necesidad de una meta puede sonar obvio pero muchos proyectos comienzan sin ningún objetivo definido; por ejemplo, una persona dedicada a analizar el rendimiento que es contratada e insertada en el equipo de diseño, puede empezar a modelar un diseño, pero cuando se le pregunta por los objetivos el analista suele contestar que el modelo es el que ayudará a resolver las preguntas que puedan surgir. Es muy común oír que los modelos son lo suficientemente flexibles para ser adaptados a diferentes problemas, pero la realidad es que se hacen necesarios modelos concreto con objetivos concretos; las medidas utilizadas y la carga de trabajo empleada dependen al final del objetivo. Qué parte del sistema necesite ser estudiada varía dependiendo del problema, por ello antes de escribir la primera línea de código o formula de un modelo, o antes de establecer un experimento que haga una medición, es importante que el analista entienda el sistema e identifique el problema que necesita ser resuelto, lo que ayudará a elegir correctamente: la métrica, la carga de trabajo y la metodología en general.Establecer unos objetivos no es un trabajo sencillo, ya que los problemas que se presentan en las medidas de rendimiento suelen ser vagos y abstractos.

2. **Objetivos parciales.** Otro error común es declarar de manera implícita o explicita cierto *partidismo* a la hora de indicar los objetivos iniciales. Por ejemplo, si nuestro objetivo es "Mostrar que nuestro sistema es mejor que los demás", el problema se convierte en la búsqueda de medidas y cargas de tal manera que nuestro sistema salga siempre vencedor, en vez de buscar las medidas y cargas de trabajo que comparan realmente los dos sistemas. Por ello una regla muy importante es ser imparcial y no tener ideas preconcebidas para obtener unas conclusiones independientes y no guiadas por ciertas creencias iniciales.

3. **Mala aproximación al problema.** Normalmente un analista adopta una aproximación mala cuando no tiene ningún sistema o metodología para entender el problema; escoger parámetros, factores, medidas y cargas de trabajo de manea aleatoria, lo que lleva a conclusiones imprecisas. Hace falta un método para comprender el problema paso por paso, para identificar los objetivos, parámetros, factores, medidas y cargas de trabajo de manera adecuada.

4. **No entender el problema.** Un analista con poca experiencia siente que no se ha hecho nada hasta que se ha construido un modelo y se tienen algunos resultados numéricos; pero con el tiempo uno se da cuenta de que gran parte del esfuerzo del análisis se invierte en definir correctamente el problema, pudiendo consumir hasta un 40 % del tiempo [3]. Como dice el refrán *un problema bien explicado ya tiene la mitad resuelto*, del resto, una gran parte del tiempo se invierte en alternativas de diseño, interpretación de resultados y presentación de conclusiones; el desarrollo del modelo en si es una pequeña parte del proceso. Así como un coche o un tren es un medio para llegar a un lugar concreto, y no un fin en si mismos, los modelos son los medios para alcanzar las conclusiones, no son el resultado final. Un analista experto en el modelado

de medidas de rendimiento pero que no se esfuerza en entender el problema concreto, encuentra que sus modelos son ignorados por aquellas personas que necesitan unos datos para tomar decisiones, ya que ellos necesitan una ayuda en sus decisiones, no un modelo.

5. **Métrica de rendimiento mal elegida.** Una métrica se refiere al criterio usado para cuantificar el rendimiento del sistema, por ejemplo las métricas más utilizadas son la capacidad de procesar datos (throughput) y el tiempo de respuesta. La elección de las métricas adecuadas depende de los servicios que sistema vaya a proveer; por ejemplo el rendimiento de una CPU se compara basándose en su capacidad de procesar datos, lo que se suele medir en términos de millones de instrucciones por segundo (MIPS); aun así, comparar los MIPS de dos CPU de diferente arquitectura como RISC (Reduced Instruction Set Computers) y CISC (Complex Instruction Set Computers), no tiene mucho sentido ya que las instrucciones de cada CPU son totalmente distintas. El error usual es utilizar métricas fácilmente calculables en vez de métricas relevantes que son más difíciles de calcular.

6. **Carga de trabajo no representativa**. La carga de trabajo usada para comparar dos sistemas debe ser representativa del uso actual y real del sistema; por ejemplo, si los paquetes en una red son una mezcla de dos tamaños: cortos y largos, la carga de trabajo para comparar dos redes debe consistir tanto en paquetes cortos como largos. La elección de la carga de trabajo tiene un impacto significativo en los resultados del estudio de rendimiento, ya que la carga equivocada puede dar conclusiones erróneas.

7. **Técnica de evaluación equivocada**. Hay tres técnicas de evaluación: medir, simular y modelar analíticamente. Los analistas normalmente tienen preferencia por una de estas tres técnicas, y la utilizan constantemente; aplicando otro refrán *Cuando aprendes a manejar un martillo todos los problemas son clavos*, pero el basarse en una sola técnica produce un modelo que aunque sea el que mejor se sabe resolver no tiene que ser el que mejor resuelva el problema, introduciendo fenómenos que no están en el sistema real y dejando fuera parámetros que pueden ser importantes.

8. **Pasar por alto parámetros importantes.** Es una buena idea hacer una lista lo más completa posible de componentes del sistema y de la carga de trabajo que afecta a su rendimiento, estas características son los denominados parámetros; por ejemplo, para un sistema operativo los parámetros pueden ser: la cantidad de tiempo que un proceso esta en la CPU, o el tamaño de memoria ocupado por el trabajo actual. Estos parámetros de carga pueden incluir el número de usuarios, patrones de llegada de peticiones, prioridad, etc; es trabajo del analista escoger un grupo de valores para cada uno de esos parámetros; ya que los resultados finales del estudio dependen de esa elección, el hecho de pasar por alto un solo parámetro importante puede hacer que los resultados no tengan ninguna validez.

9. **Pasar por alto factores importantes.** Cuando un parámetro varía en el estudio, se le llama factor; por ejemplo, de los parámetros del punto anterior, el número de usuarios puede ser un factor y el resto de parámetros permanecer fijos. No todos los parámetros tienen el mismo efecto en el rendimiento, por lo que es importante identificar aquellos parámetros que si varían afectan de manera importante al rendimiento del sistema, y usar esos parámetros como factores. Un ejemplo, si la tasa de llegada de paquetes en una red afecta más al tiempo de respuesta que el tamaño de los paquetes, será mejor utilizar varias tasas de llegada de paquetes en el estudio que varios tamaños de paquetes.

A la hora de elegir un factor entre los parámetros, aquellos factores que están bajo el control del usuario o la persona que toma las decisiones y que pueden ser fácilmente cambiados, son los mejores factores posibles ya que no merece la pena malgastar tiempo comparando alternativas que el usuario final no puede adoptar debido a que no puede cambiarlas de ninguna manera. También es importante entender las características aleatorias de algunos componentes del sistema que pueden afectar en el rendimiento, y muy probable que algunos de esos parámetros sean desconocidos para el analista (por ejemplo el funcionamiento del programador de acceso a disco del sistema), por lo que aunque puedan tener importancia, el analista los desechará por no conocerlos. La elección de factores debe basarse en su importancia no en el conocimiento que se tenga sobre ellos.

10. **Diseño del experimento mal realizado.** Este diseño está relacionado con el número de medidas o de simulaciones que se van a realizar y los valores de los parámetros en cada experimento. La elección correcta de estos valores ayuda a sacar más información de cada experimento, de esta manera no se desperdicia el tiempo ni los recursos utilizados. Por ejemplo: diseñando los experimentos de manera ingenua, en cada experimento se altera cada factor de uno en uno, lo que puede llevar a equivocaciones si dos parámetros son dependientes y no se cambian a la vez.

11. **Nivel de detalle inadecuado.** El nivel de detalle usado a la hora de modelar un sistema tiene un gran impacto en la formulación del problema, debido al exceso de datos o a la falta de datos. Para comparar alternativas que simplemente son pequeñas variaciones de un mismo modelo es mejor indicar esas variaciones que hacer un modelo de más alto nivel. Por otro lado, cuando se comparan alternativas muy diferentes, es bueno usar un modelo de alto nivel que permita comparar alternativas muy diferentes de manera rápida. El error que se suele cometer es detallar demasiado cuando se necesita un modelo de alto nivel, y abstraerse cuando se necesita un modelo de bajo nivel.

12. **Ignorar el análisis de los resultados.** Uno de los mayores problemas con las medidas en los proyectos de medida de rendimiento, es que los analistas que los realizan son buenos con las técnicas de medición pero no tienen base, conocimientos y/o experiencia analizando los datos. Recogen gran cantidad de datos pero no saben qué analizar o cómo interpretarlos, de tal manera que no se obtiene un resumen de los datos y solo se entregan los datos en bruto. Por lo que se hace necesario tener un equipo que conozca cómo analizar esos datos.

13. **Análisis equivocado.** Hay un buen número de errores comunes a la hora de realizar medidas, simulaciones y modelos analíticos, como por ejemplo simulaciones cortas, medias demasiado alteradas, etc.

14. **Análisis insensible.** Es habitual que los analistas pongan demasiado énfasis en los resultados de sus análisis, presentándolos como hechos más que como evidencias. El hecho es que los resultados pueden ser sensibles a la carga de trabajo y a parámetros del sistema, y esto se suele obviar; por lo que sin un análisis sensible a estos parámetros uno no puede estar seguro si las conclusiones cambiarían si el análisis se realiza con una configuración ligeramente diferente, además sin un análisis sensible a los parámetros, es difícil acceder a la importancia relativa de los diferentes parámetros.

15. **Ignorar los errores en la entrada.** Los parámetros de interés algunas veces no pueden ser medidos directamente, por lo que se toman medidas de otra variable y se usa para estimar el

parámetro deseado; por ejemplo, en una red de ordenadores donde los paquetes son almacenados en una lista enlazada de almacenes temporales (buffers), y cada almacén tiene un tamaño determinado, dado el número de almacenes requerido para almacenar los paquetes es imposible predecir el número de paquetes o lo que ocupa exactamente cada paquete. Estas situaciones introducen cierto desconocimiento de los datos de entrada, y el analista necesita ajustar el nivel de confianza de los datos de salida de ese modelo ya que los datos de entrada no son muy fiables.

16. **Valores extremos.** También hay que tener cuidado con valores extremos, los llamados outliers, a la hora de realizar la estadística correspondiente, ya que pueden afectar mucho a los resultados; pero para poder descubrir esos valores extremos, y sobre todo cuales se pueden ignorar y cuales no, hace falta un conocimiento profundo y cuidadoso del sistema que esta siendo analizado.

17. **Asumir un futuro sin cambios.** Normalmente se asume que el futuro será el mismo que el pasado, por lo que se usa un modelo basado en la carga y rendimiento observado en el pasado para predecir el rendimiento en el futuro. Se asume que en el futuro la carga y el rendimiento del sistema será el mismo, pero dependiendo de la cantidad de tiempo y de que se quiera asumir, esto puede no ser verdad.

18. **Ignorar la variabilidad.** Analizar sólo la media de rendimiento y no analizar la variabilidad es lo más sencillo, pero no es imposible analizarla. Si la variación es alta, la media por si sola no ayuda en nada a la hora de tomar decisiones; por ejemplo, el uso diario de ancho de banda puede no significar nada, pero el uso por horas puede aportar información útil.

19. **Análisis demasiado complejo** Si dos análisis dan la misma conclusión, es preferible utilizar aquel análisis que es más simple y fácil de explicar. Metas demasiado ambiciosas y modelos demasiado complejos dan lugar a análisis difícilmente entendibles; es mejor empezar con modelos o experimentos simples, e ir introduciendo complicaciones poco a poco hasta llegar a un nivel de detalle aceptable.

    Hay que tener cuidado a veces con los modelos que se realizan en el ámbito académico; muchas veces en estos círculos se tiende a intentar modelar y resolver problemas complejos, más que a intentar dar soluciones a modelos simples. En el mundo industrial, donde la gente que toma decisiones no tiene ningún interés por las técnicas utilizadas y donde existe hay poco tiempo para realizar los análisis, se necesitan modelos simples y gente capaz de sacar conclusiones aceptables de esos modelos simples.

20. **Presentación inadecuada de los resultados.** Una lista que no produce resultados útiles es un fallo, y esto normalmente se debe a que el análisis expone unos resultados que no son entendibles por aquellos que los van a utilizar. Desde diseñadores de un sistema, compradores o patrocinadores de un proyecto; vender los resultados a los que tomas las decisiones es labor del analista, por lo que se necesita un especial cuidado a la hora de usar palabras, imágenes y gráficos para explicar los resultados. La medida correcta para el rendimiento de un sistema no es el número de análisis realizado sino aquellos análisis que han servido de algo a las personas que los han utilizado.

21. **Ignorar aspectos sociales.** Para presentar correctamente los resultados hacen falta dos habilidades: por una lado saber hablar y escribir y por otro saber modelar y analizar. Debido a que es importante que los resultados sean aceptados y entendidos se requiere cierta ingeniería social

para que los destinatarios puedan creer, entender y aceptar los resultados. También existe la necesidad de comunicación entre miembros del equipo, exponer e intercambiar resultados, de ahí que una de las habilidades necesarias del analista sea la de tener facilidad de intercambiar resultados con otras personas.

22. **Omitir asunciones y limites.** Todo análisis tiene aspectos asumidos y limitaciones que normalmente no se incluyen en el informe final, lo que puede llevar al usuario a aplicar el análisis en otro contexto donde esas asunciones no son válidas; a veces los analistas listan las asunciones al principio del informe, pero olvidan las limitaciones al final y hacen conclusiones sobre entornos en los que el análisis no es aplicable.

Esta pequeña discusión sobre errores comunes hace posible presentar una lista de preguntas que se pueden realizar de manera general sobre cualquier análisis de rendimiento; todas estas cuestiones tienen que responderse afirmativamente.

1. ¿Está el sistema definido correctamente?
2. ¿Están los objetivos definidos de manera imparcial?
3. ¿Se han seguido sistemáticamente todos los pasos del análisis?
4. ¿Se entiende el problema correctamente antes de analizarlo?
5. ¿Son relevantes las medidas de rendimiento en el problema actual?
6. ¿Es la carga de problema la adecuada para este problema?
7. ¿Se ha escogido correctamente la técnica de evaluación?
8. ¿Está completa la lista de parámetros que afectan al sistema?
9. ¿Están todos los parámetros que afectan al rendimiento marcados como factores que serán variados?
10. ¿El experimento diseñado es eficiente en términos de tiempo y resultados?
11. ¿Es el nivel de detalle adecuado?
12. ¿Los resultados obtenidos están presentados con un análisis y con una interpretación?
13. ¿La estadística del análisis es correcta?
14. ¿Es el análisis sensible a los parámetros y carga del sistema?
15. ¿Los errores en la entrada producen solamente pequeños cambios en los resultados?
16. ¿Se han identificado y tratado correctamente los valores extremos?
17. ¿Se han modelado los posibles cambios que pueden ocurrir en un futuro en el sistema y en el modelo?
18. ¿Se ha tenido en cuenta que cantidad de variación de la entrada existe?
19. ¿Se ha tenido en cuenta que cantidad de variación hay en los datos analizados?

20. ¿Es el análisis fácil de explicar?
21. ¿Está la presentación adaptada a la audiencia?
22. ¿Se han presentado los resultados de manera gráfica tanto como se ha podido?

2.2. Métricas del rendimiento

2.2.1. ¿Qué es una métrica de rendimiento?

Antes de entender cualquier aspecto del rendimiento de un sistema informático, hay que concretar exactamente que cosas es útil e interesante medir. Normalmente las características básicas que se miden en un sistema informático son:

1. La cantidad de veces que ocurre un evento.
2. La duración de un intervalo de tiempo.
3. El valor de un parámetro concreto.

Por ejemplo, puede que necesitemos contabilizar cuántas veces un procesador inicia una petición de entrada/salida, cuánto tiempo dura cada una de esas peticiones, o cuántos bits se transmiten o almacenan en cada petición.

Para esos valores que deseamos medir, podemos obtener el valor actual del parámetro adecuado para describir el rendimiento del sistema. Dicho valor se llama *métrica de rendimiento* o unidad de medida del rendimiento.

Si estamos interesados concretamente en el tiempo, cantidad o tamaño del valor medido, podemos usar el valor directamente como nuestra unidad de medida del rendimiento del sistema, aunque lo normal es normalizar y relacionar dichos datos, como por ejemplo: operaciones por segundo. Este tipo de unidad se llama *unidad de ratio* o *capacidad de proceso*, y se calcula dividiendo la cantidad de eventos que ocurren en un intervalo de tiempo entre dicho intervalo de tiempo. Ya que estas medidas están normalizadas a una unidad de tiempo común, como pueden ser los segundos, son útiles para comparar diferentes mediciones realizadas a lo largo de diferentes intervalos de tiempo.

El escoger una métrica de rendimiento adecuada depende de los objetivos y del contexto actual, así como del coste que se requiere para obtener la información necesaria para dicha métrica. Por ejemplo, suponiendo que se necesita escoger entre dos sistemas informáticos para usar durante un corto periodo de tiempo y una tarea especifica (como usar un procesador de texto); dado que la penalización por hacer una mala elección no es mucha, el escoger la frecuencia de reloj como métrica de rendimiento puede ser una buena opción; pero desde que la frecuencia de reloj no es una métrica aceptable de rendimiento, hay que escoger una unidad mejor, sobre todo si se necesita comparar una gran cantidad de esos sistemas informáticos. Hay que tomarse un tiempo para escoger una unidad de medida que sea rigurosa y real, lo que nos lleva a preguntarnos ¿Cual es una buena unidad de medida?

2.2.2. Características de una buena métrica de rendimiento

Se pueden utilizar muchas y muy diferentes unidades de medida de rendimiento para describir un sistema informático; algunas han sido y son muy utilizadas, como los MIPS (Millones de instrucciones por segundo) y los MFLOPS (Millones de operaciones en punto flotante por segundo). Pero el

tiempo demuestra que no todas estas medidas son buenas, en el sentido que pueden llevar a confusión y/o errores; por ello es necesario entender las características de una *buena* métrica de rendimiento, lo que nos ayudará a decidir qué unidades de las que están disponibles se utilizaran en cada situación particular, o si es necesario desarrollar una nueva unidad de medida.

Una unidad de medida que satisfaga todos los requisitos [5] que se exponen a continuación, es una unidad de medida útil para un analista, y le permite realizar comparaciones precisas y detalladas entre diferentes mediciones. Este criterio fue desarrollado observando los resultados de muchos análisis de rendimiento a lo largo de muchos años; no es que sean unos requerimientos absolutos sino que al no cumplirse pueden inducir a conclusiones erróneas.

1. **Linealidad.** Desde que los humanos tendemos a pensar en términos lineales, el valor de la unidad de medida debe ser proporcional al rendimiento de la máquina; esto quiere decir que si el valor cambia una cierta proporción, el rendimiento del sistema debe variar en dicha proporción. Esto es lo más obvio de las medidas, por ejemplo, si un sistema es el doble de rápido (usando una unidad de medida de velocidad) que otro, uno espera que sus tareas se ejecuten en la mitad de tiempo, y si fuera el triple de rápido, se esperaría completar las tareas en un tercio del tiempo original.

   No todas las métricas son lineales, las métricas logarítmicas como los dB usados para describir la intensidad, por ejemplo del sonido; esta unidad no es lineal, y el incremento de una unidad significa el incremento en 10 de la magnitud observada. Estas medidas no son malas, pero las medidas lineales son más intuitivas a la hora de presentar resultados en los análisis de rendimiento.

2. **Confianza.** Una unidad de medida de rendimiento se considera de confianza si un sistema A siempre adelanta a otro sistema B cuando los valores correspondientes en dicha unidad para los dos sistemas indican que A debe adelantar a B. Pongamos un ejemplo, tenemos una unidad nueva llamada PICS que mide el rendimiento de programas de procesamiento de texto; medimos un sistema A y encontramos que los PICS para ese sistema son 128, y el PICS para el sistema B es 97; la unidad de medida PICS es de confianza si siempre el sistema A es mejor que el sistema B a la hora de ejecutar programas de procesamiento de texto.

   Mientras que esta característica parezca obvia y no necesaria, muchas veces se usan unidades de medida que no satisfacen este requerimiento; por ejemplo los MIPS o la velocidad de reloj, son unidades de medida que no dan confianza. No es raro encontrar procesadores que aun teniendo una frecuencia de reloj más baja, son mucho más potentes que otros que tienen frecuencias de reloj más altas.

3. **Repetibilidad.** Una unidad de medida es repetible si el mismo valor usando esa unidad es medible cada vez que se ejecuta el mismo experimento, lo que también implica que dicha unidad sea determinista.

4. **Facilidad de medida.** Si una unidad no es fácil de medir, es difícil que alguien la use; y cuanto más difícil sea de medir directa o indirectamente, más posibilidades hay de que dicha medida sea tomada de manera incorrecta; y es peor un valor mal medido (que se contabiliza en los resultados) que una mala unidad de medida (que no se escoge para contabilizar el rendimiento).

5. **Consistencia.** Una unidad de medida consistente es aquella en la que sus unidades y su definición permanecen constantes a lo largo de muchos sistemas diferentes y configuraciones diferentes del mismo sistema. Si las unidades no son consistentes, es imposible utilizarlas para

comparar el rendimiento de sistemas distintos o de distintas configuraciones. Unidades como los MIPS o MFLOPS, no satisfacen ese requerimiento.

6. **Independencia.** Mucha gente compra sistemas informáticos basándose en la comparación de valores de métricas de rendimiento muy comunes, por lo que no es normal encontrar cierta presión en los fabricantes para desarrollar y optimizar sus sistemas y que dicho valor de esa métrica mejore sustancialmente. Para prevenir esta influencia negativa una buena métrica tiene que ser resistente a estos trucos.

2.2.3. Procesadores y sus unidades de rendimiento

Una gran variedad de métricas de rendimiento han sido propuestas y usadas en el mundo de la informática; desgraciadamente muchas de esas unidades de medida no son buenas (como se definió en la sección anterior), o se usan y/o interpretan de manera incorrecta. Se va a listar una serie de métricas muy comunes y se va a analizar su *bondad* respecto a las características anteriores.

La frecuencia de reloj

En muchos anuncios de sistemas informáticos, uno de las partes que más se enfatiza es la frecuencia de reloj del procesador central, lo que implica que se intenta convencer al comprador que un sistema a 250MHz será siempre más rápido que otro a 200MHz. Esta unidad de medida ignora completamente como se realizan los cálculos en cada ciclo de reloj, ignora las complejas interacciones del procesador con el sistema de memoria y la entrada/salida, y sobre todo, ignora que puede que el procesador sea realmente el cuello de botella del rendimiento del resto del sistema.

Analizando las características de la frecuencia de reloj en busca de una buena unidad de rendimiento, encontramos que es una unidad repetible ya que es constante para un sistema dado, es fácil de obtener porque aparece en las especificaciones, es consistente ya que el valor de los MHz se define de la misma manera en todos los procesadores, y es independiente ya que ninguna aplicación puede alterar eso. Pero existen varios problemas y es que no es una unidad lineal y no es una unidad de confianza; ya que como mucha gente puede demostrar, el tener un procesador con un reloj más rápido no significa que el ordenador vaya más rápido que otro con velocidad de reloj inferior. En definitiva, esta unidad no es una buena unidad para medir el rendimiento.

MIPS

La cantidad de datos procesados o la proporción de datos ejecutados es una medida basada en dicha cantidad de datos entre una unidad de tiempo; dado que dicho tiempo suele estar siempre en segundos, estas medidas son útiles para comparar velocidades relativas: un coche a $50m/seg$ va más rápido que uno a $35m/seg$.

La unidad de medida MIPS es un intento de desarrollar una medida proporcional para los sistemas informáticos que permita una comparación directa de su velocidad. Mientras que en el mundo físico la velocidad es medida mediante la distancia recorrida por unidad de tiempo, los MIPS definen la distancia como la ejecución de una instrucción, siendo los MIPS: millones de instrucciones por segundo, y se definen como:

$$MIPS = \frac{n}{t_e * 10^6} \tag{2.2}$$

De esta manera, los MIPS son fáciles de calcular (punto 4), repetibles (punto 3) e independientes (punto 6); desgraciadamente no satisfacen ninguno de los otros puntos que identifican a una buena

medida de rendimiento. No es lineal, ya que doblando los MIPS no se dobla el rendimiento, y no es de confianza ni consistente debido a que no se relaciona mucho con el rendimiento.

El problema estriba en que la unidad MIPS en cada procesador significa una cantidad diferente de proceso realizado, ya que no todos los procesadores realizan el mismo trabajo con una instrucción. Por ejemplo, un procesador puede tener una instrucción de salto que compruebe un bit de condición, y otro en esa misma instrucción además decremente un contador; en este ejemplo se ve claramente como el segundo procesador realiza más trabajo en una misma instrucción. Las diferencias entre la cantidad de proceso de cada instrucción son las diferencias entre los procesadores RISC y CISC, que hacen de MIPS algo inservible como medida de rendimiento; MIPS no es mejor unidad de medida que la frecuencia de reloj.

MFLOPS

Los MFLOPS (Millones de operaciones en punto flotante por segundo) intentan corregir el principal problema de los MIPS, definiendo de manera más precisa la unidad de "distancia" de un ordenador cuando ejecuta un programa. Se define una operación aritmética entre dos números en punto flotante como unidad básica y se calcula:

$$MFLOPS = \frac{f}{t_e * 10^6} \tag{2.3}$$

Donde f es el número de operaciones en punto flotante ejecutadas en t_e segundos.

Si bien los MFLOPS son una mejora de los MIPS ya que los resultados son más claros y fácilmente comparables entre varios sistemas; aun existe un problema con los MFLOPS, y es que pueden dar un valor a un sistema que ejecuta programas que no realizan operaciones en punto flotante. Este programa puede ser la carga real del sistema y no tener relación alguna con esta unidad de medida.

Un problema más sutil que existe en los MFLOPS, es a la hora de ponerse de acuerdo en cómo medir exactamente el número de operaciones en punto flotante en un programa; ordenadores como los Cray, realizan una división en punto flotante usando aproximaciones sucesivas que implican varias multiplicaciones y cálculos; de manera similar un procesador puede calcular el valor de un seno o un coseno en una instrucción y otro necesitar varias instrucciones. ¿cómo contabilizar estas operacionesi?, ¿como una o como múltiples?. Esta cierta flexibilidad a la hora de establecer el número total de operaciones en punto flotante, hace que los MFLOPS no cumplan la característica de independencia de una buena unidad de medida; tampoco resulta de confianza y es inconsistente.

SPEC

Para realizar un estándar de la definición del trabajo realizado por un sistema informático durante un uso típico, un grupo de fabricantes se puso de acuerdo para crear el SPEC (Cooperativa de evaluación del rendimiento de sistemas); este grupo identificó una serie de programas que realizan medidas de operaciones enteras y en punto flotante y que reflejan la mayoría de las veces el uso habitual de una estación de trabajo. Además, y quizás lo más importante, también estandarizaron la metodología para medir y realizar informes de los rendimientos obtenidos ejecutando estos programas

La metodología consiste en estos puntos clave:

- Medir el tiempo necesario para ejecutar cada programa sobre el sistema que está siendo analizado.
- Dividir el tiempo medido para cada programa en el primer paso por el tiempo necesario para ejecutar cada programa en una máquina base, y así normalizar los tiempos de ejecución.

- La media geométrica de todos estos valores produce un solo número que es usado como medida de rendimiento.

Mientras que la metodología SPEC es más rigurosa que usar MIPS o MFLOPS, aun produce una unidad de medida ligeramente problemática, ya que la media de los valores normalizados de los tiempos de ejecución, no está linealmente relacionada con el tiempo de ejecución de un programa normal. Esto hace que el SPEC sea poco intuitivo, y si nos centramos en un programa concreto, puede ocurrir que se ejecute más rápido en una máquina que en otra, teniendo la primera menos SPEC que la segunda, lo que hace que esta medida no sea de confianza.

Por último, y aunque la métrica parezca independiente de influencias externas, es habitual encontrar que los desarrolladores de compiladores optimizan la salida de los mismos para este tipo de aplicaciones, por lo que los tiempos reales se ven a veces muy alterados para un mismo procesador simplemente variando el compilador. Por otro lado, los programas que se incluyen en este benchmark los decide un comité de representantes de fabricantes, que está sometido a mucha presión externa por parte de sus compañías, lo que implica un interés por introducir o modificar aplicaciones para que se ejecuten mejor en ciertos sistemas. Aunque SPEC es un paso adelante, aun falla en llegar al objetivo de una buena métrica de rendimiento.

QUIPS

La unidad de medida QUIPS se ha desarrollado conjuntamente con el programa de benchmark HINT, y es una unidad de medida muy diferente: en vez de definir el esfuerzo utilizado para alcanzar un resultado como medida de rendimiento, la métrica QUIPS define la calidad de una solución como un indicador más significativo; definiéndose la calidad rigurosamente en base a características matemáticas del problema que quiere ser resuelto. Si se divide esta medida entre el tiempo necesario para alcanzar dicho nivel de calidad, se producen los QUIPS (Mejoras de calidad por segundo).

Esta nueva unidad de medida tiene muchas de las características necesarias para ser una buena unidad; la precisión matemática de calidad definida para el problema hace que esta unidad sea insensible a influencias externas y la hace consistente cuando se necesita trasladar a otros sistemas. También es fácilmente repetible y es lineal, ya que la medida de la calidad está linealmente relacionada con el tiempo necesario para obtener la solución.

Estos aspectos son muy positivos, pero como siempre, existen unas posibles dificultades cuando se usa esta métrica como unidad de propósito general. El principal problema es que no siempre es una unidad fiable debido a que abarca pocas cosas: sistema de memoria y unidades de punto flotante; aunque es una buena medida para predecir como se comportará un sistema informático a la hora de realizar tareas matemáticas, no nos dice nada de otro tipo de aplicaciones como las relacionadas con la entrada/salida, caché de instrucciones, o incluso funcionalidades del sistema operativo como la multiprogramación. Los desarrolladores han hecho un buen trabajo con HINT ya que es una unidad fácil de medir y transportable a otros sistemas, pero es difícil cambiar la definición de calidad, lo que hace difícil desarrollar nuevos problemas para centrarse en otros aspectos del rendimiento de un sistema, ya que la definición de calidad está muy relacionada con el problema que se desea resolver; y resolver un nuevo problema es difícil debido a que debe de cumplir todas estas características.

Aun con estos pequeños problemas, QUIPS es un nuevo e importante tipo de medida que define rigurosamente aspectos interesantes del rendimiento a la vez que proporciona suficiente flexibilidad para permitir que nuevas arquitecturas demuestren su capacidad (gracias a su portabilidad). No es que sea una unidad de medida de uso general al 100 % pero funciona en cuanto a capacidad de procesamiento numérico se refiere. Además ha contribuido de manera importante en el desarrollo de medidas de rendimiento más rigurosas.

Tiempo de ejecución

Dado que en última instancia estamos interesados en conocer lo rápido que un programa se ejecuta, la medida fundamental y básica en las medidas de rendimiento es el tiempo necesario que requiere una aplicación para ejecutarse. Es simple, el sistema que tarda menos es el que tiene más rendimiento, y podemos comparar tiempos directamente o derivar de esos tiempos las proporciones necesarias; aun así, sin un modo preciso de medir el tiempo, es imposible analizar o comparar cualquier característica. Es importante conocer cómo medir el tiempo de ejecución de un programa o un trozo de código, y entender las limitaciones de nuestra herramienta de medida.

La técnica más básica para medir el tiempo en un sistema informático es la misma que se utiliza para medir cualquier otro tiempo; con un cronómetro empezaríamos desde cero, aunque en un sistema informático el tiempo suele comenzar desde que el sistema fue encendido o desde una fecha concreta. Medimos un intervalo de tiempo leyendo los valores de dicho contador al principio del proceso que quiere ser medido y al final; siendo el intervalo de tiempo la diferencia entre ambos valores. Si usamos los pasos de reloj del sistema, contabilizaremos dicha diferencia entre los pasos y luego multiplicaremos por el tiempo necesario para un paso del reloj.

Por poner un ejemplo más significativo, vamos a ver un pseudo-programa de cómo se realizaría este cálculo. Considerando la función inicio_contadores() como aquella que inicializa las estructuras necesarias para acceder a los contadores internos del sistema; este temporizador es un simple contador que es incrementado continuamente cada periodo de tiempo definido en la variable ciclo_reloj; leer la variable contador_ciclos significa obtener el valor actual del temporizador.

```
main()
{
     int   i;
     float a;

     inicio_contadores();

     /* Leemos el contador al principio */
     comienzo=contador_ciclos;

     /* Cálculos a medir */
     for(i=0;i<100;i++)
          a=i*a/10;

     /* Contador al final de los cálculos */
     fin=contador_ciclos;

     tiempo_transcurrido=(fin-comienzo)*ciclo_reloj;
}
```

Para empezar a contabilizar una porción de código, se obtiene el valor actual del contador y es almacenado en la variable comienzo; al final del código se vuelve a obtener dicho valor. La diferencia entre esos dos valores es el número total de ciclos de reloj que se han necesitado para completar el trabajo, y el tiempo total no es más que esa cantidad de ciclos multiplicada por el tiempo que necesita cada ciclo para ejecutarse.

Esta técnica para medir el tiempo necesario para la ejecución de un código se la suele llamar *wall clock*, ya que mide el tiempo total que un usuario debe esperar para obtener los resultados producidos

por una aplicación. Esta medida incluye el tiempo necesario en acceder a memoria, esperar entrada/salida, nueva memoria y otras necesidades del sistema que se ejecutan para este proceso; esto nos da una pista de un problema, si el sistema donde se ejecuta esta prueba es de tiempo compartido, puede que los contadores reflejen el tiempo en el que la aplicación ha estado esperando por tiempo de procesador mientras otras tareas del sistema se han ejecutado.

Hay gente que opina que añadir este tiempo extra a las medidas no es justo, y que es mejor medir el tiempo exacto que el proceso utiliza la CPU, el llamado tiempo de uso de CPU, que no incluye el tiempo que el programa está a la espera de la CPU a consecuencia de que otra aplicación está ejecutándose. Lo malo de esta medida es que a veces la aplicación no está en la CPU debido a que esta esperando por datos, y ese tiempo no se contabiliza. Lo importante es dejar bien claro lo que se está midiendo para que el lector del informe pueda valorar si esa información le es o no útil.

Además del consumo extra del sistema, el tiempo de ejecución del mismo programa de una medición a otra puede variar mucho si utiliza datos generados aleatoriamente, o del estado del sistema: ocupación de memoria, estado de los buffers, etc. Lo que hace que el tiempo de ejecución del programa no sea determinista, y hace necesario medir el tiempo de ejecución varias veces e incluir la media y la varianza de dichos tiempos en el informe.

El tiempo medido de esta manera proporciona una unidad de medida que es intuitiva, fiable, repetible, fácil de medir, consistente entre sistemas e independiente de influencias externas; satisface todas las características de una buena medida de rendimiento, por lo que el tiempo de ejecución es una de las mejores métricas de rendimiento que se pueden utilizar a la hora de analizar el rendimiento de un sistema informático.

2.2.4. Otros tipos de medidas de rendimiento

Además de las medidas de rendimiento basadas en tareas del procesador, hay muchas otras medidas que se utilizan en los análisis de rendimiento; por ejemplo, el tiempo de respuesta del sistema, que indica el tiempo necesario en atender una petición del usuario; esta métrica se suele usar para analizar sistemas dedicados al procesamiento de transacciones en línea. La capacidad de procesar trabajo es una medida del número de trabajos u operaciones que son completados por unidad de tiempo; el rendimiento de un sistema de procesamiento de vídeo en tiempo real puede ser medido en términos del número de fotogramas que puede procesar en un segundo; el ancho de banda de una red de comunicaciones cuantifica el número de bits que pueden ser transmitidos a través de la red en un segundo. Como se ha visto, se pueden definir muchas unidades de medidas para adecuarse al problema que se quiere analizar.

2.2.5. Ganancia y mejora relativa

La *Ganancia* (Speedup) y la *mejora relativa* son métricas útiles para comparar sistemas ya que normalizan el rendimiento entorno a una base; aunque normalmente se definan en términos de capacidad de procesamiento o velocidad, también se pueden usar directamente con tiempos de ejecución.

Ganancia

La ganancia de un sistema B respecto a un sistema A esta definida por S_B, A de tal manera que $R_B = S_B, A * R_A$, donde R_A y R_B son las unidades de velocidad (métricas de velocidad) que estamos comparando. Podemos decir que el sistema B es S_B, A veces más rápido que el sistema A. Si definimos la velocidad como la "distancia" entre el tiempo (recordemos que la distancia pueden ser las instrucciones procesadas), tenemos $R_A = D_A/T_A$ y $R_B = D_B/T_B$ donde D_A y D_B son

las distancias y T_A junto con T_B es el tiempo; si suponemos que la distancia (trabajo realizado) es el mismo $D = D_A = D_B$, tenemos la siguiente definición de ganancia del sistema B sobre el sistema A:

$$S_B, A = \frac{R_B}{R_A} = \frac{D/T_B}{D/T_A} = \frac{T_A}{T_B} \tag{2.4}$$

Mejora relativa

La mejora relativa es otra técnica para normalizar el rendimiento, expresando la mejora de un sistema a otro a través de tantos por ciento. Si usamos las medidas de velocidad R_A y R_B con los sistemas A y B, la mejora relativa del sistema B sobre el sistema A se define como:

$$\triangle_B, A = \frac{R_B - R_A}{R_A} \tag{2.5}$$

Si asumimos como antes que el tiempo de cada medida se hace ejecutando el mismo programa y por lo tanto la "distancia" es la misma, $R_A = D/T_A$ y $R_B = D/T_B$, definimos la mejora relativa como:

$$\triangle_B, A = \frac{R_B - R_A}{R_A} = \frac{D/T_B - D/T_A}{D/T_A} = \frac{T_A - T_B}{T_B} = S_B, A - 1 \tag{2.6}$$

Lo más normal es que el valor de $\triangle_B, A$ se multiplique por 100 para expresar la mejora relativa como un porcentaje respecto al sistema base. Esta definición produce valores positivos si el sistema B es más rápido que el sistema A, y valores negativos si el sistema B es más lento.

Ejemplo

Por poner un ejemplo de como aplicar estas dos maneras de normalizar datos, vamos a exponer los datos de cuatro sistemas (Cuadro 2.2), cada uno de los cuales tarda un tiempo distinto en realizar una misma tarea; utilizaremos como sistema base el sistema número uno.

Sistema x	Tiempo de Ejecución T_x	Ganancia $S_x, 1$	Cambio relativo $\triangle_x, 1\,\%$
1	480	1	0
2	360	1,33	+33
3	540	0,89	-11
4	210	2,89	+129

Cuadro 2.2: Ejemplo de aceleración y cambio relativo

2.2.6. Métricas de medias y métricas de objetivos

Una de las características más importantes de una métrica de rendimiento es que se pueda confiar en sus valores, por lo que un problema con muchas de las medidas que hemos comentado es que los valores que se obtienen pueden no ser útiles. Lo que hace a una unidad de medida que sea de confianza es la precisión y la consistencia con que la que se realiza cada medición hasta llegar al

objetivo; aquellas métricas que lo miden todo, sea útil o no para nuestro análisis, se llaman *Métricas de medias*, frente a aquellas que miden sólo el trabajo útil que son las *métricas de objetivos*.

Para ver la diferencia entre estos dos tipos de métricas, vamos a considerar un problema consistente en el producto de un vector; este problema ejecuta N sumas en punto flotante y N multiplicaciones, con un total de 2N operaciones en punto flotante. Si el tiempo para ejecutar las sumas es t_+ ciclos y el tiempo de las multiplicaciones es t_* ciclos, el tiempo total es $t_1 = N(t_+ + t_*)$ ciclos.

```
s=0;
for(i=1;i<N;i++)
      s=s+x[i]*y[i];
```

El ratio de ejecución de operaciones es:

$$R_1 = \frac{2N}{N(t_+ + t_*)} = \frac{2}{(t_+ + t_*)} FLOPS/ciclo \tag{2.7}$$

Ya que no es necesario realizar la suma en las multiplicaciones en la que alguno de sus miembros es cero, es posible reducir el tiempo total de ejecución si evitamos sumar y multiplicar dichos casos.

```
s=0;
for(i=1;i<N;i++)
      if(x[i]!=0 && y[i]!=0)
            s=s+x[i]*y[i];
```

Si la condición requiere $t_i f$ ciclos adicionales para ejecutarse, el tiempo total necesario para ejecutar el programa es: $t_2 = N[t_i f + f * (t_+ + t_*)]$ ciclos, donde f es la proporción de N cuyos $x[i]$ e $y[i]$ no son cero. Dado que ahora el número total de sumas y multiplicaciones ejecutado es $2Nf$, el ratio de ejecución actual es:

$$R_1 = \frac{2Nf}{N[t_i f + f * (t_+ + t_*)]} = \frac{2f}{t_i f + f * (t_+ + t_*)} FLOPS/ciclo \tag{2.8}$$

Demos valores concretos: $t_i f = 4 ciclos$,$t_+ = 5 ciclos$,$t_* = 10 ciclos$, $f = 10\,\%$, y cada ciclo son 4ns (una velocidad de 250Mhz):

- $t_1 = N(t_+ + t_*) = N(5 + 10) * 4ns = 60Nns$
- $t_2 = N[t_i f + f * (t_+ + t_*)] = N[4 + 0.1(5 + 10)] * 4ns = 22Nns$

Para realizar el mismo trabajo vemos que la ganancia del programa 2, en base al primer programa es $S_2, 1 = 60N/22N = 2.73$.

Pero vamos a calcular los ratios de ejecución de instrucciones de cada programa:

- $R_1 = 2/(60ns) = 33 MFLOPS/ciclo$
- $R_2 = 2(0.1)/(22ns) = 9,09 MFLOPS/ciclo$

Por lo que usando métrica de objetivos tenemos que el programa es un 173 % más rápido ya que ejecuta el mismo trabajo en menos tiempo: 60 ns a 22 ns. Pero si utilizamos una métrica de medias, que indica las operaciones realizadas por ciclo, el primer programa es el *más potente* ya que tiene más MFLOPS que el segundo; esto último ocurre porque las métricas basadas en simples medias dan un crédito a cualquier trabajo realizado, sea útil o no. Con este ejemplo se han visto los problemas de elegir la métrica incorrecta para obtener unas conclusiones.

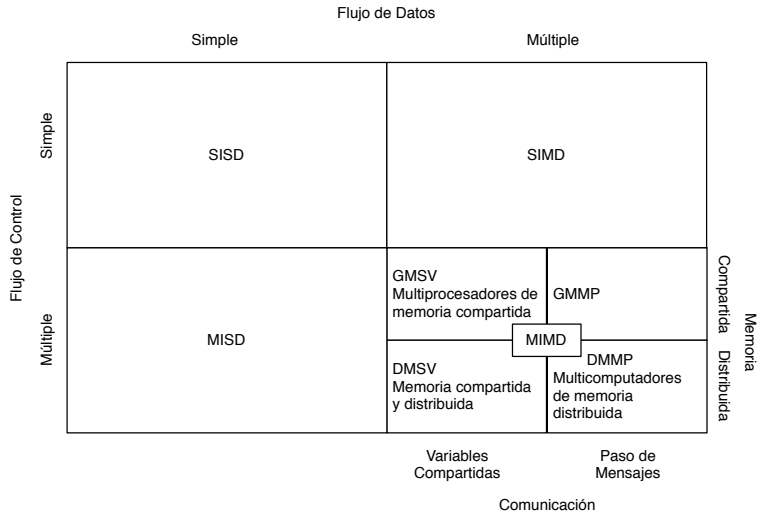

Figura 2.1: La clasificación de Flynn-Johnson de los sistemas informáticos

2.3. Arquitecturas paralelas

No hay que olvidar que, aunque las especificaciones del benchmark TPC-C estén destinadas a un análisis de rendimiento, el rendimiento que vamos a estudiar es el de máquinas multiprocesador; donde en un mismo instante puede haber 16 o 32 procesos ejecutándose de manera simultanea.

Estas arquitecturas paralelas son muy variadas, sería imposible discutir todas y cada una de ellas con detalle: puntos fuertes, características que las distinguen, problemas, etc, por lo que normalmente se trata con modelos abstractos y simples que son aplicables a un bueno número de máquinas existentes (y futuras). El problema es que con estos modelos no se puede predecir el rendimiento de manera precisa, y son tan simples que no representan ninguna máquina real; aun así estos modelos ayudan conceptualmente al desarrollo de algoritmos y al análisis de bastantes problemas de manera que resultan mucho más manejables.

2.3.1. Clasificación básica

De cara al programador hay dos maneras de ver las cosas, el paralelismo *implícito* y el paralelismo *explícito*. El paralelismo implícito no requiere la intervención del programador, ya que el propio código que teclea es paralelo sin necesidad de indicar que parte lo es; en cambio en el paralelismo explícito, el programador tiene constancia de dicho paralelismo e interviene en él de alguna manera.

En cuanto a los sistemas paralelos se pueden dividir en dos grandes categorías: *Flujo de control* y *Flujo de datos* [10]; los primeras están basados en los mismos principios que la máquina de von Neumann, excepto porque múltiples instrucciones pueden ejecutarse a la vez. Los sistemas paralelos de flujo de datos son totalmente diferentes ya que no hay ningún puntero a un grupo de instrucciones que se estén ejecutando en un momento dado; el control está totalmente distribuido.

En 1996, M. J. Flynn propuso una clasificación en 4 categorías de los sistemas informáticos basándose en el flujo de datos y el flujo de instrucciones; esta clasificación se ha convertido en un estándar y se usa en muchos ámbitos. Flynn acuñó cuatro abreviaturas para estas 4 clases: SISD,

SIMD, MISD y MIMD, basándose en el número de flujos de instrucciones (simple o múltiple) y en los flujos de datos (simples o múltiples) [2], Fig. 2.1.

Una breve descripción de cada tipo [17], [18]:

1. **SISD.** (Single Instruction stream, Single Data stream) Flujo único de instrucciones y flujo único de datos. Este es el concepto de arquitectura serie de Von Neumann donde, en cualquier momento, sólo se está ejecutando una única instrucción. A menudo a los SISD se les conoce como computadores serie escalares. Todas las máquinas SISD poseen un registro simple que se llama contador de programa que asegura la ejecución en serie del programa. Conforme se van leyendo las instrucciones de la memoria, el contador de programa se actualiza para que apunte a la siguiente instrucción a procesar en serie. Prácticamente ningún sistema puramente SISD se fabrica hoy en día ya que la mayoría de procesadores modernos incorporan algún grado de paralelismo como es la segmentación de instrucciones o la posibilidad de lanzar dos instrucciones al mismo tiempo (superescalares). Fig.2.2.

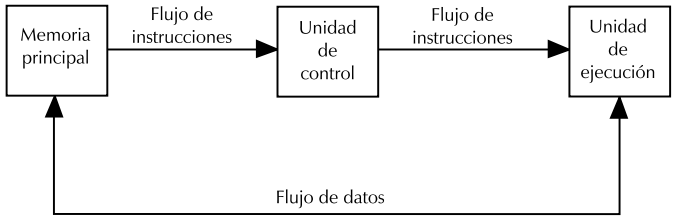

Figura 2.2: Organización Simple Instruction Simple Data (SISD)

2. **SIMD.** (Single Instruction stream, Multiple Data stream) Flujo de instrucción simple y flujo de datos múltiple. Esto significa que una única instrucción es aplicada sobre diferentes datos al mismo tiempo. En las máquinas de este tipo, varias unidades de procesado diferentes son invocadas por una única unidad de control. Al igual que las MISD, las SIMD soportan procesamiento vectorial (matricial) asignando cada elemento del vector a una unidad funcional diferente para procesamiento concurrente. Por ejemplo, el cálculo de la paga para cada trabajador en una empresa, es repetir la misma operación sencilla para cada trabajador; si se dispone de un arquitectura SIMD esto se puede calcular en paralelo para cada trabajador. Por esta facilidad en la paralelización de vectores de datos (los trabajadores formar un vector) se les llama también procesadores vectoriales. , Fig. 2.3.

3. **MISD.** (Multiple Instruction stream, Single Data stream) Flujo múltiple de instrucciones y único flujo de datos. Esto significa que varias instrucciones actúan sobre el mismo y único trozo de datos. Este tipo de máquinas se pueden interpretar de dos maneras. Una es considerar la clase de maquinas que necesitando unidades de procesamiento diferentes recibieran instrucciones distintas operando sobre los mismos datos. Esta clase de arquitectura ha sido clasificada por numerosos arquitectos de computadores como impracticable o imposible, y en estos momentos no existen ejemplos que funcionen siguiendo este modelo. Otra forma de interpretar los MISD es como una clase de máquinas donde un mismo flujo de datos fluye a través de numerosas unidades de proceso. Arquitecturas altamente segmentadas, como los procesadores vectoriales, son clasificados a menudo bajo este tipo de máquinas. Las arquitecturas segmentadas, o encauzadas, realizan el procesamiento vectorial a través de una serie de etapas, cada una

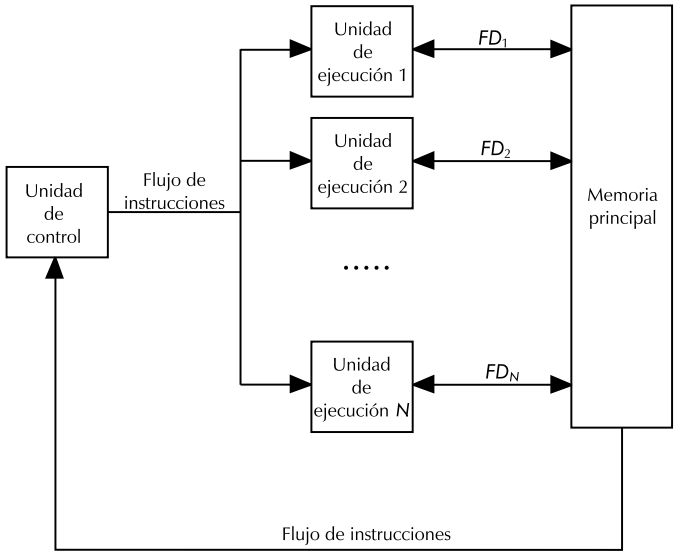

Figura 2.3: Organización Simple Instruction Multiple Data (SIMD)

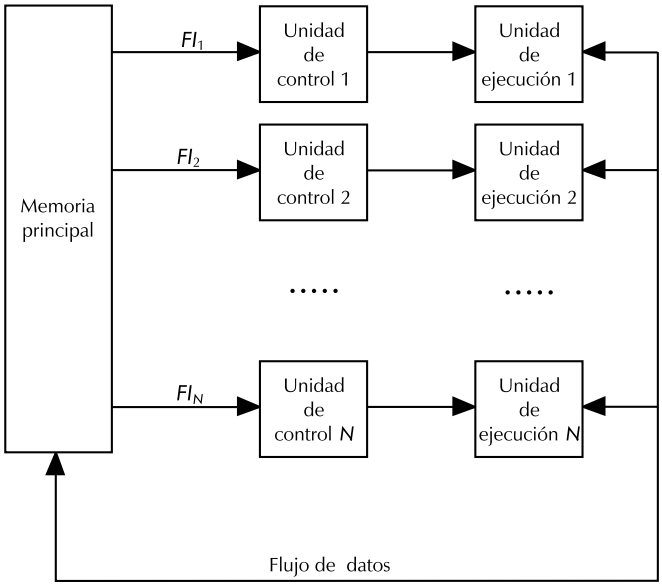

Figura 2.4: Organización Multiple Instruction Simple Data (MISD)

ejecutando una función particular produciendo un resultado intermedio. La razón por la cual dichas arquitecturas son clasificadas como MISD es que los elementos de un vector pueden ser considerados como pertenecientes al mismo dato, y todas las etapas del flujo representan múltiples instrucciones que son aplicadas sobre ese vector, Fig. 2.4.

4. **MIMD.** (Multiple Instruction stream, Multiple Data stream) Flujo de instrucciones múltiple y flujo de datos múltiple. Son máquinas que poseen varias unidades de proceso en las cuales se pueden realizar múltiples instrucciones sobre datos diferentes de forma simultánea. Las MIMD son las más complejas, pero son también las que potencialmente ofrecen una mayor eficiencia en la ejecución concurrente o paralela. Aquí la concurrencia implica que no solo hay varios procesadores operando simultáneamente, sino que además hay varios programas (procesos) ejecutándose también al mismo tiempo, Fig. 2.5.

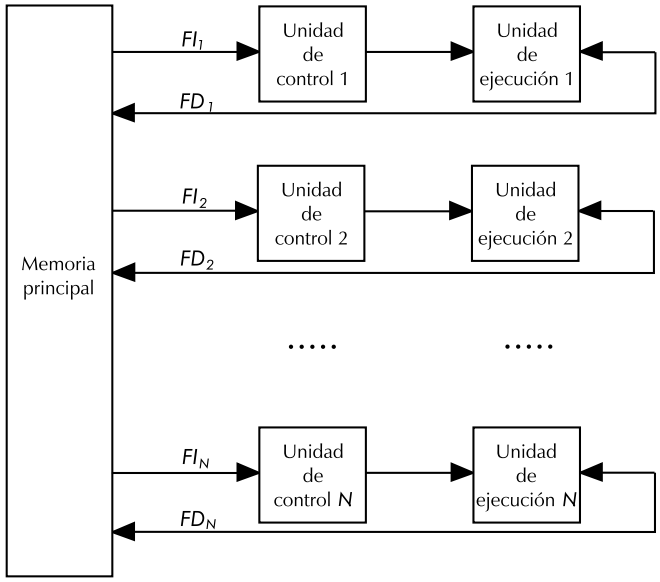

Figura 2.5: Organización Multiple Instruction Multiple Data (MIMD)

2.3.2. Otras clasificaciones

La clasificación de Flynn ha demostrado funcionar bastante bien para la tipificación de sistemas, y se ha venido usando desde décadas por la mayoría de la comunidad informática. Sin embargo, los avances en tecnología y diferentes topologías, han llevado a sistemas que no son tan fáciles de clasificar dentro de los 4 tipos de Flynn. Por ejemplo, las arquitecturas híbridas. Para solucionar esto se han propuesto otras clasificaciones, donde los tipos SIMD y MIMD de Flynn se suelen conservar, pero que sin duda no han tenido el éxito de la de Flynn.

La figura 2.6 muestra una clasificación ampliada que incluye alguno de los avances en arquitecturas de computadores en los últimos años. No obstante, tampoco pretende ser una caracterización completa de todas las arquitecturas paralelas existentes.

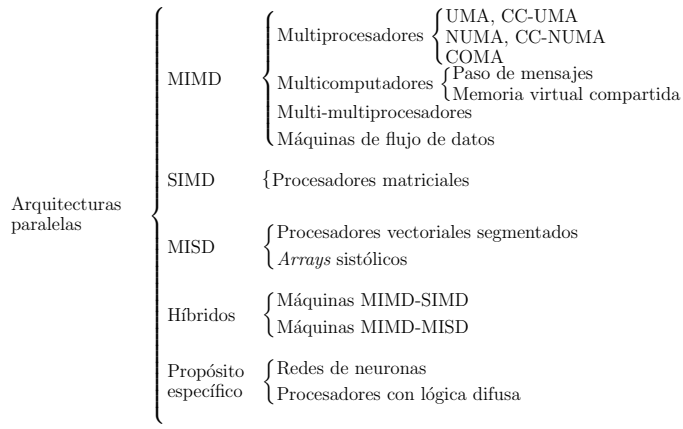

Figura 2.6: Clasificación extendida de las arquitecturas paralelas

Multiprocesadores

De todas estas arquitecturas se va a hacer una mención especial a los **multiprocesadores**. Un multiprocesador se puede ver como un computador paralelo compuesto por varios procesadores interconectados que pueden compartir un mismo sistema de memoria. Los procesadores se pueden configurar para que ejecute cada uno una parte de un programa o varios programas al mismo tiempo.

Dado que los multiprocesadores comparten los diferentes módulos de memoria, pudiendo acceder varios procesadores a un mismo módulo, a los multiprocesadores también se les llama sistemas de memoria compartida. Dependiendo de la forma en que los procesadores comparten la memoria, podemos hacer una subdivisión de los multiprocesadores:

- **UMA.** (Uniform Memory Access) En un modelo de Memoria de Acceso Uniforme, la memoria física está uniformemente compartida por todos los procesadores. Esto quiere decir que todos los procesadores tienen el mismo tiempo de acceso a todas las palabras de memoria. Cada procesador puede tener su caché privada, y los periféricos son también compartidos de alguna manera. A estos computadores se les suele llamar sistemas fuertemente acoplados dado el alto grado de compartición de los recursos. La red de interconexión toma la forma de bus común, conmutador cruzado, o una red multietapa. Cuando todos los procesadores tienen el mismo acceso a todos los periféricos, el sistema se llama *multiprocesador simétrico*. En este caso, todos los procesadores tienen la misma capacidad para ejecutar programas, tal como el Kernel o las rutinas de servicio de entrada/salida. En un *multiprocesador asimétrico* , sólo un subconjunto de los procesadores pueden ejecutar programas. A los que pueden, o al que puede ya que muchas veces es sólo uno, se le llama maestro. Al resto de procesadores se les llama procesadores adheridos (attached processors). La figura 2.7 muestra el modelo UMA de un multiprocesador. Es frecuente encontrar arquitecturas de acceso uniforme que además tienen coherencia caché, a estos sistemas se les suele llamar **CC-UMA** (Cache-Coherent Uniform Memory Access).

- **NUMA.** Un multiprocesador de tipo NUMA es un sistema de memoria compartida donde el tiempo de acceso varía según el lugar donde se encuentre localizado el acceso. La figura 2.8 muestra una posible configuración de tipo NUMA, donde toda la memoria es compartida pero

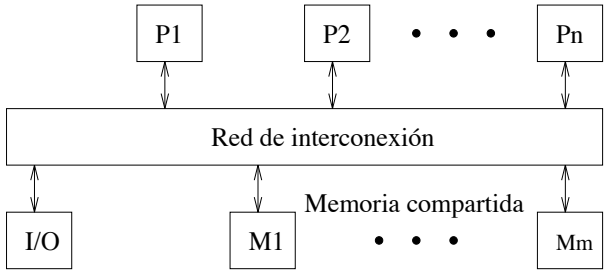

Figura 2.7: Modelo UMA de multiprocesador

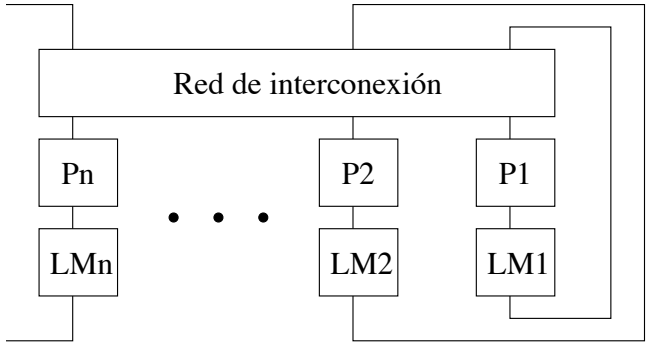

Figura 2.8: Modelo NUMA de multiprocesador

local a cada módulo procesador. Otras posibles configuraciones incluyen los sistemas basados en agrupaciones (clusters) de sistemas como el de la figura que se comunican a través de otra red de comunicación que puede incluir una memoria compartida global.

La ventaja de estos sistemas es que el acceso a la memoria local es más rápido que en los UMA aunque un acceso a memoria no local es más lento. Lo que se intenta es que la memoria utilizada por los procesos que ejecuta cada procesador, se encuentre en la memoria de dicho procesador para que los accesos sean lo más locales que se pueda.

Aparte de esto, se puede añadir al sistema una memoria de acceso global. En este caso se dan tres posibles patrones de acceso. El más rápido es el acceso a memoria local. Le sigue el acceso a memoria global. El más lento es el acceso a la memoria del resto de módulos.

Al igual que hay sistemas de tipo CC-UMA, también existe el modelo de acceso a memoria no uniforme con coherencia de caché **CC-NUMA** (Cache-Coherent Non-Uniform Memory Access) que consiste en memoria compartida distribuida y directorios de cache.

- **COMA.** (Cache Only Memory Access) Un multiprocesador que solo use caché como memoria es considerado de tipo COMA. La figura 2.9 muestra el modelo COMA de multiprocesador. En realidad, el modelo COMA es un caso especial del NUMA donde las memorias distribuidas se convierten en cachés. No hay jerarquía de memoria en cada módulo procesador. Todas

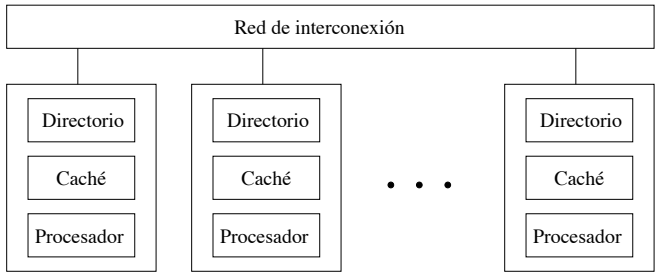

Figura 2.9: Modelo COMA de multiprocesador

las cachés forman un mismo espacio global de direcciones. El acceso a las cachés remotas se realiza a través de los directorios distribuidos de las cachés. Dependiendo de la red de interconexión empleada, se pueden utilizar jerarquías en los directorios para ayudar en la localización de copias de bloques de caché. El emplazamiento inicial de datos no es crítico puesto que el dato acabará estando en el lugar en que se use más.

2.4. El benchmark TPC-C

Los **benchmarks** producidos por el TPC son estándares industriales; el TPC distribuye gratuitamente las especificaciones de estas pruebas, por lo que cualquiera es libre de implementar y publicar un resultado TPC. Sin embargo, hay que aclarar que todos los datos de la implementación tienen que ser enviados al TPC para ser revisados, ya que si un fabricante realiza su prueba TPC de manera incorrecta o de una manera parcial y subjetiva deberá retirar los resultados y no los podrá usar públicamente, por lo que estas reglas protegen a los usuarios de resultados falsos o confusos a la vez que mantienen la credibilidad de los resultados de un benchmark TPC.

Las especificaciones de los *benchmarks* TPC son documentos de unas 30-50 páginas que definen como configurar, ejecutar y documentar un benchmark TPC; además y aparte de las pruebas de rendimiento, los benchmarks TPC necesitan varias pruebas independientes de fiabilidad y seguridad que garanticen a los usuarios que el sistema que está siendo probado se asemeje a uno de un entorno real de producción, por lo que normalmente los benchmark TPC son bastante más complejos y requieren de más tiempo para implementar que otras pruebas más sencillas que pueden realizarse contra una simple cinta de datos.

2.4.1. Características del benchmark TPC-C

El benchmark TPC-C es una carga de trabajo *de procesado de transacciones en línea (OLTP)*; OLTP (online transaction processing) es una clase de trabajo que facilita y dirige ciertas aplicaciones orientadas a transacciones, normalmente para entrada y obtención de datos en un gran número de industrias, como por ejemplo: bancos, pedidos por correo, líneas áreas, supermercados y fabricantes.

Esta prueba es una mezcla de transacciones intensivas de sólo lectura y actualización que simulan el trabajo que se puede encontrar en entornos complejos con aplicaciones OLTP, que suelen tener las siguientes características:

- La ejecución simultanea de varios tipos de transacciones que resultan en una gran complejidad.

- Ejecución de transacciones en línea (on-line) y diferidas
- Múltiples terminales conectados.
- Tiempos de ejecución de aplicaciones moderados.
- Gran cantidad de entrada y salida en disco.
- Transacciones que cumplan las propiedades ACID (Atomic, Consistent, Isolation, and Durable - Atómicas, Consistentes, Aisladas y Resistentes).
- Distribución no uniforme del acceso a los datos a través de claves primarias y secundarias.
- Bases de datos con varias tablas, de varios tamaños, diferentes tipos de atributos y relaciones.
- Con conflictos entre el acceso a datos y la actualización.

La métrica del rendimiento que obtiene el TPC-C es un indicador del rendimiento a la hora de procesar transacciones midiendo el número de pedidos procesados por minuto. Varias transacciones son usadas para simular la actividad que se produce en un negocio a la hora de procesar un pedido, y cada transacción está sujeta a tiempos de respuesta. Las unidades a utilizar para el rendimiento serán *transacciones por minuto* **tpmC**. Si se quiere cumplir estrictamente con el estándar TPC-C aparte de los resultados en unidades tpmC, se debe incluir el precio por cada unidad tpmC así como la disponibilidad y el precio de la configuración usada.

Aunque estas especificaciones expresen la implementación en términos de un modelo de datos relacional con un esquema convencional de bloqueos, la base de datos puede ser implementada usando cualquier sistema de bases de datos, sistema de ficheros, o sistema de almacenamiento de datos que proporcione una implementación funcionalmente equivalente; por lo que aunque en las especificaciones del estándar se digan las palabras “tabla”, “fila” o “columna” no implica un uso de un sistema relacional.

A la hora de comparar los resultados obtenidos por un benchmark TPC-C, hay que indicar que sólo los resultados de una prueba son comparados con los de otra prueba si ambas pruebas son conformes a la misma revisión del TPC-C; por lo que aunque la terminología que se extrae del estándar pueda parecerse a otros benchmarks del TPC o de otras entidades, no implica que estos resultados sean comparables con otros benchmarks.

Este benchmark ofrece un entorno rico que intenta emular muchas aplicaciones OLTP, aunque no siempre es posible reflejar todo el rango de aplicaciones OLTP así como sus necesidades, es necesario indicar que los resultados obtenidos por este benchmark en un sistema pueden no acercarse a las necesidades reales de un sistema/aplicación concreto debido a que el TPC-C no se aproxima a esas necesidades. Con esto se puede afirmar que el rendimiento obtenido en un sistema aplicando este benchmark no se aplica necesariamente a otras cargas de trabajo o entornos, por lo que no se recomienda extrapolar los resultados más allá del sistema donde fueron obtenidos.

Los resultados del benchmark dependen mucho de la carga de trabajo, necesidades de la aplicación y diseño e implementación del sistema, por lo que el rendimiento relativo variará como resultado de estos y otros factores; por lo tanto, TPC-C no debe ser usado como un sustituto para sacar resultados de una aplicación concreta, y mucho menos cuando contemplamos aspectos críticos como planificación de capacidad o evaluación de un producto final.

2.4.2. Algunas pautas para la implementación

Como ya dijimos el objetivo de las pruebas TPC es obtener datos de rendimiento relevantes y objetivos para la industria, para alcanzar estos objetivos hace falta que las pruebas, los sistemas y las tecnologías empleadas:

- Estén disponibles al público.
- Contemplen aspectos importantes en el área de negocio que intenta simularse.
- Exista un número de usuarios importante que implementaría lo que se intenta simular.

Estas características se usan para saber si una implementación particular es un benchmark especial, y son muy generales, se pueden listar otras características más concretas que pueden indicar en mayor o menor medida si el benchmark que se realiza es especifico:

- ¿Está la implementación públicamente disponible, documentada y soportada?
- La implementación tiene restricciones importantes o su uso está muy limitado más allá del TPC-C.
- La implementación o parte de ella se integra muy mal en un producto final.
- La implementación se aprovecha de la naturaleza limitada del benchmark TPC-C de tal manera que no puede ser aplicada generalmente al entorno que intenta simular.
- El fabricante del sistema no aprueba la implementación.
- La implementación necesita de ciertos aspectos excepcionales por parte del usuario final, programador o administrador.
- La implementación no se usa por usuarios finales en el área de mercado que el benchmark simula.

No es necesario que todas estas condiciones sean evitadas/cumplidas para que una implementación pueda tildarse de específica, pero ayudan en menor o mayor grado a clasificarla.

Capítulo 3

Análisis del sistema TPC-C

3.1. Requisitos

Normalmente en un análisis de requisitos se recogen las necesidades del proyecto y se organizan de tal manera que se obtienen una serie de requisitos funcionales y no funcionales. En el caso de un estándar ya escrito, los requisitos vienen ya descritos en el documento del estándar.

Pueden existir requisitos que no se vayan a implementar; pero aquí se plantearán y se analizarán las especificaciones del benchmark TPC-C, por lo que puede que haya ciertos aspectos que aunque debieran de estar en un diseño se presenten en este análisis.

El análisis está realizado en base a las especificaciones del *TPC benchmark C* con versión 5.2 de diciembre del 2003, que se pueden encontrar en `http://www.tpc.org`

3.1.1. Descripción general

El benchmark TCP-C se centra en analizar el rendimiento de entornos OLTP, por lo que se centra en un ejemplo lo más general posible basado en la actividad de una empresa de distribución a gran escala, con el propósito de analizar las características principales de una aplicación de este tipo a la vez que nos centramos en un conjunto más reducido de operaciones, y evitar analizar ciertas operaciones que son poco utilizadas o los recursos que utilizan no son de interés.

La empresa que vamos a analizar con este benchmark es un distribuidor mayorista que dispone de una serie de almacenes distribuidos geográficamente; cada almacén da servicio a 10 zonas, y en cada zona se sirve a 3000 clientes. También, un almacén debe de controlar las existencias de 100.000 productos vendidos por la empresa.

Los clientes contactan con la empresa para realizar pedidos o consultar su estado; estos pedidos están compuestos de una media de 10 elementos distintos, y de un 1 % de estos elementos no se dispone de existencias en el almacen local teniéndose que encargar a otro almacén. Este sistema también se usa para anotar los pagos de los clientes, procesar los pedidos y observar las existencias de los productos para anticiparse a la escasez de estos.

3.1.2. Entidades y relaciones de la base de datos

Los componentes de la base de datos del TPC-C son 9 tablas relacionadas entre si, cuya estructura podemos observar en la figura 3.1. No se indican los atributos de cada entidad ya que serán detallados

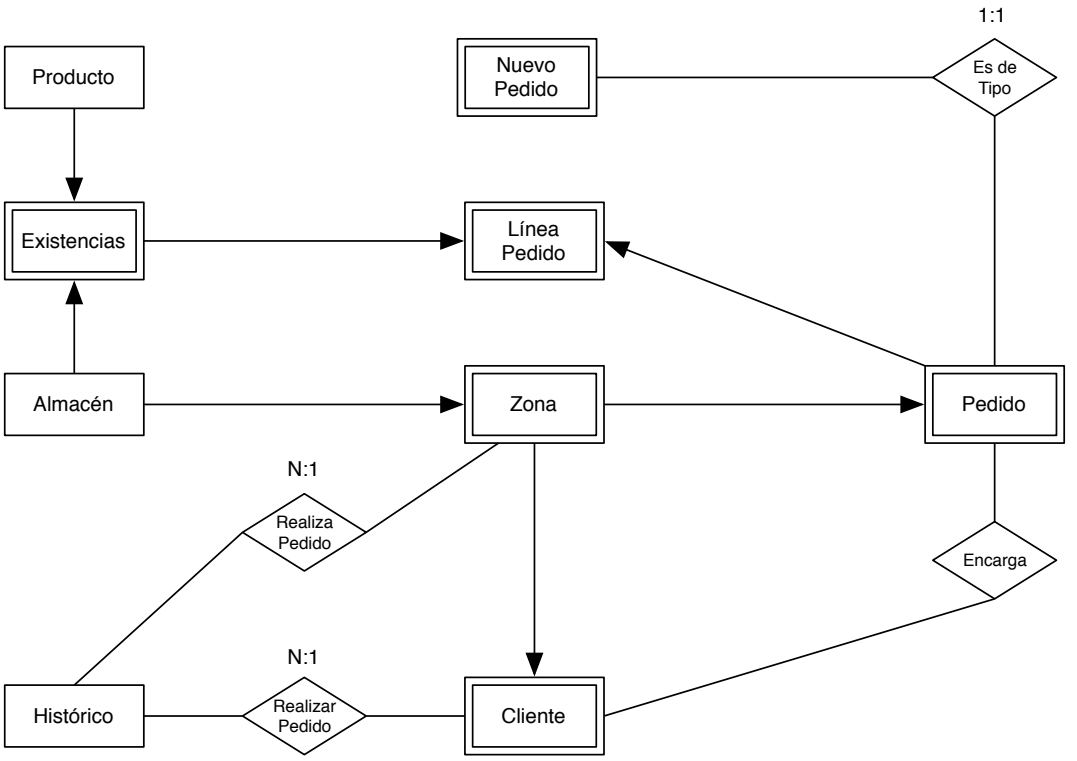

Figura 3.1: Diagrama entidad-relación del sistema de almacenamiento

más tarde junto con el tipo de atributo y la clave de la entidad.

Relaciones

Como se puede ver la mayoría de relaciones son de entidad débil, y muchas entidades dependen de otras; se puede decir que las únicas entidades que siempre perduran son los almacenes y los productos, así como el histórico que es simplemente un historial de operaciones realizadas.

- Un *almacén* está compuesto de 10 *zonas*, por lo que zona es entidad débil de almacén.
- Una *zona* tiene asociados *clientes* y *pedidos*; como antes, clientes y pedidos se convierten en entidades débiles que dependen de zona.
- Un *cliente encarga pedidos* y esos pedidos los encarga en una zona determinada (la suya u otra) por lo que en el *histórico* se almacenan tanto los datos del cliente que realiza el pedido como los de la zona donde se ha realizado el pedido.
- Un *pedido* puede ser de tipo *nuevo pedido*, que indica que aun no se ha enviado al cliente.
- El *pedido* se compone de múltiples *líneas de pedido*, por lo que dichas líneas son entidades débiles de pedido.

- Cada *línea de pedido* indica un producto y su cantidad pero se relaciona con *productos* a través de *existencias*.
- Las *existencias* indican la cantidad de cada *producto* que resta en un *almacén*.

Tipos de atributos

Los tipos de atributos empleados para describir los atributos de las entidades de la base de datos son los siguientes:

- **N Identificadores Únicos**: Un atributo que es capaz de almacenar cualquier identificador (ID) dentro de un conjunto mínimo de N identificadores únicos, independientemente de la representación real del atributo: binario, decimal, letras, etc.
- **Texto de tamaño variable, N**: El atributo debe ser capaz de almacenar cualquier cadena de caracteres de longitud variable con una longitud máxima de N caracteres. Si el atributo es almacenado como una cadena de longitud fija, y su longitud es más corta que N, se debe rellenar con espacios.
- **Texto de tamaño fijo, N**: El atributo debe ser capaz de almacenar cualquier cadena de caracteres de longitud N fija.
- **Fecha y hora**: El atributo es capaz de almacenar una fecha entre el 1 de Enero de 1900 y el 31 de Diciembre de 2100 con una resolución de al menos un segundo.
- **Numero, N dígitos**: El atributo debe ser capaz de almacenar cualquier valor de N dígitos decimales. Para el caso de valores monetarios, tiene que asegurarse la precisión en los decimales de la unidad más pequeña de dinero: en el caso de euros, la precisión mínima está en 1 céntimo.
- **Nulo**: significa fuera del rango de los valores aceptados para un atributo conocido además de ser siempre el mismo valor para ese atributo.

En las tablas, los atributos tendrán una de estas definiciones, para definir que son capaces de contener; esto no implica ninguna dependencia con la implementación física a utilizar más tarde cuando se implemente el sistema.

Entidades

Dado el diagrama relacional anterior, y listados los tipos de atributos de los que disponemos, vamos a listar todas las entidades con sus atributos y sus campos clave.

Almacén (WAREHOUSE)

Ver cuadro 3.1.

Zona (DISTRICT)

Ver cuadro 3.2.

Cliente (CUSTOMER)

Ver cuadro 3.3.

Nombre del campo	Tipo de campo	Comentarios
W_ID	2*W identificadores únicos	Siendo W el número de almacenes
W_NAME	Texto de tamaño variable, 10	
W_STREET_1	Texto de tamaño variable, 20	
W_STREET_2	Texto de tamaño variable, 20	
W_CITY	Texto de tamaño variable, 20	
W_STATE	Texto de tamaño fijo, 2	
W_ZIP	Texto de tamaño fijo 9	
W_TAX	Número, 4 dígitos	Impuestos de ventas
W_YTD	Número, 12 dígitos	Balance total anual
Clave primaria: W_ID		

Cuadro 3.1: Descripción de la tabla Almacén

Nombre del campo	Tipo de campo	Comentarios
D_ID	20 identificadores únicos	10 poblados por almacén
D_W_ID	2*W identificadores únicos	
D_NAME	Texto de tamaño variable, 10	
D_STREET_1	Texto de tamaño variable, 20	
D_STREET_2	Texto de tamaño variable, 20	
D_CITY	Texto de tamaño variable, 20	
D_STATE	Texto de tamaño fijo, 2	
D_ZIP	Texto de tamaño fijo 9	
D_TAX	Número, 4 dígitos	Impuestos de ventas
D_YTD	Número, 12 dígitos	Balance total anual
D_NEXT_O_ID	10.000.000 identificadores únicos	Siguiente número de orden disponible
Clave primaria: D_ID, D_W_ID (D_W_ID) clave foránea de Almacén (W_ID)		

Cuadro 3.2: Descripción de la tabla Zona

Nombre del campo	Tipo de campo	Comentarios
C_ID	96.000 identificadores únicos	3.000 poblados por zona
C_D_ID	20 identificadores únicos	
C_W_ID	2*W identificadores únicos	
C_FIRST	Texto de tamaño variable, 16	
C_MIDDLE	Texto de tamaño fijo, 2	
C_LAST	Texto de tamaño variable, 16	
C_STREET_1	Texto de tamaño variable, 20	
C_STREET_2	Texto de tamaño variable, 20	
C_CITY	Texto de tamaño variable, 20	
C_STATE	Texto de tamaño fijo, 2	
C_ZIP	Texto de tamaño fijo, 9	
C_PHONE	Texto de tamaño fijo 16	
C_SINCE	Fecha y hora	
C_CREDIT	Texto de tamaño fijo 2	“GC”=bueno, “BC”=malo
C_CREDIT_LIM	Número, 12 dígitos	
C_DISCOUNT	Número, 4 dígitos	
C_BALANCE	Número con signo, 12 dígitos	
C_YTD_PAYMENT	Número, 12 dígitos	
C_PAYMENT_CNT	Número, 4 dígitos	
C_DELIVERY_CNT	Número, 4 dígitos	
C_DATA	Texto de tamaño variable, 500	Información variada
Clave primaria: C_W_ID, C_D_ID, C_ID (C_W_ID, C_D_ID) clave foránea de Zona (D_W_ID, D_ID)		

Cuadro 3.3: Descripción de la tabla Cliente

Nombre del campo	Tipo de campo	Comentarios
H_C_ID	96.000 identificadores únicos	
H_C_D_ID	20 identificadores únicos	
H_C_W_ID	2*W identificadores únicos	
H_D_ID	20 identificadores únicos	
H_W_ID	2*W identificadores únicos	
H_DATE	Fecha y hora	
H_AMOUNT	Numero, 6 dígitos	
H_DATA	Texto de tamaño variable, 24	Información variada
Clave primaria: ninguna, en este contexto no es necesario identificar únicamente a cada tupla. (H_C_W_ID, H_C_D_ID, H_C_ID) clave foránea de Cliente (C_W_ID, C_D_ID, C_ID) (H_W_ID, H_D_ID) clave foránea de Almacén (D_W_ID, D_ID)		

Cuadro 3.4: Descripción de la tabla Histórico

Nombre del campo	Tipo de campo	Comentarios
NO_O_ID	10.000.000 identificadores únicos	
NO_D_ID	20 identificadores únicos	
NO_W_ID	2*W identificadores únicos	
Clave primaria: NO_W_ID, NO_D_ID, NO_O_ID (NO_W_ID, NO_D_ID, NO_O_ID) clave foránea de Pedido (O_W_ID, O_D_ID, O_ID)		

Cuadro 3.5: Descripción de la tabla Nuevo-Pedido

Histórico (HISTORY)

Ver cuadro 3.4.

Nuevo-Pedido (NEW-ORDER)

Ver cuadro 3.5.

Pedido (ORDER)

Ver cuadro 3.6.

Línea-Pedido (ORDER-LINE)

Ver cuadro 3.7.

Producto (ITEM)

Ver cuadro 3.8.

Existencias (STOCK)

Ver cuadro 3.9.

Nombre del campo	Tipo de campo	Comentarios
O_ID	10.000.000 identificadores únicos	
O_D_ID	20 identificadores únicos	
O_W_ID	2*W identificadores únicos	
O_C_ID	96.000 identificadores únicos	
O_ENTRY_D	Fecha y hora	
O_CARRIER_ID	10 identificadores únicos o nulo	
O_OL_CNT	Número, 2 dígitos	De 5 a 15 líneas de pedido
O_ALL_LOCAL	Número, 1 dígito	
Clave primaria: O_W_ID, O_D_ID, O_ID (O_W_ID, O_D_ID, O_C_ID) clave foránea de (C_W_ID, C_D_ID, C_ID)		

Cuadro 3.6: Descripción de la tabla Pedido

Nombre del campo	Tipo de campo	Comentarios
OL_O_ID	10.000.000 identificadores únicos	
OL_D_ID	20 identificadores únicos	
OL_W_ID	2*W identificadores únicos	
OL_NUMBER	15 identificadores únicos	
OL_I_ID	200.000 identificadores únicos	
OL_SUPPLY_W_ID	2*W identificadores únicos	
OL_DELIVERY_D	Fecha y hora o nulo	
OL_QUANTITY	Número, 2 dígitos	
OL_AMOUNT	Número, 6 dígitos	
OL_DIST_INFO	Texto de tamaño fijo, 24	
Clave primaria: OL_W_ID, OL_D_ID, OL_O_ID, OL_NUMBER (OL_W_ID, OL_D_ID, OL_O_ID) clave foránea de Pedido (O_W_ID, O_D_ID, O_ID) (OL_SUPPLY_W_ID, OL_I_ID) clave foránea de Existencias (S_W_ID, S_I_ID)		

Cuadro 3.7: Descripción de la tabla Línea-Pedido

Nombre del campo	Tipo de campo	Comentarios
I_ID	200.000 identificadores únicos	Poblado con 100.000 productos
I_IM_ID	200.000 identificadores únicos	Identificador de imagen asociada
I_NAME	Texto de tamaño variable, 24	
I_PRICE	Número, 5 dígitos	
I_DATA	Texto de tamaño variable, 50	Nombre de la marca
Clave primaria: I_ID		

Cuadro 3.8: Descripción de la tabla Producto

Nombre del campo	Tipo de campo	Comentarios
S_I_ID	200.000 identificadores únicos	100.000 poblados por almacén
S_W_ID	2*W identificadores únicos	
S_QUANTITY	Número, 4 dígitos	
S_DIST_01	Texto de tamaño variable, 24	
S_DIST_02	Texto de tamaño variable, 24	
S_DIST_03	Texto de tamaño variable, 24	
S_DIST_04	Texto de tamaño variable, 24	
S_DIST_05	Texto de tamaño variable, 24	
S_DIST_06	Texto de tamaño variable, 24	
S_DIST_07	Texto de tamaño variable, 24	
S_DIST_08	Texto de tamaño variable, 24	
S_DIST_09	Texto de tamaño variable, 24	
S_DIST_10	Texto de tamaño variable, 24	
S_YTD	Número, 8 dígitos	
S_ORDER_CNT	Número, 4 dígitos	
S_REMOTE_CNT	Número, 4 dígitos	
S_DATA	Texto de tamaño variable, 50	Información variada
Clave primaria: S_W_ID, S_I_ID (S_W_ID) clave foránea de Almacén (W_ID) (S_I_ID) clave foránea de Producto (I_ID)		

Cuadro 3.9: Descripción de la tabla Existencias

3.2. Transacciones

3.2.1. Introducción

Vamos a utilizar ciertos términos y funciones estadísticas que conviene definir antes de explicar detalladamente qué trabajos se van a realizar sobre la base de datos.

- **Transacción:** Hablamos de transacción como la unidad de trabajo de nuestro sistema, aunque cada transacción se divide en múltiples operaciones en la base de datos.
- **Rango:** Especificamos un rango de valores entra a y b como [a..b]
- **Elemento aleatorio uniforme:** entre un rango de valores [a..b], como un elemento seleccionado de manera independiente y uniforme entre a y b, ambos incluidos.
- **Elemento aleatorio no-uniforme:** también representado por la función NURAND(X,a,b), un número aleatorio entre un rango de valores [a..b], seleccionado de manera no uniforme. La fórmula utilizada para su cálculo es: `NURAND(A, x, y)=( ((random(0, A) | random(x, y) ) + C) modulo (y - x + 1) ) + x` donde:
  - **a | b** Es la operación lógica OR entre los bits de a y b.
  - **random([a..b])** Es un número aleatorio entre a y b.
  - **La constante A** Depende del rango [a..b], cuyo valor será:
    - Para el rango [0..999], usando en C_LAST: A=255

    - Para el rango [0..3000], usando en C_ID: A=1023
    - Para el rango [0..100000], usando en OL_I_ID: A=8191
  - **La constante C** Se encuentra dentro del rango [0..A], se elige aleatoriamente en tiempo de ejecución, y puede ser variada sin alterar el rendimiento, aunque debe usarse la misma constante C durante toda la ejecución de la prueba. Las normas para tener una buena constante C, son las siguientes:
    - Llamemos C-Load a la constante C usada para generar C_LAST cuando se pobló la base de datos.
    - Llamemos C-Run la constante C usada para generar C_LAST en ejecución.
    - Obtenemos el valor C-Delta como el valor absoluto de la diferencia entre C-Load y C-Run.
    - El valor de C-Delta debe estar en el rango de [65..119] excepto los valores 96 y 112.

- **Aplicación:** Es el software que, no formando parte del incluido en el sistema de prueba, sirve para implementar los perfiles de transacción que se especifican en el estándar TPC-C; desde procedimientos, estructuras hasta restricciones de integridad referencial (según el sistema de almacenamiento que se use).

- **Terminal:** Se refiere al dispositivo capaz de mostrar caracteres y de recoger las entradas del usuario; dicho dispositivo no debe tener ningún conocimiento de la aplicación excepto el formato de los campos y su posición en la terminal.

3.2.2. Nuevo pedido

La transacción de *Nuevo Pedido* consiste en introducir un pedido completo en el sistema. Es una operación con una carga media de lectura y escritura; y dado que es una de las más frecuentes, es el núcleo de la carga de trabajo, e intenta simular un comportamiento de carga variable en un sistema real.

Datos de entrada

- El número de almacén local (W_ID) permanece constante durante la transacción.

- El número de la zona (D_ID) donde realizar el pedido es aleatorio entre [1..10]

- El número de líneas de pedido (O_OL_CNT) es aleatorio entre [5..15], con una media de 10 líneas.

- Para cada línea de pedido:

  1. Se selecciona un número de producto entre 1 y 10000 para OL_I_ID con una distribución NURAND(8191,1,100000).
  2. Para el almacén que provee ese producto (OL_SUPPLY_W_ID), el 99 % de las veces es el almacén local, pero un 1 % es otro almacén distinto. Esto se puede solucionar fácilmente con un número aleatorio entre [1..100], de tal manera que si ese número es 1, se escoja un almacén local (OL_SUPPLY_W_ID=W_ID) y en el resto de casos el almacén remoto (OL_SUPPLY_W_ID es aleatorio entre los almacenes restantes).

3. Una cantidad de producto (OL_QUANTITY) es seleccionada aleatoriamente entre [1 ..10].

- Una fecha para el pedido (O_ENTRY_D), usando la fecha actual del sistema.

Hay que recordar que puede haber pedidos **locales**: aquellos en los que el almacén desde donde se realiza el pedido (O_W_ID) es igual al almacén que provee los productos (OL_SUPPLY_W_ID); y **remotos**: en los que no coinciden.

Perfil de la transacción

Resumen de acciones lectura/escritura a realizar:

- **Lectura**
  - Almacén (WAREHOUSE), mediante W_ID (clave)
  - Zona (DISCTRICT), mediante D_W_ID, D_ID (clave)
  - Cliente (CUSTOMER), mediante C_W_ID, C_D_ID, C_ID (clave)
  - Producto (ITEM), mediante I_ID (clave)
  - Existencias (STOCK), mediante S_I_ID, S_W_ID (clave)
- **Escritura**: inserción y actualización
  - Nueva inserción en nuevo pedido (NEW_ORDER)
  - Nueva inserción en pedido (ORDER)
  - Actualización de zona (DISTRICT)
  - Actualización de cliente (CUSTOMER)
  - Actualización de existencias (STOCK)
  - Nueva inserción en línea de pedido (ORDER_LINE)

Para un número de almacén (W_ID), número de zona (D_W_ID, D_ID), número de cliente (C_W_ID, C_D_ID, C_ID), número de líneas de pedido *num_lp*, y sus correspondientes productos (OL_I_ID) provenientes de un almacén (OL_SUPPLY_W_ID) y en una determinada cantidad (OL_QUANTITY):

1. Buscamos el almacén W_ID (en WAREHOUSE) y obtenemos los impuestos de ese almacén: W_TAX.
2. Buscamos la zona D_W_ID, D_ID (en DISTRICT), obtenemos sus impuestos D_TAX, obtenemos el número del siguiente pedido O_NEXT_O_ID, y lo incrementamos en uno.
3. Buscamos el cliente C_W_ID, C_D_ID, C_ID (en CUSTOMER) y obtenemos: su descuento C_DISCOUNT, su apellido C_LAST y su crédito C_CREDIT.
4. Insertamos una nueva fila en NEW_ORDER y ORDER donde: O_CARRIER_ID es puesto a nulo, si todas las líneas de pedido son locales O_ALL_LOCAL es puesto a 1 y el número de líneas de pedido O_OL_CNT es puesto a *num_lp*.
5. Para cada línea de pedido O_OL_CNT:

a) Buscamos el producto I_ID (en ITEM) y obtenemos: su precio I_PRICE, su nombre I_ITEM y sus datos I_DATA.

b) Buscamos la línea en existencias (STOCK) del producto S_I_ID,S_W_ID, donde S_W_ID = W_ID, y obtenemos: la cantidad en almacén S_QUANTITY, e información variada S_DATA, S_DIST_xx (donde xx es el número de zona: D_ID).
Si el valor de lo que tenemos S_QUANTITY supera el de lo que pedimos OL_QUANTITY en 10 o más unidades, S_QUANTITY = S_QUANTITY - OL_QUANTITY; si no, S_QUANTITY = (S_QUANTITY - OL_QUANTITY) + 91 (stock infinito).
La cantidad total de producto servido S_YTD se incrementa con la cantidad de producto pedido OL_QUANTITY, y en el caso de que el pedido sea local, el número de pedidos sobre las existencias S_ORDER_CNT aumenta en uno; si no es local se aumenta S_REMOTE_CNT en uno.

c) El precio total de la línea de pedido OL_AMOUNT es OL_QUANTITY * I_PRICE.

d) Se inserta una nueva tupla en ORDER_LINE para reflejar la nueva línea del pedido donde: OL_DELIVERY_D es puesto a nulo, OL_NUMBER es único entre el resto de líneas de pedido que tienen el mismo identificador de orden OL_O_ID, el número de zona OL_D_ID se establece a D_ID,y la información de zona OL_DIST_INFO es el valor de S_DIST_xx (donde xx es OL_D_ID).

3.2.3. Pago

La transacción de pago actualiza el balance económico del cliente y actualiza las estadísticas de pagos en la zona y el almacén. Es una transacción ligera, de lectura/escritura y con bastante frecuencia de ejecución. Además incluye un acceso a la tabla de clientes (CUSTOMER) sin usar la clave primaria.

Datos de entrada

- El número de almacén (W_ID) permanece constante durante la transacción.
- El número de zona (D_ID) se selecciona aleatoriamente entre [1..10] (teniendo en cuenta que D_W_ID = W_ID).
- El cliente es seleccionado aleatoriamente con un número [1..100] de dos maneras:
  - Si <=60 (el 60 %): mediante su apellido (C_W_ID, C_D_ID, C_LAST) usando NURand (255, 0, 999) para C_LAST.
  - Si es >60 (el 40 % restante): por su número (C_W_ID, C_D_ID, C_ID), y se usa una probabilidad NURand (1023, 1, 3000) para C_ID.
- Independientemente de como haya sido seleccionado un cliente, el pago puede ser local o remoto, mediante otro número aleatorio entre [1..100]:
  - Si es <=85 (el 85 %): el cliente paga localmente, por lo que C_D_ID = D_ID y C_W_ID = W_ID.
  - Si es >85 (el 15 % restante): el cliente paga a traves de una zona y un almacén que no es el suyo, por lo que se selecciona una zona (C_D_ID) aleatoriamente entre [1..10], y un almacén aleatoriamente entre los disponibles.

- La cantidad a pagar (H_AMOUNT) es seleccionada entre [1..5000]
- Para fecha del pago (H_DATE) se usa la hora actual.

Aquí también recordamos que un pago puede ser *local*, si el cliente pertenece al almacén desde el cual se hace el pago; y es *remoto*, si el almacén desde el que se hace el pago no es el almacén al que pertenece el cliente (C_W_ID es distinto de W_ID).

Perfil de la transacción

Resumen de acciones lectura/escritura a realizar:

- **Lectura**
  - Almacén (WAREHOUSE), mediante W_ID (clave).
  - Zona (DISTRICT), mediante D_W_ID, D_ID (clave).
  - Cliente (CUSTOMER), mediante C_W_ID, C_D_ID, C_ID (clave).
  - Cliente (CUSTOMER), mediante mediante C_W_ID, C_D_ID, C_LAST (**NO** clave).
- **Escritura**: inserción y actualización
  - Nueva línea del histórico (HISTORY).
  - Actualización de almacén (WAREHOUSE).
  - Actualización de la zona (DISTRICT).

Para un almacén (W_ID), una zona del almacén (D_W_ID, D_ID), un cliente (C_W_ID, C_D_ID, C_ID o C_W_ID, C_D_ID, C_LAST) y una cantidad a pagar (H_AMOUNT).

1. Obtenemos los datos del almacén W_ID y actualizamos el balance económico del almacén: W_YTD = W_ITD + H_AMOUNT.
2. Obtenemos los datos de la zona D_W_ID, D_ID y actualizamos el balance económico de la zona: D_YTD = D_YTD + H_AMOUNT.
3. Si el cliente es seleccionado mediante su apellido C_LAST, puede ocurrir que tengamos más de un cliente, por lo que ordenaremos por el campo C_FIRST y cogeremos el cliente de la mitad de la lista.
4. Para el cliente indicado:
   a) Disminuimos el balance actual: C_BALANCE = C_BALANCE - H_AMOUNT.
   b) Aumentamos el balance anual: C_YTD_PAYMENT = C_YTD_PAYMENT + H_AMOUNT.
   c) El número de pagos C_PAYMENT_CNT es incrementado en uno.
5. Si el valor de C_CREDIT es “BC”, hay que actualizar el campo C_DATA de la siguiente manera:
   a) Obtenemos los siguientes datos: C_ID, C_D_ID, C_W_ID, D_ID, W_ID y H_AMOUNT.
   b) Los insertamos por la izquierda de C_DATA, desplazando a la derecha lo que hubiera.

c) C_DATA nunca debe sobrepasar los 500 caracteres.

6. El campo H_DATA se construye concatenando W_NAME, 4 espacios y D_NAME.

7. Insertamos una nueva tupla en el historial (HISTORY) de esta manera: H_C_ID = C_ID, H_C_D_ID = C_D_ID, H_C_W_ID = C_W_ID, H_D_ID = D_ID y H_W_ID = W_ID.

3.2.4. Estado de un pedido

La transacción de consulta de estado de un pedido, informa sobre el estado del último pedido del cliente. Es una transacción de solo lectura e incluye acceso a la tabla de clientes (CUSTOMER) sin usar un campo clave.

Datos de entrada

- El número de almacén (W_ID) permanece constante durante la transacción.
- La zona es seleccionada aleatoriamente entre [1..10] del almacén dado.
- El cliente es seleccionado, generando un número aleatorio entre [1..100]:
  - Si es <=60 (el 60 %): usando su apellido, y para ello utilizaremos una distribución NURand (255, 0, 999).
  - Si es >60 (el 40 % restante): usando su número de cliente mediante la distribución NURand (1023, 0 ,3000).

Perfil de la transacción

Resumen de acciones de lectura a realizar:

- **Lectura**
  - Cliente (CUSTOMER), mediante C_W_ID, C_D_ID, C_ID (clave).
  - Cliente (CUSTOMER), mediante mediante C_W_ID, C_D_ID, C_LAST (**NO** clave).
  - Pedido (ORDER), mediante O_W_ID, O_D_ID, O_C_ID, O_ID (clave)
  - Líneas de pedido(ORDER_LINE), mediante OL_W_ID, OL_D_ID, OL_O_ID (clave).

Para un almacén (W_ID), una zona (C_W_ID, D_ID) y un cliente (C_W_ID, C_D_ID, C_ID o C_W_ID, C_D_ID, C_LAST) dados.

1. Si el cliente es seleccionado mediante su apellido C_LAST, puede ocurrir que tengamos más de un cliente, por lo que ordenaremos por el campo C_FIRST y cogeremos el cliente de la mitad de la lista.

2. Obtenemos el pedido del cliente con el mayor O_ID

3. Obtenemos todas las líneas del pedido.

3.2.5. Envío

La transacción de envío consiste en procesar 10 pedidos nuevos. Es una transacción que no requiere mucha lectura y escritura aunque en los sistemas reales tiene poca frecuencia y no importa el tiempo que tarde.

Datos de entrada

- El número de almacén (W_ID) permanece constante durante la transacción.
- El número de la empresa de transportes (O_CARRIER_ID) se selecciona aleatoriamente entre [1..10]
- La fecha de envío (OL_DELIVERY_D) es la fecha actual del sistema.

Perfil de la transacción

Para el envío de un pedido:

- **Lectura**
  - Nuevo pedido (NEW_ORDER), mediante NO_W_ID, NO_D_ID, NO_ID (clave).
  - Pedido (ORDER), mediante O_W_ID, O_D_ID, O_C_ID, O_ID (clave).
  - Líneas de pedido (ORDER_LINE), mediante OL_W_ID, OL_D_ID, OL_O_ID (clave).
  - Cliente (CUSTOMER), mediante C_W_ID, C_D_ID, C_ID (clave).
- **Escritura**: inserción y actualización
  - Eliminar una tupla de nuevos pedidos (NEW_ORDER).
  - Actualizar pedido (ORDER).
  - Actualizar cliente (CUSTOMER).

Para un almacén (W_ID), para cada una de las diez zonas (D_W_ID, D_ID) de ese almacén y para un número de empresa de transporte (O_CARRIER_ID). Es decir, los 10 envíos son dado un almacén (W_ID), un envío por cada zona.

1. Seleccionamos, dentro de nuevo pedido (NEW_ORDER), dados NO_W_ID = W_ID y NO_D_ID = D_ID, se selecciona el pedido con el NO_O_ID más bajo (el más viejo), y lo borramos.
2. Seleccionamos el pedido correspondiente al nuevo pedido borrado: O_W_ID = W_ID, O_D_ID = D_ID, y O_ID = NO_O_ID; y actualizamos la empresa de transporte utilizada a O_CARRIER_ID.
3. Para todas las líneas de pedido que contiene el pedido, se modifica la fecha de envío OL_DELIVERY_D con la fecha actual del sistema y acumulamos la suma de los costes de cada línea OL_AMOUNT.
4. Para el cliente dueño del pedido: C_W_ID = W_ID, C_D_ID = D_ID y C_ID = O_C_ID; su balance local C_BALLANCE es incrementado por la suma de las cantidades anteriores, y la cantidad de pedidos enviados C_DELIVERY_CNT se incrementa en 1.

3.2.6. Nivel de existencias

La transacción de nivel de existencias averigua el número de productos que tienen un nivel de existencias por debajo de un límite. Es una transacción de sólo lectura pero con muchas lecturas, aunque es de poca frecuencia.

Datos de entrada

- Se necesita un número de almacén y una zona (W_ID, D_ID), que permanecerán constantes durante la transacción.
- Un límite mínimo de existencias, elegido aleatoriamente entre [10..20].

Perfil de la transacción

Se comprueban los niveles de existencia de los productos utilizados en las ultimas 20 transacciones.

- **Lectura**
  - Zona (DISTRICT), mediante D_W_ID, D_ID (clave).
  - Liínea de pedido (ORDER_LINE), mediante OL_W_ID, OL_D_ID, OL_O_ID (clave).
  - Existencias (STOCK), mediante S_I_ID, S_W_ID (clave).

Dado un número de almacén (W_ID), un número de zona (D_W_ID, D_ID) y un nivel mínimo de existencias *min*.

1. Seleccionamos la zona D_W_ID, D_ID, y obtenemos el próximo número de pedido D_NEXT_O_ID.
2. Obtenemos todas las líneas de pedido (ORDER_LINE) que cumplan: OL_W_ID = W_ID, OL_D_ID = D_ID y OL_O_ID más bajo que D_NEXT_O_ID o mayor que D_NEXT_O_ID + 20 (como se prefiera).
3. De cada línea de pedido, extraemos el producto y buscamos en stock la línea del producto: S_I_ID = OL_I_ID, S_W_ID = W_ID. En esa línea comprobamos que las existencias S_QUANTITY sean mayores que *min*, y en caso afirmativo, informamos de que se ha sobrepasado el límite.

3.2.7. Resumen y reglas de implementación

Resumen de accesos

Ver cuadro 3.10 .

Reglas de implementación

A la hora de implementar estas tablas en un sistema de almacenamiento de datos, si se quiere cumplir estrictamente la especificación del TPC-C, hay una serie de reglas a tener en cuenta.

- El uso de vistas para evitar lecturas/escrituras no está permitido.

Tabla	Lectura Indexada	Lectura no-Indexada	Actualización	Añadir	Borrar
WAREHOUSE	X				
DISTRICT	X				
CUSTOMER	X	X	X		
HISTORY				X	
ORDER	X		X	X	
NEW_ORDER	X			X	X
ORDER_LINE	X			X	
STOCK	X		X		
ITEM	X				

Cuadro 3.10: Resumen de accesos a las tablas

- Se permite distribuir los registros en un cluster dentro de la base de datos. Esto es necesario para que la aplicación realice la cantidad indicada de operaciones de lectura/escritura en cada perfil de transacción, sin agrupar operaciones a través de vistas.
- Todas las tablas deben estar pobladas correctamente.
- Se permite dividir las tablas tanto horizontal como verticalmente, aunque en el caso de realizarse se deben publicar estos detalles.
- Se permite realizar réplicas de las tablas, aunque esas réplicas deben cumplir con los mismos requisitos que las tablas originales.
- Se permite duplicar y añadir atributos extra, siempre y cuando este hecho no esté destinado a mejorar el rendimiento.
- Cada atributo descrito anteriormente debe ser discreto y accesible independientemente. Sin embargo, para los atributos de tipo *Fecha y hora* y para aquellos que sean datos (sufijo _DATA), es posible separarlos en varias partes si se considera más útil.
- La clave primaria no debe representar directamente direcciones físicas de acceso o desplazamientos físicos.
- Ya que no se inserta o se borra en todas las tablas, el sistema no debe estar configurado para aprovecharse especialmente de esto durante las pruebas. Aunque a la hora de insertar nuevos datos el límite lo establece la cantidad de espacio disponible, hay que asegurar que la cardinalidad de la tabla pueda aumentarse en un 5 % y las claves puedan almacenar un rango el doble del especificado.
- Reglas de integridad: las claves deben ser siempre únicas, y siempre se debe de poder determinar el valor de cualquier atributo.
- Se requiere un acceso transparente a los datos, de tal manera que la aplicación que implementa las especificaciones TCP-C no necesite tener conocimiento de dónde están almacenados los datos.

Nombre Tabla	Cardinalidad	Tamaño por tupla (Bytes)	Tamaño por tabla (KiBytes)
Almacén (WAREHOUSE)	1	89	0,089
Zona (DISTRICT)	10	95	0,950
Cliente (CUSTOMER)	30K	655	19650
Histórico (HISTORY)	30K	46	1380
Pedido (ORDER)	30K	24	720
Nuevo-Pedido (NEW_ORDER)	9K	8	72
Linea-Pedido (ORDER_LINE)	300K	54	16200
Existencias (STOCK)	100K	306	30600
Productos (ITEM)	100K	82	8200

Cuadro 3.11: Cardinalidades de las tablas

- Cada una de las 9 tablas debe ser identificable mediante su nombre, y el acceso a su contenido se hará a través de ese nombre, con el fin de usar mecanismos de acceso lo más generales posibles.

3.3. Poblado inicial

Cada almacén requiere un buen número de tuplas para poblar inicialmente la base de datos, lo que aquí se describe es qué cantidades relativas hacen falta.

3.3.1. Escalabilidad

El número de almacenes es la unidad base de escalado del sistema, la cardinalidad del resto de tablas depende de la cantidad de almacenes; excepto la de la tabla productos, que es constante e independiente del número de almacenes, ya que todos los almacenes comparten la misma lista de productos.

Las cardinalidades para un poblado inicial por cada almacén son las expresadas en el cuadro 3.11.

Según las especificaciones de las transacciones a realizar, sabemos que la cardinalidad de las tablas: Histórico, Nuevo-Pedido, Pedido y Línea-Pedido, variará como resultado de la repetición de las diferentes pruebas. Aunque el poblado inicial de la base de datos y las transacciones están pensadas para minimizar el impacto de esta variación de cardinalidad, disminuyendo el impacto en el rendimiento y haciendo que las pruebas sean repetibles.

3.3.2. Poblado

Cada tabla debe contener el número de filas definidas en la sección anterior antes de iniciar las pruebas; por ejemplo, la tabla Nuevo-Pedido, debe contener 2000 filas por cada almacén. Si bien el poblado inicial se indica en unas tablas hay un parámetro que se puede variar, y es el número de almacenes.

A la hora de generar carga de manera más variada, el número de almacenes sirve como unidad de medida de esa carga que se va a generar, ya que si cada almacén tiene: distritos, y esos distritos clientes con sus pedidos, también tiene existencias. El hecho de añadir un almacén multiplica la cardinalidad del resto de tablas con las que se relaciona.

0	1	2	3	4	5	6	7	8	9
BAR	OUG	ABLE	PRI	PRES	ESE	ANTU	CALLY	ATION	EING

Cuadro 3.12: Equivalencias número-texto

Definición de términos

- El término **aleatorio** significa: seleccionado independientemente y uniformemente distribuido sobre el rango especificado. Para poblar la base de datos, se permite la generación de una gran cantidad de números aleatorios para luego usarlos más tarde de manera secuencial.
- La notación **a-string aleatoria[x .. y]** (y respectivamente la notación **n-string aleatoria[x .. y]**) indica una cadena de caracteres alfanuméricos (numéricos para el caso de n-string), de mínimo x, máximo y, y media (y+x)/2. El juego de caracteres empleado para esto debe poder representar al menos 128 caracteres diferentes.

  Para generar estas cadenas, se puede implementar una concatenación de otras dos cadenas seleccionadas aleatoriamente de dos vectores de cadenas, donde:

  1. Ambos vectores contengan un mínimo de 10 cadenas diferentes.
  2. El primer vector contenga cadenas de x caracteres.
  3. El segundo vector contiene cadenas de longitud uniformemente distribuida entre cero e (y - x) caracteres.
  4. Ambos vectores pueden contener cadenas específicas para cada atributo en vez de cadenas genéricas y totalmente aleatorias, con tal de que esto no suponga ninguna mejora de rendimiento a la hora de ejecutar las pruebas.
- El apellido del cliente (C_LAST) debe ser generado por la concatenación de tres sílabas de longitud variable seleccionadas del cuadro 3.12.

  Por lo que dado un número entre 0 y 999, cada una de las 3 sílabas es obtenida por el dígito correspondiente en la representación con 3 dígitos del número. Por ejemplo, el número 836 produce el nombre ATIONPRIANTU; y el número 12 produce el nombre BAROUGABLE.
- La notación **único entre[**x**]** identifica cualquier valor dentro de un grupo de x valores contiguos y único en el grupo de filas que está siendo poblado.
- La notación **aleatorio entre [**x **..** y**]** identifica a un valor aleatorio seleccionado independientemente y distribuido de manera uniforme entre x e y (ambos incluidos), con una media de $(x+y)/2$ y con el mismo número de dígitos de precisión. Ejemplo: $[0.01..100.00]$ tiene 10000 valores únicos, mientras que $[1..100]$ tiene solo 100
- La notación **permutación aleatoria de [**x **..** y**]** identifica una secuencia de números de x a y dispuestos de manera aleatoria.
- Los códigos postales de almacén, cliente y zona (W_ZIP, C_ZIP y D_ZIP respectivamente) deben ser generados de la siguiente manera.

  1. Una primera cadena de 4 caracteres del tipo: *random n-string[0000 .. 9999].*
  2. La cadena constante 1111.

De tal manera que dado el número 345, obtenemos el siguiente código postal: 03451111. Y dado que hay 30.000 clientes por almacén y 10.000 códigos postales distintos tenemos una media de 3 clientes por almacén con el mismo código postal.

Indicaciones para el poblado de las tablas

La población inicial de la base de datos debe estar compuesta por:

- En la tabla productos (ITEM) se insertan 100.000 filas con las siguientes características:

  I_ID único entra [100.000]

  I_IM_ID aleatorio entre [1 .. 10.000]

  I_NAME a-string aleatoria entre [14 .. 24]

  I_PRICE aleatorio entre [1,00 .. 100,00]

  I_DATA a-string aleatorio [26 .. 50]. Para el 10 % de las filas se selecciona aleatoriamente, la cadena *ORIGINAL* debe incluirse en I_DATA en una posición aleatoria.

- Una entrada en la tabla de almacenes para cada almacén configurado con estas características:

  W_ID único entre el número de almacenes configurados.

  W_NAME a-string aleatoria entre [6 .. 10]

  W_STREET_1 a-string aleatoria entre [10 .. 20]

  W_STREET_2 a-string aleatoria entre [10 .. 20]

  W_CITY a-string aleatoria entre [10 .. 20]

  W_STATE a-string aleatoria de 2 caracteres.

  W_ZIP generado como se indicó anteriormente.

  W_TAX aleatorio entre[0,0000 .. 0,2000]

  W_YTD = 300.000,00

  Para cada tupla de la tabla de almacenes tenemos que añadir:

  - 100.000 tuplas a la tabla de existencias, ya que existencias contiene la cantidad de un producto para un almacén determinado.

    S_I_ID único entre [100.000]
    S_W_ID = W_ID
    S_QUANTITY aleatorio entre [10 .. 100]
    S_DIST_01 a-string aleatorio de 24 caracteres
    S_DIST_02 a-string aleatorio de 24 caracteres
    S_DIST_03 a-string aleatorio de 24 caracteres
    S_DIST_04 a-string aleatorio de 24 caracteres
    S_DIST_05 a-string aleatorio de 24 caracteres
    S_DIST_06 a-string aleatorio de 24 caracteres
    S_DIST_07 a-string aleatorio de 24 caracteres

S_DIST_08 a-string aleatorio de 24 caracteres

S_DIST_09 a-string aleatorio de 24 caracteres

S_DIST_10 a-string aleatorio de 24 caracteres

S_YTD = 0

S_ORDER_CNT = 0

S_REMOTE_CNT = 0

S_DATA a-string aleatoria entre [26 .. 50]. Para el 10 % de las filas tiene que aparecer la cadena "ORIGINAL" en una posición aleatoria dentro de S_DATA

- 10 tuplas en la tabla de zona con las siguientes características:

  D_ID único entre [10]

  D_W_ID = W_ID

  D_NAME a-string aleatoria entre[6 .. 10]

  D_STREET_1 a-string aleatoria entre [10 .. 20]

  D_STREET_2 a-string aleatoria entre [10 .. 20]

  D_CITY a-string aleatoria entre [10 .. 20]

  D_STATE a-string aleatoria de dos caracteres.

  D_ZIP generado como se indicó anteriormente.

  D_TAX aleatorio entre [0,0000 .. 0,2000]

  D_YTD = 30.000,00

  D_NEXT_O_ID = 3.001

  Para cada tupla de la tabla de zona añadimos:

  - 3000 tuplas en la tabla de clientes con estas características:

    C_ID único entre [3.000]

    C_D_ID = D_ID

    C_W_ID = D_W_ID

    C_LAST generado como se indicó anteriormente y basándose en una iteración entre 0 y 999 para los primeros 1000 clientes; y generando números aleatorios no uniformes con la función NURand(255,0,999) para los 2000 clientes restantes. Recordemos que dicha función de números no aleatorios tiene una constante de funcionamiento que tiene que ser escogida de manera aleatoria entre los diferentes experimentos.

    C_MIDDLE = "OE"

    C_FIRST a-string aleatoria entre [8 .. 16]

    C_STREET_1 a-string aleatoria entre [10 .. 20]

    C_STREET_2 a-string aleatoria entre [10 .. 20]

    C_CITY a-string aleatoria entre [10 .. 20]

    C_STATE a-string aleatoria de 2 caracteres.

    C_ZIP generado como se indicó anteriormente.

    C_PHONE n-string aleatoria de 16 dígitos.

    C_SINCE fecha y hora del momento de poblado.

    C_CREDIT = "GC". Para el 10 % de las filas, para el resto C_CREDIT = "BC"

C_CREDIT_LIM = 50.000,00

C_DISCOUNT aleatorio entre [0,0000 .. 0,5000]

C_BALANCE = -10,00

C_YTD_PAYMENT = 10,00

C_PAYMENT_CNT = 1

C_DELIVERY_CNT = 0

C_DATA a-string aleatoria entre [300 .. 500]

Que a su vez, por cada cliente que se añade, generamos una entrada en la tabla de histórico con su clave y las siguientes características:

H_C_ID = C_ID

H_C_D_ID = H_D_ID = D_ID

H_C_W_ID = H_W_ID = W_ID

H_DATE fecha y hora actual

H_AMOUNT = 10,00

H_DATA a-string aleatoria entre [12 .. 24]

- Siguiendo en población generada por cada zona, añadimos 3000 tuplas a la tabla de pedido con las siguientes características:

  O_ID único entre [3.000]

  O_C_ID escogido secuencialmente de una permutación de [1 .. 3.000]

  O_D_ID = D_ID

  O_W_ID = W_ID

  O_ENTRY_D fecha y hora actual.

  O_CARRIER_ID Aleatorio entre [1 .. 10] si O_ID <2.101, si no nulo

  O_OL_CNT Aleatorio entre [5 .. 15]

  O_ALL_LOCAL = 1

  Y por cada pedido que tengamos, hay que generar las líneas de pedido según el parámetro O_OL_CNT que indica cuántas líneas hay en un pedido. Estas líneas de pedido además de cumplir las siguientes características también están cumpliendo las normas que se indicaron en la transacción de *nuevo pedido*.

  OL_O_ID = O_ID

  OL_D_ID = D_ID

  OL_W_ID = W_ID

  OL_NUMBER único entre [O_OL_CNT]

  OL_I_ID aleatorio entre [1 .. 100.000]

  OL_SUPPLY_W_ID = W_ID

  OL_DELIVERY_D Pondremos O_ENTRY_D si OL_O_ID <2.101, y nulo en otro caso

  OL_QUANTITY = 5

  OL_AMOUNT Pondremos 0.00 si OL_O_ID <2.101, aleatorio entre [0,01 .. 9.999,99]

- Por último, en la zona, añadimos 900 entradas en la tabla de nuevo-pedido que se corresponden con los 900 últimos pedidos que hemos generado para esa zona.

Transacción	Tanto por ciento de ejecuciones
Nuevo Pedido	45 %
Pago	43 %
Estado Pedido	4 %
Envío	4 %
Nivel de Existencias	4 %

Cuadro 3.13: Distribución de las proporciones de las transacciones

NO_O_ID = O_ID
NO_D_ID = D_ID
NO_W_ID = W_ID

La implementación no debiera aprovecharse del hecho de que algunos campos están poblados inicialmente con un valor fijo; por ejemplo, el espacio para C_CREDIT_LIM no se pude reservar sólo una vez y enlazar desde el resto de registros a ese valor.

3.4. Métrica de rendimiento

3.4.1. Usuarios simulados

El sistema que simulamos es activado por usuarios que son también simulados. Cada usuario realiza los siguientes pasos en orden:

1. Selecciona la transacción a realizar.
2. Pasa un tiempo entre que el sistema le muestra la pantalla donde introducir los datos
3. Introduce los datos en esa pantalla, lo que conlleva un tiempo.
4. Manda realizar la transacción, y pasa un tiempo mientras se completa.
5. Se muestran los datos en su terminal, y se emplea un tiempo en leerlos.
6. Vuelta a empezar.

Importante: Para esta implementación se van a ignorar todos los requerimientos de tiempos de espera y tiempos de respuesta.

Los 5 tipos de transacciones están disponibles en todos los terminales, y aunque la selección de una transacción es aleatoria se debe mantener una proporción entre el total de transacciones enviadas (cuadro 3.13).

El propósito de estos porcentajes para cada tipo de transacción es tener, aproximadamente, una transacción de pago por cada transacción de nuevo pedido; y una transacción de envío, estado de pedido y nivel de existencias por cada 10 transacciones de nuevo pedido. Con estos porcentajes se puede asegurar que los pedidos se procesan completamente.

3.4.2. Reglas para el balanceo de transacciones

Las transacciones deben seleccionarse de manera uniformemente aleatoria a la vez que se mantienen los porcentajes de mezcla para cada tipo de transacción. Esto se puede realizar de dos maneras: por pesos y con una baraja.

Todas la terminales deben seleccionar las transacciones usando la misma técnica, y no se permite trucar los resultados usando diferentes técnicas en cada terminal.

Pesos

Un peso se asocia a cada tipo de transacción, y los porcentajes se alcanzan seleccionando cada nueva transacción de manera uniformemente aleatoria a partir de una distribución con pesos. Se deben cumplir las siguientes reglas:

- Los pesos iniciales se eligen por la persona que realiza las pruebas de tal manera que cumplan los porcentajes.

- Para alcanzar esos porcentajes, el simulador de usuario puede ajustar dinámicamente dichos pesos asociados a cada tipo de transacción durante la ejecución del benchmark. Aunque estos ajustes tienen que estar dentro de un margen del 5 % de los valores originales.

Baraja

Una o más cartas de una baraja se asocian a cada tipo de transacción, de tal manera que los porcentajes requeridos se alcanzan seleccionando una nueva transacción aleatoriamente de la baraja cuyo contenido garantiza los porcentajes. Se deben de cumplir las siguientes reglas:

- Cualquier número de terminales pueden compartir la misma baraja, incluso cada terminal puede de tener su propia baraja.

- Una baraja debe estar compuesta de uno o más conjuntos de 23 cartas:

  - 10 cartas para nuevo pedido.
  - 10 cartas para pago.
  - 1 carta para estado pedido.
  - 1 carta para envío.
  - 1 carta para nivel de existencias.

  El tamaño mínimo de una barajaa es un conjunto anteriormente descrito, y si se usa más de una baraja, todas las barajas deben ser del mismo tamaño.

- Cada paso a través de la baraja debe ser hecho de manera distinta y aleatoria. Si una baraja es accedida de manera secuencial, hay que barajarla cada vez que la terminamos; en cambio si elegimos cartas de la baraja de manera aleatoria, no las podemos devolver a la baraja hasta que se hayan acabado.

3.4.3. Cálculo de la unidad de medida

Los porcentajes que se han visto para elegir el tipo de transacción representan un ciclo completo en el negocio simulado, consistente en muchas transacciones que introducen nuevos pedidos, preguntan por el estado de los pedidos actuales, envían pedidos pendientes, anotan pagos y monitorizan el estado de las existencias en los almacenes.

La métrica utilizada para el informe final se llama *Maximum Qualified Throughput* (**MQTh**). Esta medida es más una medida de capacidad de proceso del negocio más que el ratio de ejecución de transacciones, ya que tiene en cuenta todas las transacciones con los porcentajes anteriormente dichos.

El MQTh es el número total de transacciones de nuevo pedido completadas dividido por el tiempo transcurrido en la ejecución del benchmark. El nombre de esta unidad a la hora de informar es **tpmC**, y no debe ser interpolada o extrapolada de ninguna manera, también deben ser truncados sus decimales.

Para que esta unidad sea válida, no se deben haber ignorado más de un 1 % de transacciones de envío debido a que no había pedidos para enviar en la tabla de nuevo pedido.

3.4.4. Intervalo de medida

Las pruebas deben ser hechas de tal manera que se pueda obtener la capacidad real del sistema a la hora de procesar trabajo. Aunque el intervalo de tiempo pueda ser de 2 horas, el sistema debe estar configurado para que al menos el benchmark pueda ejecutarse durante 8 horas manteniendo todas las características que se han descrito a lo largo del análisis.

Por ejemplo, no sería valida una configuración en la cual el rendimiento es mayor en los primeros momentos de la medida, y en el resto es peor debido a que se utiliza un sistema de almacenamiento dedicado al principio, pero si quisiéramos ejecutar la prueba durante 8 horas se usaría un medio compartido y el rendimiento sería inferior.

También se puede incluir un tiempo de calentamiento, en el cual el sistema se va estabilizando hasta un estado en el cual el rendimiento no sufra casi variaciones. Este tiempo de calentamiento, si se usa, no se debe tener en cuenta.

Para que la medida obtenida sea tomada como válida, durante el intervalo, debe ocurrir:

1. La media de líneas de pedido en cada pedido debe estar entre 9,5 y 10,5.
2. El número de líneas de pedido remotas debe ser entre un 0,95 % y un 1,05 %, en el caso de que exista más de un almacén.
3. El número de pagos remotos debe ser de al menos un 14 % y como mucho de un 16 %.
4. El número de búsquedas de clientes a través de su apellido en la transacción de pago y en la transacción de estado de pedido debe ser de al menos un 57 % y como mucho de un 63 %, medidos de manera independiente por cada tipo de transacción.

Duración

El intervalo de tiempo durante el cual se realizan mediciones debe ser:

1. Después del calentamiento.
2. Durante un mínimo de 120 minutos (2 horas)

3. Y lo suficientemente largo para generar resultados reproducibles que sean representativos del rendimiento del sistema que se está probando.

3.5. Resumen del análisis

Dado el gran número de requisitos que son necesarios para completar una implementación del benchmark TPC-C, y para completar la documentación de este análisis, se van a incluir una serie de casos de uso que simbolizan de manera abstracta los procesos más significativos que se realizan en dicho benchmark.

3.5.1. Casos de uso

Diagrama de casos de uso

Comenzaremos con el diagrama general de casos de uso (Fig. 3.2), donde aparecen dos actores en el sistema que simbolizan:

- **Usuario del benchmark:** La persona que ejecuta el benchmark para obtener una medida de rendimiento de un sistema.
- **Usuario simulado:** Que a través de la terminal introduce los datos de las transacciones en el sistema. Se puede decir que es el operador virtual del terminal.

Configurar Benchmark

Introducción

A la hora de ejecutar el benchmark, existen unos parámetros importantes de los que depende el resultado final y el modo de trabajo del benchmark. Es aquí donde se indica que necesita el benchmark para funcionar.

Actores

El usuario del benchmark.

Pre condiciones

Ninguna, ya que esto sucede lo primero de todo.

Post condiciones

El benchmark se considera *configurado* y se puede ejecutar con esta configuración.

Acciones

1. El usuario del benchmark indica que quiere configurar los parámetros de ejecución del benchmark TPC-C.
2. El sistema le facilita un interfaz para realizar dicha configuración.

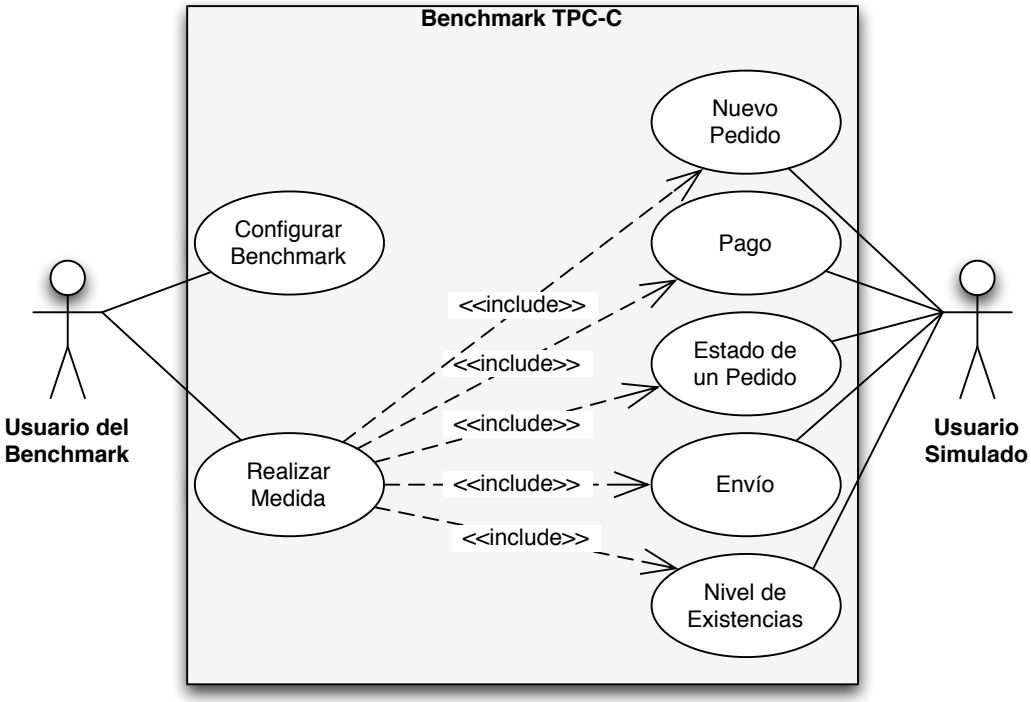

Figura 3.2: Casos de uso del benchmark TPC-C

3. El usuario introduce:
    - El número de terminales que van a simularse.
    - Cuántas transacciones van a realizarse en total.
    - El número de almacenes a la hora de generar un poblado de datos.
4. El sistema almacena dichos datos como una configuración del benchmark.
5. El sistema queda listo para ejecutarse con esa nueva configuración.

Relaciones con otros casos de uso

De manera directa ninguna, pero lo sucedido en este caso de uso es necesario para que el resto de casos de uso funcionen.

Realizar Medida

Introducción

Esta es la ejecución del benchmark puramente, donde se realizan una serie de tareas y se obtienen unos resultados en base a la métrica de rendimiento que se ha definido en el análisis. Este caso de uso depende de aquellos casos de uso que simbolizan al usuario simulado y las tareas que realiza.

Actores

El usuario del benchmark.

Pre condiciones

El sistema tiene que estar *configurado*, por lo que se debe disponer de alguna configuración antes de ejecutar el benchmark.

Post condiciones

- Se obtiene una medición del rendimiento del sistema así como otros datos estadísticos.
- Se ha ejecutado el benchmark correctamente.

Acciones

1. El usuario indica al sistema una configuración del mismo y lo lanza para obtener una medición de rendimiento.
2. El sistema crea una serie de tablas en un medio de almacenamiento y las puebla (llena de datos) tal y como se indica en la sección de *Poblado* (Pag. 59).

   Para ello utiliza el parámetro de configuración referente al número de almacenes.
3. El sistema prepara el número de terminales indicadas y con ese número prepara también el número de instancias del sistema de almacenamiento de datos que deben estar preparadas para responder a la carga.

4. El sistema reparte el número total de transacciones entre las terminales.
5. Cada terminal ejecuta las transacciones adecuadas para cumplir el porcentaje de transacciones indicado en la sección de *Métrica de rendimiento* (Pag. 64).
    - Nuevo pedido ($\ll include \gg$ Caso de uso de nuevo pedido).
    - Pago ($\ll include \gg$ Caso de uso de Pago).
    - Estado de un pedido ($\ll include \gg$ Caso de uso de Estado de un pedido).
    - Envío ($\ll include \gg$ Caso de uso de Envío)
    - Nivel de existencias ($\ll include \gg$ Caso de uso de Nivel de existencias).
6. El sistema espera información por parte de los servidores encargados de atender las peticiones de las terminales.
7. El sistema realiza una estadística y muestra la medida de rendimiento así como infamación extra del número de transacciones ejecutadas.

Relaciones con otros casos de uso

Para la correcta realización de este caso de uso se necesita de los siguientes casos de uso: Nuevo Pedido, Pago, Estado de un pedido, Envío y Nivel de existencias.

Nuevo Pedido

Introducción

Transacción de nuevo pedido, donde el sistema simula que la entrada de un pedido adicional al negocio.

Actores

El usuario simulado.

Pre condiciones

- Es sistema está en ejecución.
- El sistema de almacenamiento ha sido poblado con datos según las reglas de poblado (Pag. 3.3).
- Esta transacción se ejecuta en un servidor lanzado para atender a las peticiones de una terminal simulada.

Post condiciones

- Se ha insertado un nuevo pedido en el sistema
- El sistema de benchmark sigue funcionando.
- El sistema de almacenamiento contiene datos correctos y no corruptos.

Acciones

1. El usuario simulado ha decidido que quiere realizar una transacción de nuevo pedido.
2. El usuario simulado provee de los datos necesarios para dicha transacción a la terminal (Todos los detalles en *Nuevo Pedido, datos de entrada*, Pag. 51).
    a) El número de almacén local (W_ID).
    b) El número de la zona (D_ID).
    c) El número de líneas de pedido (O_OL_CNT).
    d) Para cada línea de pedido:
        1) Un número de producto (OL_I_ID).
        2) Almacén que provee el producto (OL_SUPPLY_W_ID).
        3) Una cantidad de producto (OL_QUANTITY).
        4) Una fecha para el pedido (O_ENTRY_D).
3. La terminal encarga el siguiente trabajo al servidor que se le ha asignado (Todos los detalles en *Nuevo Pedido, perfil de la transacción*, Pag. 52).
    a) Obtener los datos del almacén W_ID.
    b) Obtener los datos de la zona D_W_ID,D_ID e incrementar O_NEXT_O_ID.
    c) Obtener el cliente C_W_ID, C_D_ID, C_ID y obtener sus datos.
    d) Crear una entrada en pedido y en nuevo pedido con la misma clave.
    e) Para cada línea de pedido:
        1) Buscamos el producto.
        2) Actualizamos las existencias del producto en el almacén.
        3) Calculamos el precio de esa línea
        4) Insertamos una tupla en líneas de pedido con estos datos.
4. La transacción se da por finalizada.

Relaciones con otros casos de uso

Es utilizado por el caso de uso: realizar medida.

Pago

Introducción

Transacción de pago, donde se simula que un cliente paga una cantidad de dinero para cubrir su deuda con la empresa.

Actores

El usuario simulado.

Pre condiciones

- Es sistema está en ejecución.
- El sistema de almacenamiento ha sido poblado con datos según las reglas de poblado (Pag. 59).
- Esta transacción se ejecuta en un servidor lanzado para atender a las peticiones de una terminal simulada.

Post condiciones

- Se ha anotado el pago del cliente.
- El sistema de benchmark sigue funcionando.
- El sistema de almacenamiento contiene datos correctos y no corruptos.

Acciones

1. El usuario simulado ha decidido que quiere realizar una transacción de pago.
2. El usuario simulado provee de los datos necesarios para dicha transacción a la terminal (Todos los detalles en *Pago, datos de entrada*, Pag 53).
   a) El número de almacén (W_ID).
   b) El número de la zona (D_ID).
   c) Los datos de un cliente: a veces por su identificador (C_W_ID, C_D_ID, C_ID), y a veces por su apellido (C_LAST).
   d) La cantidad a pagar (H_AMOUNT).
   e) La fecha del pago (H_DATE).
3. La terminal encarga el siguiente trabajo al servidor que se le ha asignado (Todos los detalles en *Pago, perfil de la transacción*, Pag. 54).
   a) Obtener los datos del almacén (W_ID) e incrementar su balance económico en H_AMOUNT.
   b) Obtener los datos de la zona (D_W_ID, D_ID) e incrementar su balance económico en H_AMOUNT.
   c) Obtener los datos del cliente y:
      1) Incrementar su balance anual en H_AMOUNT.
      2) Disminuir el balance actual en H_AMOUNT.
      3) Incrementar en uno el número de pagos.
      4) Dependiendo del campo C_CREDIT actualizar el campo C_DATA.
   d) Preparar una entrada en el histórico con su campo H_DATA.
   e) Añadir dicha entrada al histórico.

Relaciones con otros casos de uso

Es utilizado por el caso de uso: realizar medida.

Estado de un pedido

Introducción

Transacción de consulta de estado de un pedido; el sistema simula consultar el estado del último pedido realizado por el cliente.

Actores

El usuario simulado.

Pre condiciones

- Es sistema está en ejecución.
- El sistema de almacenamiento ha sido poblado con datos según las reglas de poblado (Pag. 59).
- Esta transacción se ejecuta en un servidor lanzado para atender a las peticiones de una terminal simulada.

Post condiciones

- Se ha informado al cliente del estado de su último pedido.
- El sistema de benchmark sigue funcionando.
- El sistema de almacenamiento contiene datos correctos y no corruptos.

Acciones

1. El usuario simulado ha decidido que quiere realizar una transacción de consulta de estado de un pedido.
2. El usuario simulado provee de los datos necesarios para dicha transacción a la terminal (Todos los detalles en *Estado de pedido, datos de entrada*, Pag. 55).
   a) El número de almacén (W_ID).
   b) El número de la zona (D_ID).
   c) Los datos de un cliente: a veces por su identificador (C_W_ID, C_D_ID, C_ID), y a veces por su apellido (C_LAST).
3. La terminal encarga el siguiente trabajo al servidor que se le ha asignado (Todos los detalles en *Estado de pedido, perfil de la transacción*, Pag. 55).
   a) Se obtienen los datos del cliente.
   b) Se obtiene el pedido del cliente con mayor O_ID.
   c) Se obtienen todas las líneas de dicho pedido.

Relaciones con otros casos de uso

Es utilizado por el caso de uso: realizar medida.

Envío

Introducción

Transacción de envío, donde el sistema elige un pedido que no esté enviado, selecciona uno de cada zona por cada almacén, y lo tramita como enviado.

Actores

El usuario simulado.

Pre condiciones

- Es sistema está en ejecución.
- El sistema de almacenamiento ha sido poblado con datos según las reglas de poblado (Pag. 59).
- Esta transacción se ejecuta en un servidor lanzado para atender a las peticiones de una terminal simulada.

Post condiciones

- Se ha marcado el pedido como enviado.
- El sistema de benchmark sigue funcionando.
- El sistema de almacenamiento contiene datos correctos y no corruptos.

Acciones

1. El usuario simulado ha decidido que quiere realizar una transacción de envío de pedidos.
2. El usuario simulado provee de los datos necesarios para dicha transacción a la terminal (Todos los detalles en *Envío, datos de entrada*, Pag. 56).
   a) El número de almacén (W_ID).
   b) El número de la empresa de transportes (O_CARRIER_ID).
   c) La fecha de envío (OL_DELIVERY_D).
3. La terminal encarga el siguiente trabajo al servidor que se le ha asignado (Todos los detalles en *Envío, perfil de la transacción*, Pag. 56). Para cada una de las zonas del almacén escogido:
   a) Se selecciona el nuevo pedido más viejo, es decir el NO_O_ID más pequeño, y se borra.
   b) Se selecciona el pedido correspondiente al nuevo pedido borrado y se actualiza la empresa de transportes (O_CARRIER_ID).
   c) Para todas las líneas de pedido, calculamos el coste total y actualizamos en cada línea la fecha de envío.
   d) Al balance actual del cliente se le suma el total de las líneas de pedido y se incrementa el número de pedidos enviados a ese cliente.

Relaciones con otros casos de uso

Es utilizado por el caso de uso: realizar medida.

Nivel de existencias

Introducción

Transacción de consulta de nivel de existencias, donde se revisan las últimas líneas de pedido de los últimos 20 pedidos en una zona y de esos productos se comprueban las existencias por si hace falta encargar más.

Actores

El usuario simulado.

Pre condiciones

- Es sistema está en ejecución.
- El sistema de almacenamiento ha sido poblado con datos según las reglas de poblado (Pag. 59).
- Esta transacción se ejecuta en un servidor lanzado para atender a las peticiones de una terminal simulada.

Post condiciones

- Se ha comprobado sin problemas las existencias de los productos incluidos en los últimos 20 pedidos de una zona.
- El sistema de benchmark sigue funcionando.
- El sistema de almacenamiento contiene datos correctos y no corruptos.

Acciones

1. El usuario simulado ha decidido que quiere realizar una transacción de comprobar el nivel de existencias.
2. El usuario simulado provee de los datos necesarios para dicha transacción a la terminal (Todos los detalles en *Nivel de existencias, datos de entrada*, Pag. 57).
   a) Un almacén (W_ID).
   b) Una zona dentro de ese almacén (D_ID).
   c) Un límite mínimo de existencias.
3. La terminal encarga el siguiente trabajo al servidor que se le ha asignado (Todos los detalles en *Nivel de existencias, perfil de la transacción*, Pag. 57).
   a) Obtenemos los datos de la zona, incluyendo el próximo número de pedido.

b) Obtenemos todas las líneas de pedido correspondientes a los últimos 20 pedidos de esa zona.

c) Para cada producto en la línea de pedido, comprobamos que el stock del almacén no esté por debajo del límite indicado.

Relaciones con otros casos de uso

Es utilizado por el caso de uso: realizar medida.

3.5.2. Diagramas de secuencia

Por último, dentro de los casos de uso y para intentar sacar una pequeña lista de clases o subsistemas de análisis, así como intentar ver las partes en las que se divide la organización del benchmark TPC-C con las especificaciones que hemos indicado; se incluyen los diagramas de secuencia para los casos de uso más significativos.

Configurar Benchmark

Figura 3.3.

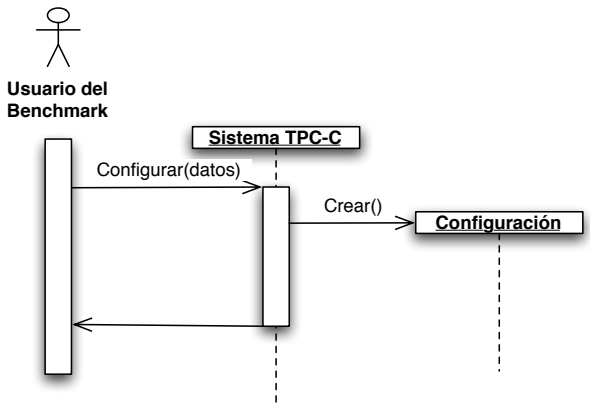

Figura 3.3: Diagrama de secuencia del caso de uso *Configurar Benchmark*

Realizar Medida

Figura 3.4.

Nuevo Pedido

Figura 3.5.

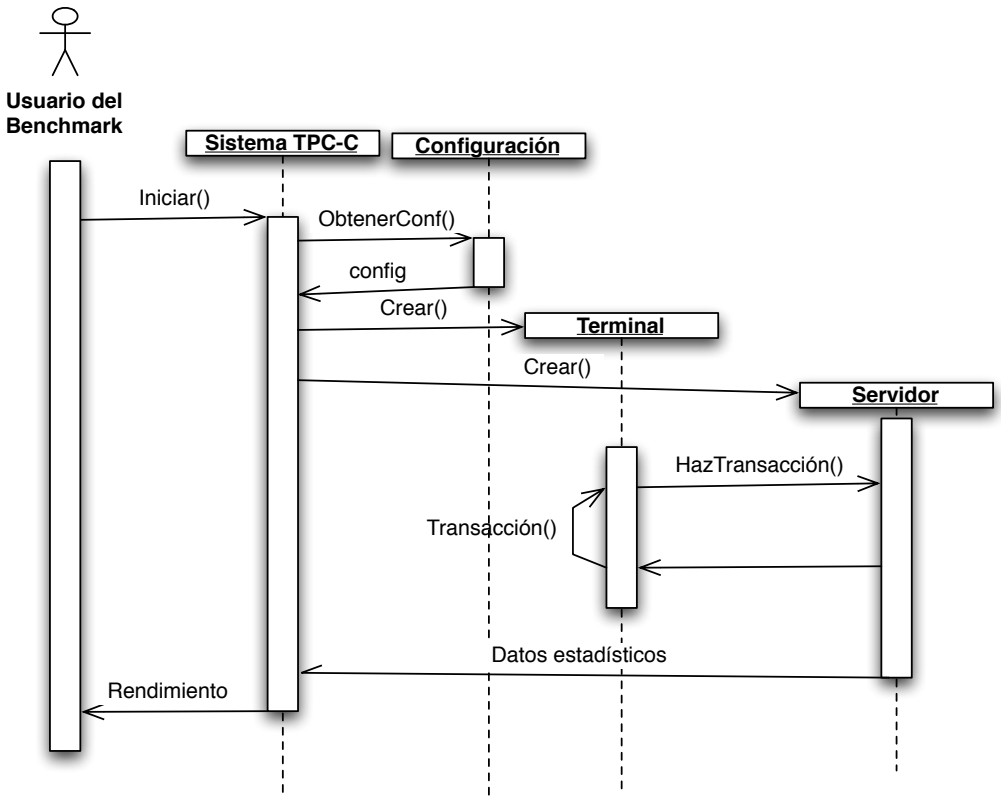

Figura 3.4: Diagrama de secuencia del caso de uso *Realizar Medida*

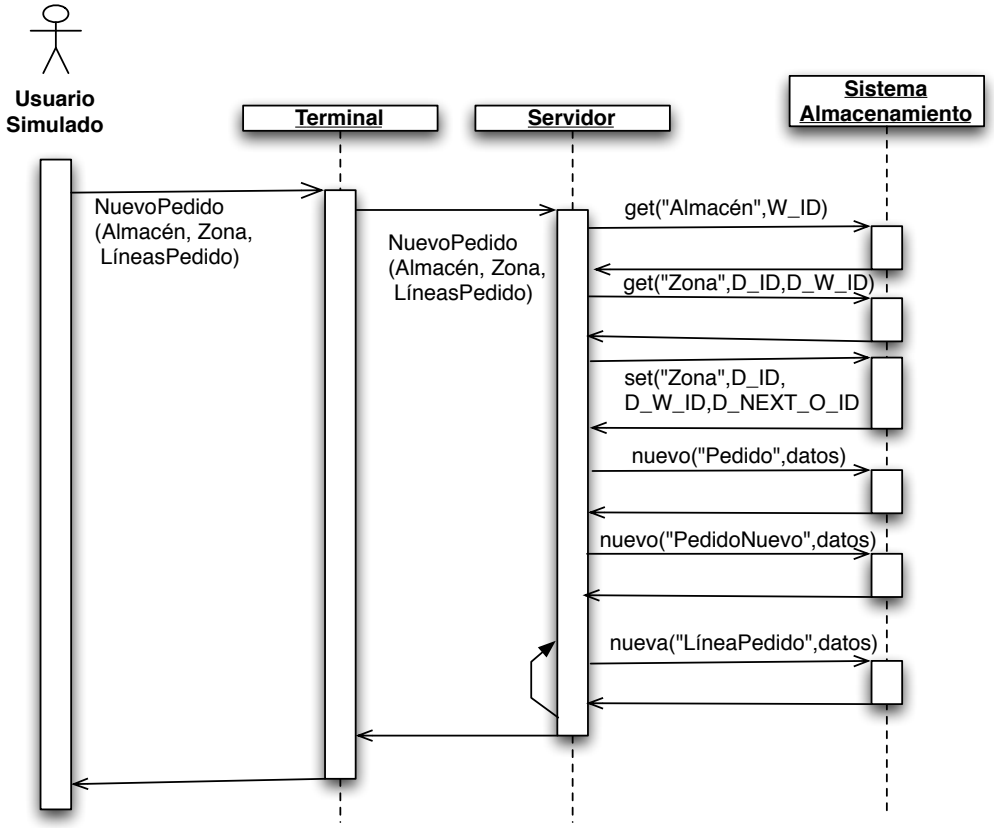

Figura 3.5: Diagrama de secuencia del caso de uso *Nuevo Pedido*

3.5.3. Diagrama de clases de análisis

De estos diagramas de secuencia, ya que se han expuesto algunas posibles clases, podemos derivar las siguientes clases de análisis.

- *Clase Terminal*: Dado que el usuario simulado, al fin y al cabo es simulado, los datos necesarios para cada transacción los *generará* esta clase.
- *Clase Servidor*: Es la encargada de realizar las transacciones; tiene una lista de los almacenes disponibles, y los utiliza como referencia para acceder a los datos. Realiza la lógica de negocio.
- *Clase Sistema Almacenamiento*: Maneja las estructuras de datos para acceder, añadir, borrar y modificar registros de datos. Esta clase es usada por los múltiples servidores de manera concurrente.
- *Clase IdAlmacenamiento*: Para completar a la clase servidor, y dado que en algún lugar se tienen que guardar las referencias a los diferentes sistemas de almacenamiento, que más tarde se enviaran al sistema de almacenamiento para que opere con ellos, se ha creado esta clase.

El diagrama de clases de análisis se encuentra en la figura 3.6.

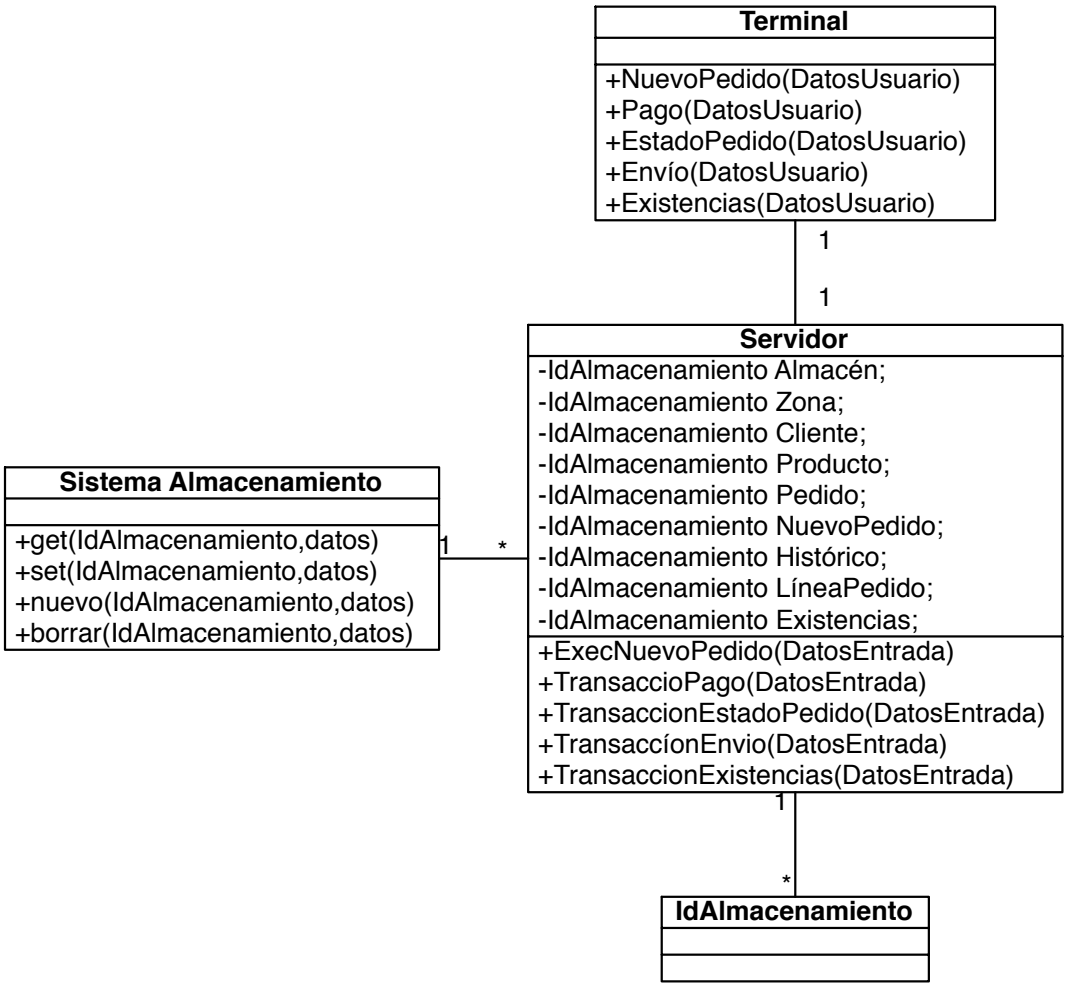

Figura 3.6: Diagrama de clases de análisis

Capítulo 4

Diseño e implementación

En las especificaciones originales del benchmark TPC-C, existen reglas y parámetros para realizar un análisis de rendimiento completo: desde la herramienta a implementar, hasta plantillas para los documentos que hay que presentar así como una explicación detallada de los datos a obtener.

En este caso lo que se va a diseñar e implementar es la herramienta que permite obtener un resultado numérico del rendimiento de un sistema con su simple ejecución con una configuración concreta.

4.1. Diseño de la arquitectura

Para explicar el diseño descendente que se ha realizado, se empezará por el diagrama de arquitectura, que dará paso a los diferentes subsistemas de la aplicación. Ya que la aplicación de primeras tiene dos partes diferenciadas: subsistema de medida de rendimiento y subsistema de almacenamiento de datos, el trabajo aplicado para dividir lo mejor posible en módulos el programa ha sido importante, ya que si no, la cantidad de código acumulada en un solo fichero hubiera hecho difícil la búsqueda y solución de problemas, así como casi imposible la adaptación a pequeños cambios o actualizaciones.

4.1.1. Diseño de los subsistemas

Empezaremos con una división funcional de los componentes del sistema agrupados en subsistemas. El sistema principal que implementa el benchmark se divide en 3 subsistemas (Fig. 4.1).

- Subsistema generador: Se encarga de generar los datos de *poblado* como los datos de los usuarios simulados que al final acaban siendo los datos que las *terminales* envían a los servidores para que realicen una transacción.
- Subsistema TPC-C: es el programa principal del benchmark, que se divide en 2 componentes:
    - Administrador: es el encargado de cuidar y preparar el sistema para que los servidores comiencen su trabajo. Realiza las siguientes tareas cortas:
        - Inicializa el subsistema de almacenamiento.
        - Carga el *poblado* generado anteriormente en el subsistema de almacenamiento.
        - Lanza los servidores.

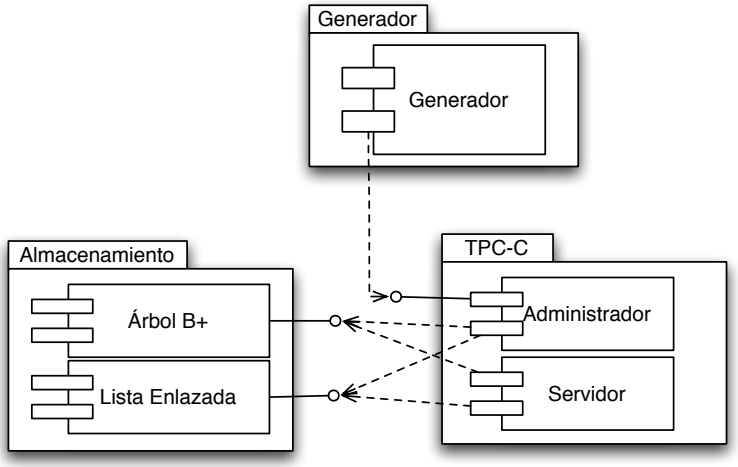

Figura 4.1: Diferentes subsistemas del benchmark TPC-C

    - Espera sus estadísticas y las da un formato final.
  - Servidor: se encarga de realizar las transacciones que tiene indicadas en los datos que las *terminales* le han encargado. Trabaja de manera conjunta con el subsistema de almacenamiento: el servidor se encarga de la lógica del negocio, y el almacenamiento de guardar y recuperar los datos.

- Subsistema de almacenamiento: Se encarga de almacenar datos de manera genérica y de proporcionar un método rápido de acceso y modificación de los mismos. Dado que no siempre las necesidades son las mismas, se emplean 2 componentes para prestar este servicio:

  - Árbol B+: proporciona un método de almacenamiento indexado, donde se permite: buscar, insertar, eliminar y modificar; siendo la operación más rápida la de buscar.
  - Lista Enlazada: debido a que hay tablas que no tienen clave, este sistema de almacenamiento permite de una manera sencilla almacenar estos datos sin campos clave. Solo permite 2 operaciones: añadir y borrar.

  En ambos componentes existe una *sincronización* para que múltiples procesos o hilos de ejecución puedan acceder a los datos sin interferir entre si.

4.1.2. Diseño detallado de la arquitectura

Existen varios subsistemas y no todos ellos se ejecutan a la vez, ni en el mismo proceso, por lo que hace falta un diagrama que explique claramente cómo es el funcionamiento habitual del benchmark antes de explicar en detalle los fundamentos y la composición de cada subsistema.

En la figura 4.2, se puede observar de manera menos estándar (sin usar UML) cómo se distribuye todo el sistema TPC-C en diferentes sistemas y procesos, por lo que se va detallar el funcionamiento de los mismos.

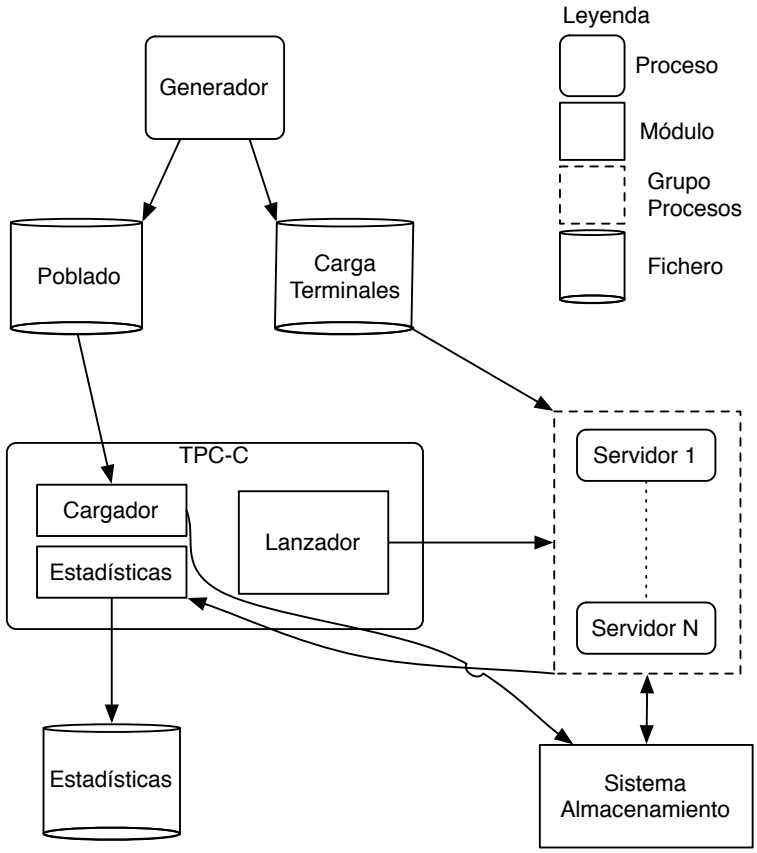

Figura 4.2: Arquitectura del sistema TPC-C

1. **Proceso Generador:** es el primero que se ejecuta para producir un *experimento* que consiste en:

   - Unos datos de *poblado*, que servirán como datos iniciales a la hora de que los servidores comiencen a funcionar.
   - Simula a los usuarios de las terminales, y con ello genera unos datos que son los que recibirán los servidores para realizar las transacciones. Es lo que llamaremos *carga* de trabajo generada por las terminales.

   El proceso generador necesita además los parámetros de funcionamiento:

   - La cantidad de *almacenes* para poblar la base de datos. Recordemos que el almacén es la unidad de generación de carga inicial, cada almacén contiene 10 zonas, y cada zona sus clientes pedidos y entradas en el histórico; por lo que al multiplicar el número de almacenes se multiplican también estos datos.
   - La cantidad de *servidores* que simultáneamente van a atender peticiones de las terminales. Con este parámetro se generarán diferentes ficheros de *carga*
   - Pero para contabilizar esta carga hace falta indicar también el *número de transacciones* que se van a ejecutar en total; según este número, se repartirán esas transacciones entre los diferentes servidores.

   Por lo que al finalizar la ejecución del proceso generador obtenemos los datos de un *experimento* en **ficheros**, que son:

   - Datos de poblado del sistema de almacenamiento.
   - Número de servidores.
   - Carga para dichos servidores.

2. **Proceso TPC-C:** Es el encargado de realizar la medición del rendimiento, para ello preparará los datos necesarios para que el sistema pueda trabajar.

   a) Lo primero es ejecutar un *cargador*, que inicializa los sistemas de almacenamiento y los puebla con los datos de poblado. Es una operación repetitiva y no se tiene en cuenta a la hora de medir el rendimiento.

   b) Luego se prepara la sincronización de los procesos servidores y se *lanzan*, dichos procesos servidores, son unidades de ejecución independientes que según el modelo de paralelización pueden ser hilos de ejecución o procesos.

   c) Por último, espera a que terminen los servidores, y mediante **memoria compartida** recoge los datos estadísticos de cada servidor y escribe un fichero con un resumen y una medida de rendimiento.

   En este proceso se ejecuta el *subsistema de almacenamiento*, este sistema es el mismo entre los procesos TPC-C y servidores gracias a que está situado en **memoria compartida**, por lo que los datos almacenados en la fase de carga son los mismos que se utilizarán por los servidores.

3. **Proceso servidor:** Pueden existir múltiples procesos servidores, pero se explicará el funcionamiento de uno en concreto. Cuando se lanza un proceso servidor, este realiza un trabajo cíclico consistente en procesar transacciones; este es el orden de operaciones:

a) Obtiene una transacción junto con sus datos del fichero de carga.

b) La identifica y la realiza.

c) Anota la transacción realizada en un área de **memoria compartida**.

Si bien este proceso parece sencillo, es el que más trabajo necesita para realizarse ya que cada transacción lleva implícita una carga de trabajo. También es aquí donde se explotan todas las características de multiproceso del sistema de almacenamiento mediante un sistema *lector-escritor* que se explicará más adelante. Este sistema permite una ordenación de los accesos de tal manera que para una tabla, sólo un proceso pueda estar modificándola, pero cuando no se está modificando, múltiples procesos puedan estar accediendo a sus datos.

4.1.3. Tecnologías de la implementación

Una vez definida la arquitectura, hay que fijar que tecnologías se utilizarán; es verdad que definiendo la arquitectura ya se han dicho bastantes de estas tecnologías, pero vamos a listar e introducir cada una de ellas.

Árboles B+

Los árboles B y los árboles B+ [20] son casos especiales de árboles de búsqueda. Un árbol de búsqueda es un tipo de árbol que sirve para guiar la localización de un registro, dado el valor de uno de sus campos. Los índices multinivel pueden considerarse como variaciones de los árboles de búsqueda. Cada bloque o nodo del índice multinivel puede tener hasta p valores del campo de indexación y p punteros. Los valores del campo de indexación de cada nodo guían al siguiente nodo (que se encuentra en otro nivel), hasta llegar al bloque del fichero de datos que contiene el registro deseado. Al seguir un puntero, se va restringiendo la búsqueda en cada nivel a un subárbol del árbol de búsqueda, y se ignoran todos los nodos que no estén en dicho subárbol.

Los árboles de búsqueda difieren un poco de los índices multinivel. Un árbol de búsqueda de orden p es un árbol tal que cada nodo contiene como mucho $p - 1$ valores del campo de indexación y p punteros, colocados de la siguiente manera:

$$(P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q) \tag{4.1}$$

Donde:

- $q \leq p$
- P_i es un puntero a un nodo hijo.
- K_i es un valor de búsqueda o valor clave.

Además se cumple que:

1. Todos los valores de búsqueda son únicos.
2. Dentro de cada nodo se cumple: $K_1 < K_2 < ...K_{q-1}$
3. Para todos los valores de X del subárbol al que nos lleva el puntero P_i, se tiene:
   - $K_{i-1} < X < K_i$ para $1 < i < q$

- $X < K_i$ para $i = 1$
- $K_{i-1} < X$ para $i = q$

Al buscar un valor X, se sigue el puntero P_i apropiado de acuerdo con las 3 restricciones del punto 3. Para insertar valores de búsqueda en el árbol y eliminarlos, sin violar las restricciones anteriores, se utilizan algoritmos que no garantizan que el árbol de búsqueda esté equilibrado (que todas las hojas estén al mismo nivel). Es importante mantener equilibrados los árboles de búsqueda porque esto garantiza que no habrá nodos en niveles muy profundos que requieran muchos accesos a bloques durante una búsqueda. Además, las eliminaciones de registros pueden hacer que queden nodos casi vacíos, con lo que hay un desperdicio de espacio importante que también provoca un aumento en el número de niveles.

El árbol B es un árbol de búsqueda, con algunas restricciones adicionales, que resuelve hasta cierto punto los dos problemas anteriores. Estas restricciones adicionales garantizan que el árbol siempre estará equilibrado y que el espacio desperdiciado por la eliminación, si lo hay, nunca será excesivo. Los algoritmos para insertar y eliminar se hacen más complejos para poder mantener estas restricciones. No obstante, la mayor parte de las inserciones y eliminaciones son procesos simples, se complican sólo en circunstancias especiales: cuando se intenta insertar en un nodo que está lleno o cuando se intenta borrar en un nodo que está ocupado hasta la mitad.

Árbol B

Un árbol B de orden p, se define de la siguiente manera:

1. La estructura de cada nodo tiene la siguiente forma:

$$(P_1, (K_1, Pr_1), P_2, (K_2, Pr_2), P_3, (K_3, Pr_3), ...P_{q-1}, (K_{q-1}, Pr_{q-1}), P_q) \tag{4.2}$$

   Donde:

   - $q \leq p$
   - P_i es un puntero a un nodo hijo.
   - Pr_i es un puntero al nodo de datos cuya clave es K_i

2. Dentro de cada nodo se cumple: $K_1 < K_2 < ...K_{q-1}$
3. Para todos los valores de X del subárbol al que nos lleva el puntero P_i, se tiene:
   - $K_{i-1} < X < K_i$ para $1 < i < q$
   - $X < K_i$ para $i = 1$
   - $K_{i-1} < X$ para $i = q$
4. Cada nodo tiene como mucho p punteros a otros nodos del árbol.
5. Cada nodo excepto el raíz (el primero) y las hojas (los últimos), tiene al menos $p/2$ punteros a nodos del árbol. El nodo raíz tiene como mínimo 2 punteros a nodos del árbol, excepto si es el único nodo.
6. Un nodo con q punteros a nodos y $q \leq p$, tiene $q - 1$ campos de indexación o campos clave.
7. Todos los nodos hoja están al mismo nivel. Los nodos hoja tienen la misma estructura que los nodos internos, pero los punteros a nodos del árbol son nulos.

Como se puede observar, en los árboles B todos los valores del campo de indexación aparecen alguna vez en algún nivel del árbol, junto con un puntero al fichero de datos.

Árbol B+

En un árbol B+ los punteros a datos se almacenan sólo en los nodos hoja del árbol, por lo cual, la estructura de los nodos hoja difiere de la de los nodos internos. Los nodos hoja tienen una entrada por cada valor del campo de indexación, junto con un puntero al registro del fichero de datos. Estos nodos están enlazados para ofrecer un acceso ordenado a los registros a través del campo de indexación. Los nodos hoja de un árbol B+ son similares al primer nivel (nivel base) de un índice. Los nodos internos del árbol B+ corresponden a los demás niveles del índice. Algunos valores del campo de indexación se repiten en los nodos internos del árbol B+ con el fin de guiar la búsqueda.

En un árbol B+ de orden p, la estructura de los nodos *internos* es la siguiente:

1. Todo nodo interno es de la forma:

$$(P_1, K_1, P_2, K_2, P_3, K_3, ...P_{q-1}, K_{q-1}, P_q) \tag{4.3}$$

   Donde:

   - $q \leq p$
   - Cada P_i es un puntero a un nodo interno u hoja del árbol.

2. Dentro de cada nodo interno se cumple: $K_1 < K_2 < ...K_{q-1}$

3. Para todos los valores de X del subárbol al que nos lleva el puntero P_i, se tiene:

   - $K_{i-1} < X < K_i$ para $1 < i < q$
   - $X < K_i$ para $i = 1$
   - $K_{i-1} < X$ para $i = q$

4. Cada nodo interno tiene como mucho p punteros a otros nodos del árbol.

5. Cada nodo interno excepto el raíz tiene al menos $p/2$ punteros a nodos del árbol. El nodo raíz, si es interno, tiene al menos dos punteros a nodos del árbol.

6. Un nodo con q punteros a nodos y $q \leq p$, tiene $q - 1$ campos clave.

La estructura de los nodos *hoja* de un árbol B+ de orden p es la siguiente:

1. Todo nodo hoja es de la forma:

$$((Pr_1, K_1), (Pr_2, K_2), (Pr_3, K_3), ...(Pr_{q-1}, K_{q-1}), P_siguiente) \tag{4.4}$$

   Donde:

   - $q \leq p$
   - Cada Pr_i es un puntero al registro de datos cuya clave es K_i.
   - $P_siguiente$ Es un puntero al nodo hoja siguiente.

2. Cada nodo hoja tiene al menos $p/2$ valores.

3. Todos los nodos hoja están al mismo nivel.

Como las entradas en los nodos internos de los árboles B+ contienen valores del campo de indexación y punteros a nodos del árbol, pero no contienen punteros a los registros del fichero de datos, es posible *empaquetar* más entradas en un nodo interno de un árbol B+ que en un nodo similar de un árbol B. Por tanto, si el tamaño de bloque (nodo) es el mismo, el orden p será mayor para el árbol B+ que para el árbol B. Esto puede reducir el número de niveles del árbol B+, mejorándose así el tiempo de acceso. Como las estructuras de los nodos internos y los nodos hoja de los árboles B+ son diferentes, su orden p puede ser diferente.

Se ha demostrado [20] por análisis y simulación que después de un gran número de inserciones y eliminaciones aleatorias en un árbol B, los nodos están ocupados en un 69 % cuando se estabiliza el número de valores del árbol. Esto también es verdadero en el caso de los árboles B+. Si llega a suceder esto, la división y combinación de nodos ocurrirá con muy poca frecuencia, de modo que la inserción y la eliminación se volverán muy eficientes.

Lenguaje C

Para implementar el benchmark se ha escogido como lenguaje de programación el lenguaje C. C es un lenguaje de programación de propósito general que ofrece economía sintáctica, control de flujo y estructuras sencillas y un buen conjunto de operadores. No es un lenguaje de muy alto nivel y más bien un lenguaje pequeño, sencillo y no está especializado en ningún tipo de aplicación. Esto lo hace un lenguaje potente, con un campo de aplicación ilimitado y sobre todo, se aprende rápidamente. En poco tiempo, un programador puede utilizar la totalidad del lenguaje [19].

Este lenguaje ha estado estrechamente unido al sistema operativo UNIX, puesto que fueron desarrollados conjuntamente. Sin embargo, este lenguaje no está ligado a ningún sistema operativo ni a ninguna máquina concreta. Se le suele llamar lenguaje de programación de sistemas debido a su utilidad para escribir compiladores y sistemas operativos, aunque de igual forma se puede desarrollar cualquier tipo de aplicación.

La base del C proviene del BCPL, escrito por Martin Richards, y del B escrito por Ken Thompson en 1970 para el primer sistema UNIX en un DEC PDP-7. Estos son lenguajes sin tipos, al contrario que el C que proporciona varios tipos de datos. Los tipos que ofrece son caracteres, números enteros y en coma flotante, de varios tamaños. Además se pueden crear tipos derivados mediante la utilización de punteros, vectores, registros y uniones. El primer compilador de C fue escrito por Dennis Ritchie para un DEC PDP-11 y escribió el propio sistema operativo en C.

C trabaja con tipos de datos que son directamente tratables por el hardware de la mayoría de computadoras actuales, como son los caracteres, números y direcciones. Estos tipos de datos pueden ser manipulados por las operaciones aritméticas que proporcionan los procesadores. No proporciona mecanismos para tratar tipos de datos que no sean los básicos, debiendo ser el programador el que los desarrolle. Esto permite que el código generado sea muy eficiente y de ahí el éxito que ha tenido como lenguaje de desarrollo de sistemas. No proporciona otros mecanismos de almacenamiento de datos que no sea el estático y no proporciona mecanismos de entrada ni salida. Ello permite que el lenguaje sea reducido y los compiladores de fácil implementación en distintos sistemas. Por contra, estas carencias se compensan mediante la inclusión de funciones de librería para realizar todas estas tareas, que normalmente dependen del sistema operativo.

Originariamente, el manual de referencia del lenguaje para el gran público fue el libro de Kernighan y Ritchie, escrito en 1977. Es un libro que explica y justifica totalmente el desarrollo de aplicaciones en C, aunque en él se utilizaban construcciones, en la definición de funciones, que podían provocar confusión y errores de programación que no eran detectados por el compilador. Como los tiempos cambian y las necesidades también, en 1983 ANSI establece el comité X3J11 para que desarrolle una definición moderna y comprensible del C. El estándar está basado en el manual de

referencia original de 1972 y se desarrolla con el mismo espíritu de sus creadores originales. La primera versión de estándar se publicó en 1988 y actualmente todos los compiladores utilizan la nueva definición. Una aportación muy importante de ANSI consiste en la definición de un conjunto de librerías que acompañan al compilador y de las funciones contenidas en ellas. Muchas de las operaciones comunes con el sistema operativo se realizan a través de estas funciones. Una colección de ficheros de encabezamiento, headers, en los que se definen los tipos de datos y funciones incluidas en cada librería. Los programas que utilizan estas bibliotecas para interactuar con el sistema operativo obtendrán un comportamiento equivalente en otro sistema.

PARMACS

Estas macros implementan código que permite concurrencia y sincronización en múltiples arquitecturas, para que las aplicaciones sean programadas de manera independiente y puedan ser trasladadas a otros sistemas simplemente cambiando la implementación de dichas macros.

Las macros han sido desarrolladas en el Argonne National Laboratory y son empleadas usando el preprocesador m4. Las especificaciones originales de estas macros se pueden encontrar en [7] [8]. PARMACS ofrece unas primitivas de sincronización básicas, creación paralela de procesos y asignación de memoria compartida; existen muchas implementaciones para diferentes sistemas como Encore Multimax, SGI, Alliant y otros, que se encuentran disponibles al público.

Para las pruebas y simulaciones se ha utilizado una versión de PARMACS implementada usando hilos POSIX en C. Aun así, y de forma general estás son las funciones que han sido usadas, dentro del extenso catálogo de PARMACS:

- *MAIN_INITENV(int numproc)*: Esta macro inicializa el entorno de PARMACS. El primer argumento es opcional e indica el número máximo de procesos. Se utiliza en la función principal del programa para inicializar el sistema de control de procesos.

- *MAIN_END()*: Esta macro finaliza el entorno PARMACS, y debe de ser lo último que ejecute la aplicación.

- *MAIN_ENV()*: Esta macro declara las variables y las definiciones de las estructuras utilizadas por el entorno PARMACS. Debe aparecer sólo una vez en la aplicación, al principio del fichero de código principal. En el caso de la implementación mediante hilos POSIX, no requiere ningún parámetro

- *EXTERN_ENV()*: Esta macro contiene definiciones de estructuras utilizadas, y es útil para el resto de ficheros de código que, sin ser el principal, utilicen las macros y por lo tanto necesiten al menos tener constancia de como se definen la estructuras que se usan.

- *CLOCK(int reloj)*: Esta macro almacena en la variable reloj la fecha actual, las unidades en esta implementación son los segundos transcurridos desde el 1 de enero de 1970.

- *CREATE(void (*proc)(void))*: Esta macro crea un nuevo proceso que ejecutará la función indicada por “proc”. El nuevo proceso podrá acceder a toda aquella memoria que este compartida.

- *WAIT_FOR_END()*: Esta macro bloquea el proceso que la utiliza hasta que hayan finalizado todos los procesos lanzados con CREATE. En esta implementación no lleva argumentos, pero en otras se suele indicar el número de procesos por los que esperar.

- *G_MALLOC(int tam)*: Asigna “tam” bytes de memoria compartida y devuelve un puntero a dicha zona. G_MALLOC termina con un punto y coma por lo que no puede usarse con expresiones.
- *G_FREE(void *ptr)*: Libera la memoria asignada con G_MALLOC y apuntada por el puntero “ptr”.
- *LOCKDEC(lk)*: Declara una variable de tipo bloqueo con el nombre “lk”.
- *LOCKINIT(lock lk)*: Esta macro inicializa una variable declarada como tipo bloqueo, de tal manera que el primer proceso que la utilice llamando a LOCK(lk), pueda entrar en la sección crítica y el resto se queden a la espera.
- *LOCK(lock lk)*: Esta macro entra en una sección crítica protegida por la variable lk; de tal manera que si el proceso que la ejecuta es el primero, puede entrar sin problemas; pero si ya existe otro proceso dentro de la sección crítica, se queda a la espera.
- *UNLOCK(lock lk)*: Esta macro sale de una sección crítica controlada por la variable lk. Si hay algún proceso bloqueado por esta variable, uno de ellos es desbloqueado.
- *BARDEC(b)*: Esta macro declara una variable de tipo barrera.
- *BARINIT(barrier b)*: Esta macro inicializa una variable del tipo barrera. Esta operación sólo se necesita una vez al inicio de la aplicación.
- *BARRIER(barrier b, int n)*: Un proceso que ejecute esta macro, se queda bloqueado hasta que “n” procesos la ejecuten con el mismo identificador de barrera “b”; en ese momento todos los procesos son desbloqueados.
- *GET_PID()*: Esta macro devuelve el identificador numérico del proceso/hilo que lo ejecuta.

Estas macros se integran en el programa en C, si bien hay más macros dentro del conjunto PARMACS, sólo se han explicado las que se utilizan en esta aplicación. Como se ha podido observar, la comunicación entre los procesos es mediante *memoria compartida*.

El hecho de usar para las pruebas iniciales una implementación de PARMACS mediante hilos POSIX, ayudará a la aplicación a ser portada a RSIM con pocos o ningún cambio.

Ayuda en la depuración de errores

Para controlar y observar de manera detallada la ejecución de un programa, asi como para buscar errores que se nos pueden presentar inesperadamente, existen los denominados *debuggers*. Los debuggers son aplicaciones que preparan un entorno de ejecución controlado, donde poder observar, incluso paso a paso, cómo funciona nuestra aplicación.

Pero los debuggers no son del todo automáticos, hay que indicarles qué observar, y en algunos casos indicarles como observar ciertas partes de nuestro programa; por lo que puede ser muy tedioso estár informado en todo momento de muchos valores a lo largo de la ejecución del programa.

Uno de los métodos más simples para informar de lo que está sucediendo en cada punto importante de la aplicación, es imprimir por pantalla información detallada de los datos que está manejando la aplicación en un momento dado. Esto no es siempre deseable, ya que no siempre nos interesa la información de todos los subsistemas de la aplicación.

Para implementar la funcionalidad de salida por pantalla restringiendo dicha salida a uno o varios módulos concretos, se ha implementado un simple sistema de ayuda a la depuración de errores mediante macros del preprocesador de C. El sistema está definido en el fichero *debug.h*, y su funcionamiento es muy sencillo: Si a la hora de compilar un módulo, está definida la constante `DEBUG` dicho modulo se compilará con instrucciones que mostrarán datos detallados del estado interno en el momento de la ejecución.

Cada módulo, utiliza las siguientes primitivas:

- `DEBUGS(...)` Se redefine la funcion DEBUGS como *puts*, por lo que utiliza la misma declaración que puts. La funcionalidad de puts consiste en mostrar por pantalla una cadena de texto.
- `DEBUGF(...)` Se redefine la funcion DEBUGF como *printf*, por lo que utiliza la misma declaración que printf. Esta función permite imprimir una cadena de texto con un cierto formato donde intervienen uno o más parámetros.

Con estas dos macros, todos los módulos pueden ofrecer una salida por pantalla, y si se observa el código fuente se encontrará que son usadas de manera habitual. Hay que ser cuidadoso a la hora de activar la salida por pantalla en algunos módulos, ya que puede degradar en gran medida el rendimiento de la aplicación.

Otras tecnologías de la implementación

Otras tecnologías que se han empleado para la realización de la implementación del benchmark TPC-C:

- *Listas Enlazadas*: Una lista enlazada es una colección de elementos o nodos, en donde cada nodo contiene unos datos y un enlace al nodo siguiente.
- Preprocesador de macros *GNU m4*: para expandir las macros PARMACS en los ficheros de código fuente.(`http://www.gnu.org/software/m4/`)
- *GNU make*: para automatizar la traducción de código con macros PARMACS a código c, y la generación de un fichero ejecutable. (`http://www.gnu.org/software/make/`)
- *Doxygen*: para la estandarización y generación de documentación del código fuente. (`http://www.doxygen.org/`)

4.1.4. Organización del código fuente

El esquema de subsistemas descrito anteriormente se refleja en la organización del código fuente, ya que a no ser que cada subsistema fuera extremadamente sencillo, el no aplicarlo no ya en la organización de las funciones sino en el código fuente, haría del proyecto un "objeto" muy difícil de manejar.

- **Subsistema de almacenamiento**: que incluye los componentes
  - Componente: *Árbol B+*.
    - Interfaz: `arbolbmas.h`
    - Implementación: `arbolbmas.parmacs.c readwrite.parmacs.c`

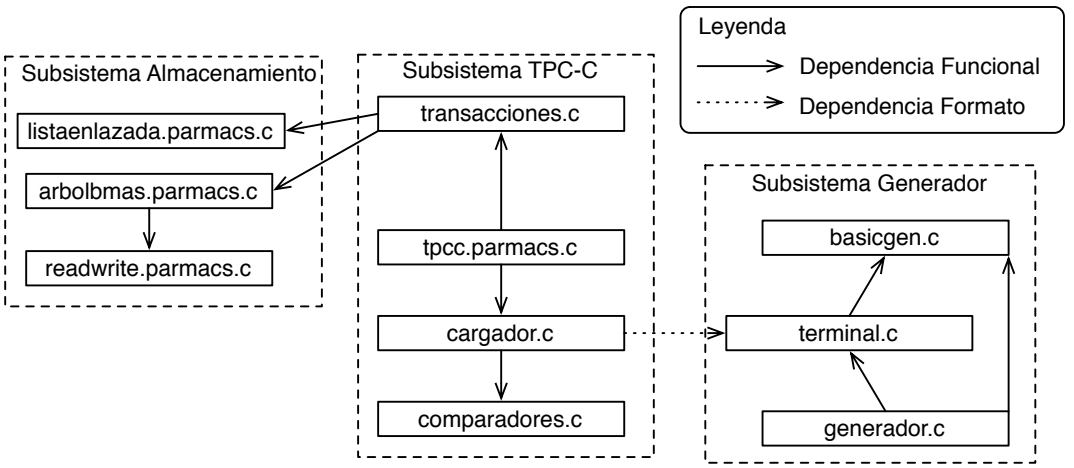

Figura 4.3: Diagrama de dependencias entre módulos.

  - Otras cabeceras: `arbolbmas-priv.h readwrite.parmacs.h`
- Componente: *Lista enlazada*
  - Interfaz: `listaenlazada.parmacs.h`
  - Implementación: `listaenlazada.parmacs.c`

- **Subsistema TPC-C**
  - Componente *Administrador*
    - Implementación: `tpcc.parmacs.c cargador.c comparadores.c`
    - Otras cabeceras: `registros.h cargador.h`
  - Componente *Servidor*
    - Interfaz transacciones.h
    - Implementación transacciones.c tpcc.parmacs.c
- **Subsistema Generador** sólo con un componente: *Generador*
  - Interfaz: `terminal.h`
  - Implementación: `generador.c terminal.c basicgen.c`

Las dependencias entre los diferentes módulos en los que está dividio el código fuente asi como el subsistema al que pertenecen, se pueden observar en la figura 4.3.

4.2. Subsistema de almacenamiento

4.2.1. Fundamentos de listas enlazadas

Una lista enlazada es una colección de elementos o nodos, en donde cada nodo contiene unos datos y un enlace al nodo siguiente. La lista enlazada es una de las estructuras dinámicas más simples

que implementa de manera versátil una lista dinámica de elementos donde continuamente se añaden o se borran elementos.

Un nodo de la lista enlazada se define en la figura 4.4

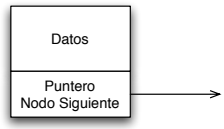

Figura 4.4: Nodo de una lista enlazada

De tal manera que cuando se enlazan varios nodos obtenemos la estructura de la figura 4.5

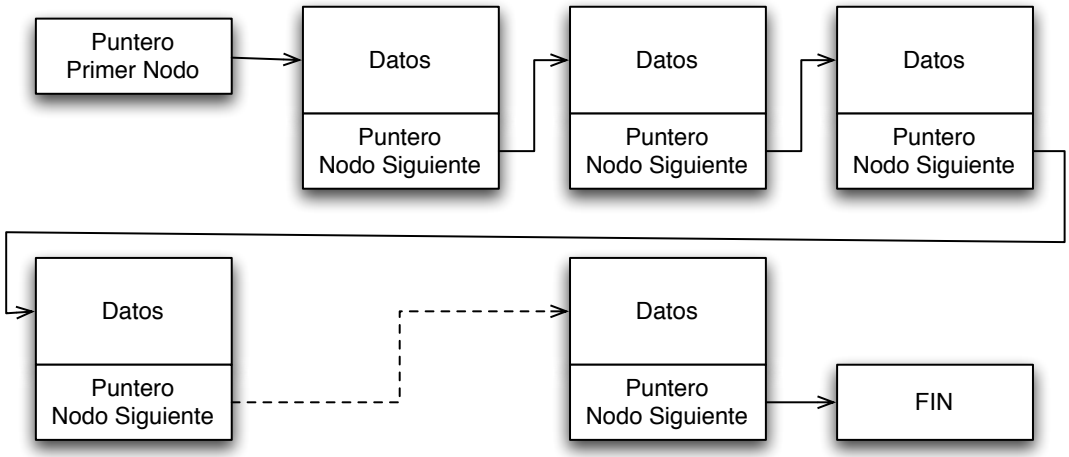

Figura 4.5: Estructura de lista enlazada

Dicha estructura no tiene porqué estar ordenada, aunque a veces encontraremos modificaciones como: que los datos tengan un orden en la lista, o exista un puntero al último nodo de la lista

Operación de búsqueda

La operación de búsqueda en las listas enlazadas es una de las más costosas, debido a que la búsqueda es secuencial y de orden n, es decir, hay que recorrer todos los nodos para encontrar el que buscamos; en el peor de los casos, esté ordenada o no, tengamos acceso desde el final o no, si buscamos un dato que no se encuentra en la lista acabaremos recorriendo todos los nodos.

El procedimiento es el siguiente:

1. Colocarse en el primer nodo de la lista.
2. Comprobar si los datos son los que buscamos.
3. Si lo son, hemos terminado.

4. Si no lo son, colocarse en el siguiente nodo; si el nodo no es final volver al segundo punto.
5. Si el nodo siguiente es el final, hemos terminado de manera no satisfactoria.

Operación de Inserción

Una de las ventajas de las listas enlazadas es que no hace falta desplazar los elementos cuando se insertan (ni cuando se borran). Para insertar un nuevo valor en la lista enlazada existen dos maneras: al principio de la lista, tras algún valor de la lista (lista ordenada).

Inserción al principio de la lista:

1. Obtener un nuevo nodo y almacenar el valor en la parte datos de dicho nodo.
2. En este nuevo nodos, en la parte del puntero al nodo siguiente, almacenar el valor de la variable *puntero al primer nodo*.
3. Actualizar la variable *puntero al primer nodo* con la dirección del nuevo nodo creado.

Inserción tras un valor de la lista:

1. Obtener un nuevo nodo y almacenar el valor en la parte datos de dicho nodo.
2. Buscar entre que dos nodos insertar, dichos nodos los llamaremos, anterior y siguiente.
3. Del nodo anterior, actualizar el valor de su puntero al nodo siguiente a la dirección del nuevo nodo.
4. En el nuevo nodo, actualizar el valor de su puntero al nodo siguiente a la dirección del nodo siguiente.
5. Si siguiente es el primer nodo, no hay anterior, y procederemos como cuando insertamos al principio de la lista.
6. Si anterior es el último nodo, no existe siguiente, por lo que en el nuevo nodo, su puntero al siguiente nodo apuntará a FIN.

Operación de borrado

En este caso también tenemos que considerar dos alternativas: borrar el primero, o borrar un nodo intermedio.

Borrar el primer nodo:

1. Hacer que la variable *puntero al primer nodo* apunte al nodo siguiente del nodo apuntado originalmente.
2. Destruir el nodo desenlazado.

Borrar un nodo intermedio:

1. Encontrar el nodo a borrar, en dicha búsqueda habrá dos nodos: el anterior al nodo que queremos borrar, y el siguiente al nodo que queremos borrar.
2. Actualizar el puntero al nodo siguiente del nodo anterior al que queremos borrar, y ponerle el valor de la dirección del nodo siguiente al que queremos borrar.

Proceso A	Proceso B
Obtiene el valor del puntero al último nodo	
	Obtiene el valor del puntero al último nodo
Accede al último nodo y actualiza su siguiente nodo al nuevo nodo de A	
	Accede al último nodo y actualiza su siguiente nodo al nuevo nodo de B
	Actualiza el puntero al último nodo con el nuevo nodo de B
Actualiza el puntero al último nodo con el nuevo nodo de A	

Cuadro 4.1: Ejemplo de cómo corromper una lista enlazada con dos procesos

3. Destruir el nodo desenlazado.

4. Si el nodo que queremos borrar es el primero, actuaremos como en el caso de borrar el primer nodo.

5. Si el nodo que queremos borrar es el último, haremos que en el nodo anterior al que queremos borrar, su puntero al nodo siguiente, apunte a FIN.

4.2.2. Sincronización de la lista enlazada

Cuando se habla de sincronizar se habla de que varios procesos o hilos de ejecución puedan interactuar a la vez y sin interferencias con la misma estructura de datos. El hecho de no controlar que muchos procesos accedan a la estructura, no ocasionaría problemas si todos leen la estructura, pero en el momento que uno o más procesos intenten modificarla, pueden existir problemas.

Por ejemplo, si dos procesos A y B, quieren añadir al final un nodo a través de un puntero de final de cola puede ocurrir que desaparezca uno de los nodos insertados.

Como se puede observar en el cuadro 4.1, nuestra estructura de datos ha acabado corrupta ya que, el penúltimo nodo apunta al nodo añadido por B, pero el puntero del último nodo apunta al nodo añadido por A; esto hace que la estructura quede inservible, y si se ejecutan más operaciones sobre la misma el resultado puede ser inesperado y no el deseado.

Modelo de sincronización

En el benchmark TPC-C la única tabla que utiliza la lista enlazada es la de *histórico*, y no hace más que añadir nodos al final a través de un puntero que apunta al último nodo. Para evitar problemas como los anteriores se ha puesto un cerrojo a las operaciones sobre la lista de tal manera que: no puede ejecutarse ninguna operación sobre la lista si una anterior no ha terminado. Al código que se ejecuta entre que se adquiere y se libera un bloqueo de este tipo se le llama *sección crítica*

En este caso el ejemplo anterior quedaría como se puede observar en el cuadro 4.2.

Como se puede ver, ahora B accede al puntero del último nodo real, y lo que se ha conseguido con este cerrojo es secuenciar las operaciones sobre la lista enlazada.

Proceso A	Proceso B
Obtiene el cerrojo de la lista enlazada	
Obtiene el valor del puntero al último nodo	
	Intenta obtener el cerrojo de la lista enlazada, pero como ya está asignado se queda a la espera.
Accede al último nodo y actualiza su siguiente nodo al nuevo nodo de A	
Actualiza el puntero al último nodo con el nuevo nodo de A	
Libera el cerrojo	
	Despierta del bloqueo ya que A lo ha liberado, y obtiene el cerrojo
	Accede al último nodo y actualiza su siguiente nodo al nuevo nodo de B
	Actualiza el puntero al último nodo con el nuevo nodo de B
	Libera el cerrojo

Cuadro 4.2: Ejemplo de inserción ordenada con dos procesos

4.2.3. Implementación de la lista enlazada

Esta lista enlazada implementa inserción por el final y borrado por el principio, aunque en el benchmark TPC-c no se usa el borrado por el principio. También incluye registros genéricos de tamaño N, que se explicarán más adelante.

Estructuras y tipos de datos

Las estructuras y tipos de datos utilizadas, están declaradas en *listaenlazada.parmacs.h*.

▷ struct struct_LENodo

Estructura para manejar los enlaces al nodo siguiente. Esto es lo que se añadirá a los registros genéricos que se quieran insertar en la lista. Contenido:

- `struct struct_LENodo *siguiente;` Puntero al siguiente elemento de la lista. Con valor NULL cuando no hay más nodos en la lista.

▷ struct struct_ListaEnlazada

Almacena la información necesaria para manejar la lista. Contenido:

- `struct struct_LENodo *principio;` Puntero al primer nodo de la lista, o NULL cuando está vacía.
- `struct struct_LENodo *fin;` Puntero al último nodo de la lista, o NULL cuando está vacía.
- `int regsize;` Tamaño del registro genérico usado

- `LOCKDEC(bloqueo);` Cerrojo de la lista.

▷ **LENodo**

Se define el tipo de dato LENodo de la siguiente manera:
`typedef struct struct_LENodo LENodo;`

▷ **ListaEnlazada**

Se define el tipo de dato ListaEnlazada de la siguiente manera:
`typedef struct struct_ListaEnlazada ListaEnlazada;`

Interfaz de uso

Para utilizar la implementación de la lista enlazada, se han dispuesto las siguientes funciones declaradas en `listaenlazada.parmacs.h`.

▷ **le_nueva**

- Declaración: `ListaEnlazada *le_nueva(int regsize)`
- Descripción: Inicializa un identificador de lista enlazada.
  - Se inicializan los punteros a null
  - Se almacena el tamaño del registro.
  - Se inicializa el cerrojo.
- Parámetros:
  - `int regsize` (entrada): Tamaño del registro de datos genérico con el que trabajar.
- Devuelve: un puntero a la estructura de lista enlazada, que estará inicializada como vacía.

▷ **le_insertar_final**

- Declaración: `void le_insertar_final(ListaEnlazada *lista,void *registro)`
- Descripción: (función sincronizada) Inserta al final de la lista.
  - Reserva memoria para el registro y la estructura de nodo.
  - Actualiza los punteros para colocar el nodo.
  - Copia los datos del nodo.
- Parámetros:
  - `ListaEnlazada *lista` (entrada): Lista enlazada sobre la que operar.
  - `void *registro` (entrada): Puntero al registro que insertar en la lista.

▷ **le_borrar_primero**

- Declaración: `int le_borrar_primero(ListaEnlazada *lista, void *registro)`
- Descripción: (función sincronizada) Borra el primer elemento de la lista y lo devuelve.
  - Primero comprueba si la lista está vacía.
  - En caso de no estarlo, copia los datos al registro.
  - Y luego actualiza los punteros internos.
- Parámetros:
  - `ListaEnlazada *lista` (entrada): Lista enlazada sobre la que operar.
  - `void *registro` (salida): Lugar donde depositar el registro borrado.
- Devuelve: Un entero indicando el resultado de la operación.
  - `OP_OK` Operación Correcta.
  - `OP_NODATA` Sin datos debido a que la lista estaba vacía.

▷ **le_destruir**

- Declaración: `void le_destruir(ListaEnlazada *lista)`
- Descripción: Va borrando todos los nodos uno a uno.
- Parámetros:
  - `ListaEnlazada *lista` (entrada): Lista enlazada sobre la que operar.

Peculiaridades

La primera peculiaridad en esta lista y que se aplicará también en el árbol B+, es que se habla de *registros genéricos*. Un registro genérico es aquel que puede albergar cualquier tipo de dato, por tanto la lista no tiene en cuenta el contenido de los datos para su funcionamiento interno.

Si bien se ha definido como un nodo de la lista la estructura struct_LENodo, que sólo tiene un puntero al nodo siguiente. A la hora de reservar memoria se reserva espacio para

- El puntero al nodo siguiente (lo que es la estructura LENodo).
- Los datos (el tamaño está indicado por regsize).

Por lo que tendremos una asignación como la siguiente (extraída de la función le_insertar_final): `newrecord=G_MALLOC(lista->regsize+sizeof(LENodo));`

A la hora de obtener los datos de un registro cualquiera, no tenemos más que, dado el puntero del registro, desplazarnos el tamaño del puntero y leer a partir de ahí: `memcpy(registro,tmp+1,lista->regsize);`

El segundo punto interesante, y que también se aplicará en el árbol B+, es que a la hora de añadir datos, aunque se le pase una zona de memoria con los datos, la lista enlazada reserva una zona de memoria compartida para guardar ese dato, con lo que queda accesible por los demás procesos.

4.2.4. Fundamentos de árboles B+

Ya se ha hablado (pag. 85) sobre la estructura de un árbol B+; si decimos que un árbol B+ es de orden p decimos que:

- Si el nodo es interno tiene $p - 1$ claves y p enlaces.
- Si el nodo es hoja, tiene $p - 1$ claves y enlaces.

Los nodos de un árbol B+ siguen la estructura de la figura 4.6

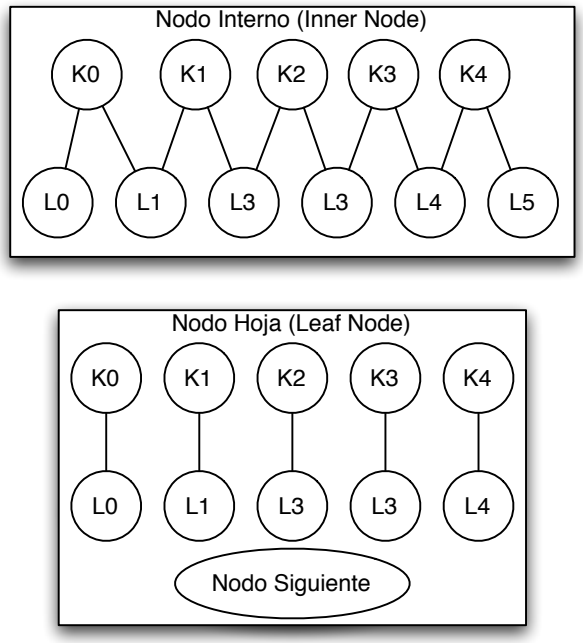

Figura 4.6: Nodos de un árbol B+

La estructura que sirva para manejar un árbol B+ al menos tendrá un puntero al nodo raíz, y un puntero al primer nodo hoja, ya que los nodos hoja están unidos formando una lista enlazada, que es de gran utilizad cuando se realizan búsquedas no indexadas.

Aunque se han estudiado muchas mejoras [4], [22], sobre todo en lo referente a la relación: eficiencia de caché respecto al acceso a los nodos, en este análisis y en esta implementación sólo se utilizan los fundamentos básicos de arboles B+.

Búsqueda indexada

La búsqueda a través de un campo clave es el punto fuerte de los arboles B+, entre millones de entradas se puede acceder a aquella que andamos buscando en cuestión de milésimas de segundo. Para entender las razones de esta afirmación, se va explicar un ejemplo.

Supongamos que tenemos un árbol B+ de orden 51, es decir, aquel árbol cuyos nodos internos tienen 51 enlaces y 50 claves, y cuyos nodos hoja tienen 50 enlaces y 50 claves. Llamaremos altura

del árbol, a los nodos que hay que recorrer desde el raíz, hasta llegar a un nodo hoja, ambos inclusive. La cantidad de datos que podemos almacenar es la siguiente:

- Altura 1: 50 registros. Sólo existe un nodo, el raíz que es a su vez nodo hoja.
- Altura 2: $51 * 50 = 2.550$ registros. Del nodo raíz parten 51 enlaces a nodos hoja que a su vez pueden almacenar 50 registros.
- Altura 3: $51 * 51 * 50 = 130.050$ registros. Del nodo raíz parten 51 enlaces a nodos internos, que tienen 51 enlaces a nodos hoja; estos últimos albergan 50 enlaces a registros.
- Altura 4: $51 * 51 * 51 * 50 = 6.632.550$ registros. Ya tenemos varios millones de registros con solo 4 alturas.

Nos quedaremos con el árbol B+ de orden 51 y de 4 alturas, se supone que está totalmente lleno y que por lo tanto podremos acceder a los más de 6 millones de registros. La operación de búsqueda a partir de una clave K es la siguiente:

1. Se accede al nodo raíz y se marca como nodo actual.
2. Dado que las claves del nodo actual están ordenadas, se busca la clave K entre todas las claves del nodo actual mediante una *búsqueda binaria*, si no se encuentra, se dice la posición aproximada, es decir, entre que dos claves se encuentra.
3. Entre la clave K_{i-1} y la clave K_i se encuentra el enlace L_i que nos lleva al nodo cuyas claves son: mayores que K_{i-1} y menores o iguales que K_i. (fig. 4.7)

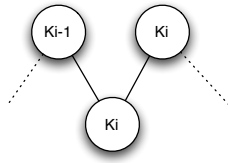

Figura 4.7: Estructura de un nodo interno

4. La posición aproximada es aquella i tal que: $K_{i-1} < K \leq K_i$.
5. Accedemos al nodo apuntado por la posición i (L_i), y lo llamamos nodo actual.
6. Si el nodo es hoja, repetimos desde el punto dos y accederemos al dato.
7. Si el nodo es interno, volvemos al punto 2.

Para recorrer los 4 niveles del nodo se han hecho 4 búsquedas binarias entre 50 elementos, que obviamente son mucho menos costosas que una búsqueda binaria entre 6 millones de elementos. De ahí que acceder a un registro entre varios millones sólo conlleve unos pocos milisegundos.

Búsqueda no indexada

Este es el punto débil de los árboles B+, si no se posee un campo clave por el cual buscar un nodo en el árbol, hay que recorrer todos los nodos. Aun así, el árbol B+ presenta cierta facilidad para realizar esa tarea; debido a que todos sus nodos hoja están enlazados entre si, y solo los nodos hoja contienen enlaces a los datos, obtenemos una *lista simplemente enlazada* con los nodos hoja, que podremos recorrer para encontrar el dato buscado.

Utilizando el ejemplo anterior de un árbol B+ de orden 51 con 4 niveles, y suponiendo el peor caso, y es que el nodo buscado esté al final de la lista enlazada. Dado que no se está buscando por clave y por lo tanto no se puede utilizar la búsqueda binaria en cada nodo, se tendrían que realizar más de 6 millones de comparaciones antes de llegar al nodo final.

El algoritmo de búsqueda sería el siguiente:

1. Se accede al primer nodo hoja y se le nombra: nodo actual.
2. Se busca uno a uno entre los registros apuntados por el nodo actual el dato que buscamos.
3. Si se encuentra, se finaliza la búsqueda.
4. Si no se encuentra, se utiliza el puntero al nodo siguiente, y hacemos del nodo siguiente el nodo actual; por último volvemos al punto 2.

Inserción

La inserción de un registro extra en un árbol B+ puede ser tan costosa como muy simple; no es que sea una de las operaciones más peligrosas a la hora de manejar esta estructura pero sin conviene hacerse metódicamente para evitar errores.

Toda inserción en un árbol B+ comienza con una búsqueda del nodo hoja donde debiera encontrarse la clave del registro que queremos insertar. Si localizado el nodo hoja, este tiene espacio suficiente para albergar el nuevo registro, el algoritmo es el siguiente.

1. Lo primero es realizar una búsqueda binaria para conocer en que posición debiera insertarse el registro actual. Utilizando la función de búsqueda binaria aproximada que se utilizó en la búsqueda indexada, obtenemos dicha posición.
2. Si el nodo admite claves duplicadas y la clave existe, continuamos normalmente; pero si no las admite y la clave existe, se anula la inserción.
3. Si la posición de inserción es i en el nodo hoja, desplazamos todos los pares de: $(K_j, Lregistro_j)$ una posición a la derecha desde $j = ocupados$ (que simboliza la cantidad actual de enlaces ocupados en el nodo) hasta $j = i$, lo hacemos al revés (de derecha a izquierda) para que la operación $K_{j+1} = K_j, Lregistro_{j+1} = Lregistro_j$ no sobreescriba datos existentes.
4. Por último, insertamos nuestro nuevo registro copiando la clave en K_i y actualizando el puntero al registro $Lregistro_i = NuevoRegistro$.

Es una operación más complicada cuando el nodo hoja está lleno, en ese caso hay que dividir el nodo hoja, para obtener dos nodos hoja llenos hasta la mitad, y así poder insertar en uno de los dos nuevos nodos creados. La división de un nodo hoja se realiza de esta manera:

1. Dado un nodo hoja lleno, se busca *la mitad* de dicho nodo.

2. La mitad de dicho nodo se indica como i. Se crea un nuevo nodo vacío.

3. Se mueven los datos: de i hasta el final del nodo al nuevo nodo creado. Tenemos dos nodos: *izquierdo* (de 0 a $i-1$) y *derecho* (de i hasta el final).

4. Al tener un nuevo nodo hoja, hace falta enlazarlo con el nodo anterior, que es un nodo interno. Recordemos que los nodos internos tienen claves para indicar que camino tomar, así que al añadir un nuevo enlace en el nodo superior, necesitaremos una nueva clave para guiar las búsquedas.

5. Dada una clave K_j todos aquellos valores menores o iguales se encuentran en el enlace L_j y los mayores en el enlace L_{j+1}

6. Utilizamos la última clave del nodo izquierdo como clave de unión.

Se ha dejado el algoritmo a la mitad para resumir los pasos hechos hasta el momento: cuando un nodo hoja esta lleno, se divide por la mitad en dos nodos: izquierdo y derecho, siendo la ultima clave del izquierdo la clave de unión. Ahora hay que introducir dicha clave en el nodo superior, que es un nodo interno.

1. Nos posicionamos en el nodo interno superior al nodo hoja, y buscamos la posición i por la que se accede al nodo hoja.

2. Como hemos dividido anteriormente, desde esa posición i sólo se accede a la mitad de los valores por lo que, idealmente, debiéramos colocar la clave de unión como K_i ya que en el nodo izquierdo están los valores menores o iguales que K_i.

3. Para poder realizar esto movemos todas las claves desde K_ultima a K_i un lugar a la derecha: $K_{j+1} = K_j$.

4. Realizamos lo mismo con los enlaces pero desde L_ultimo a L_{i+1}.

5. Al realizar este desplazamiento de enlaces dejamos un hueco para insertar un enlace al nodo derecho (fig. 4.8).

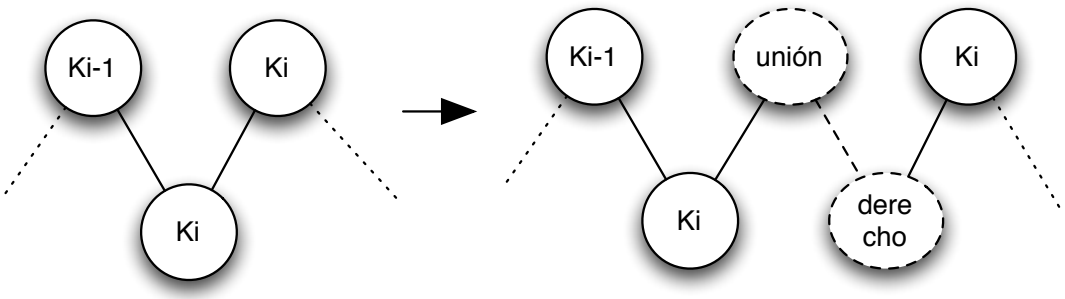

Figura 4.8: Inserción de una clave de unión en un nodo interno

Una vez hecho esto, no habíamos insertado aun el registro en el nodo hoja, y al dividir el nodo hoja, sólo nos queda seleccionar si va en el izquierdo o en el derecho y realizar una inserción normal en el nodo hoja adecuado. No olvidar completar la lista enlazada, el nodo izquierdo apunta al nuevo nodo derecho, y el nuevo nodo derecho apunta donde antes apuntaba el viejo nodo izquierdo, es decir, una inserción entre medias de una lista simplemente enlazada.

Por último puede ocurrir un problema, y es que al ir a insertar en el nodo interno superior, este esté lleno. La técnica a aplicar es la misma: dividir el nodo interno, obtener una clave de unión e insertar la clave de unión en el nodo interno superior; pero hay que tener cuidado con ciertos detalles.

1. Los nodos internos tienen distinto número de claves que de enlaces, y su disposición es como la de *varios triángulos juntos*, siendo el vértice superior la clave y los dos inferiores los enlaces. A la hora de dividir un nodo interno por la mitad, el triángulo se dividirá en dos, y la clave del vértice superior desaparecerá del nodo y pasará a ser la clave de unión.

2. Si dividimos el nodo raíz, hay que insertar un nodo interno nuevo y con sólo dos enlaces y una clave: la clave de unión, y los enlaces a las dos partes del nodo dividido.

Para la tarea de inserción, muchas veces es útil tener punteros de navegación en los nodos: nodo anterior, nodo siguiente, etc. Estos punteros pueden estar en el propio nodo, o los podemos ir almacenando según vamos descendiendo en el árbol.

Borrado

Al igual que con la inserción el borrado es una operación que puede ser sencilla simplemente borrando una entrada en un nodo hoja; o bastante compleja, necesitando borrar varias entradas en varios nodos internos. Se va a explicar igual que se indicó la inserción, comenzando por un borrado simple en un nodo hoja; el algoritmo es el siguiente:

1. Dada una clave K, buscar el nodo hoja que debiera contener la clave K.

2. Si no se encuentra, hemos terminado.

3. Si la encontramos, estará en la posición i del nodo hoja.

4. Desplazar los pares de clave y enlace (K_j, L_j) desde $i + 1$ hasta $ultimo$, una unidad a la izquierda: $K_j = K_{j+1}$

5. Liberar la memoria del registro asociado a la vieja clave K_i.

En este momento puede ocurrir que el nodo rebase el límite de tamaño mínimo. Si bien se había dicho que un nodo como mínimo puede tener $(maximo/2)$ claves; es muy normal experimentar con dicho límite para optimizar el funcionamiento del árbol en un entorno concreto.

Se supondrá que el nodo ha rebasado su límite de tamaño mínimo y hay que hacer *algo* para solucionarlo. Ese algo consiste en dos soluciones:

- *Balancear*: Dados dos nodos, se balancea su contenido entre los dos, terminando ambos nodos con el mismo número de claves y enlaces ocupados.

- *Fusionar*: Dados dos nodos, se fusiona uno con otro, desapareciendo el primero, y pasando su contenido a formar parte del segundo.

Vecino izquierdo	Vecino derecho	Acción
No esta lleno	No Hay	Fusión del actual con el izquierdo
Lleno	No Hay	Balancear el actual con el izquierdo
No Hay	No esta lleno	Fusión del actual con el derecho
No Hay	Lleno	Balancear el actual con el izquierdo
Otro padre distinto al del nodo actual	Mismo padre Y puede admitir	Fusión del actual con el derecho
Otro padre distinto al del nodo actual	Mismo padre Y esta lleno	Balancear el actual con el derecho
Mismo padre Y puede admitir	Otro padre distinto al del nodo actual	Fusión del actual con el izquierdo
Mismo padre Y esta lleno	Otro padre distinto al del nodo actual	Balancear el actual con el izquierdo
Mismo padre Y puede admitir	Mismo padre Y puede admitir	Fusión del actual con el derecho (podría ser izquierdo también)
Mismo padre Y lleno	Mismo padre Y puede admitir	Balancear el actual y el izquierdo
Mismo padre Y puede admitir	Mismo padre Y lleno	Balancear el actual con el derecho

Cuadro 4.3: Resumen de acciones de borrado

En este caso hay que tener *anotados* los nodos izquierdo y derecho del nodo hoja donde nos encontramos, así como si el padre de dichos nodos es el mismo que el del nodo actual. Hay que tener esto en cuenta ya que a la hora de balancear dos nodos, o de fusionar dos nodos, sólo se balancean o fusionan aquellos nodos que tengan el mismo padre, para así solo alterar un enlace y una clave de dicho padre.

Las situaciones con las que nos podemos encontrar y la acción a tomar son las indicadas en el cuadro 4.3.

El algoritmo de fusión es el siguiente:

1. Dados dos nodos izquierdo y derecho, se fusiona siempre hacia la izquierda.
2. Se mueven todos los datos del nodo derecho al izquierdo.
3. Se elimina el nodo izquierdo.
4. Se anota que han ocurrido cambios para más tarde borrar el enlace del nodo superior al nodo izquierdo que ya no existe.

Y para el balanceo:

1. Dados dos nodos: izquierdo y derecho, se comprueba la ocupación de cada uno.
2. La suma de los dos tamaños entre 2 es el nuevo tamaño de ambos nodos.

3. La cantidad a desplazar es el tamaño del nodo más grande menos el tamaño nuevo.
4. Si el izquierdo es más grande que el derecho, se mueven nodos y enlaces de izquierda a derecha.
5. Si el derecho es más grande, se mueven nodos y enlaces del nodo derecho al izquierdo.

El algoritmo de borrado completo es el siguiente:

1. Continuando con el caso anterior se ha buscado y borrado un elemento de un nodo hoja.
2. Después de borrar hay *pocos* elementos.
3. Se selecciona una acción del cuadro 4.3
4. Si la acción ha sido balancear, se termina.
5. Si la acción ha sido fusionar, hay un nodo menos y hay que quitar un enlace del nodo superior.
6. Volver a aplicar el algoritmo de borrado completo en el nodo superior.

Se puede observar que este algoritmo es recursivo a la hora de descender, ya que se anotan: nodos izquierdos y derechos así como sus padres. Y luego se retrocede ese camino recursivo borrando entradas de los nodos si han sucedido fusiones de nodos, ya que en este caso hay que eliminar un enlace.

La fusión se produce siempre hacia la izquierda, lo que garantiza que el árbol siempre estará equilibrado: se irán borrando nodos hoja a la vez que se ajusta el contenido de los nodos superiores; pero esto hace también difícil que el árbol reduzca su nivel. Por ejemplo, en el árbol de orden 51 con 4 niveles, hace falta dejar al nodo raíz con sólo un enlace para que desaparezca (y así eliminar un nivel), esto supone eliminar $50 * 51 * 51 * 50 = 6.502.500$ registros, para que el árbol tenga ahora $51 * 51 * 50 = 130.050$ registros.

4.2.5. Sincronización del árbol B+

Dado que los árboles B+ se utilizan en sistemas de bases de datos como índices, se han analizado diferentes formas [11] de mejorar la sincronización y por lo tanto el bloqueo de accesos, a las estructuras de un Árbol B+. El problema del acceso concurrente por parte de varios procesos o hilos de ejecución a la estructura del árbol, ya no es dos simples enlaces que pueden corromperse, sino millones de enlaces donde el hecho de que uno no indique el nodo correcto puede suponer perder la mitad de la información almacenada.

Primera aproximación

Una aproximación a las técnicas de bloqueo pasa por analizar que operaciones deseamos bloquear. En un principio y pensando que un proceso pueda estar leyendo el área que otro proceso está alterando, la primera solución que se sugiere es: *poner un cerrojo por árbol*, de tal manera que las operaciones se realizan de manera secuencial y no ocurre una siguiente antes de que termine la anterior.

Esta solución evita todos los problemas pero es muy poco eficiente, debido a que:

- Es muy habitual que varios procesos lean a la vez el árbol. Esto no supone en la realidad ningún problema pero con esta aproximación no es posible, por lo que cada lector tiene que esperar a que el anterior termine.

- Se tiene como ejemplo el árbol anterior de orden 51 y 4 niveles, tenemos más de 6 millones de registros. Si además se tienen 16 procesos accediendo al árbol simultáneamente, con 6 millones de registros, la probabilidad de que de un nodo hoja se borre a la vez que se lea es muy baja. Suponiendo que no hay que tocar el árbol nada más que para borrar la entrada de un nodo hoja sin fusionar o balancear:
  - Si bloqueamos el árbol entero para una operación de lectura, podríamos estar realizando la operación de borrado sin problemas.
  - Si bloqueamos el árbol entero para escritura y sabemos que no vamos a tener que fusionar ni balancear, podríamos estar haciendo muchas otras lecturas que no hacemos.
- Esto último se aplica también a las inserciones si no hay que dividir el nodo donde insertamos.

Analizando estos casos, se puede ver un problema bien conocido: existe una estructura de datos en memoria común a varios procesos, unos procesos quieren leer y otros escribir. Es el problema de los *lectores escritores*

Lectores Escritores

La idea básica del problema de los lectores escritores es la siguiente: basándose en una estructura común:

- Varios procesos quieren leer y varios escribir.
- Pueden existir varios lectores a la vez en esa estructura común.
- Sólo puede existir un escritor operando en dicha estructura. Aunque muchos lectores y escritores pueden estar esperando para realizar la operación, sólo un escritor puede operar en el arbol, y cuando dicho escritor trabaja, ningún otro lector puede operar.

Soluciones se pueden encontrar en los libros de sistemas operativos [15], [14]. Muchas de estas soluciones están *desequilibradas*, es decir, dan preferencia a los lectores o a los escritores. La preferencia se asigna de tal manera que si hay preferencia a los escritores, cuando llega un escritor, se salta la cola de espera de los lectores y es el siguiente que entra a operar; o al revés, la preferencia de los lectores, indica que un lector cuando se pone a la espera para operar en el árbol siempre se salta la cola de los escritores.

Esto puede provocar *inanición*: que un escritor no llegue a escribir nunca o que un lector no llegue a leer nunca, debido a que aunque están en la cola a la espera de operar, siempre hay otra operación que se salta dicha cola y les mantiene indefinidamente a la espera.

Para evitar dicha inanición se ha utilizado un algoritmo lo más equilibrado posible (cuadro 4.4). Se compone de:

- *num_lectores*: Una variable que controla el número de lectores.
- *seccion_critica*: Un cerrojo para acceder a dicha variable.
- *cola*: Un cerrojo que actúa como una cola donde se apuntan tanto lectores como escritores.
- Un cerrojo para los escritores.

La nomenclatura de las diferentes acciones de la solución al problema lector escritor son:

Lector	Escritor

`LOCK(cola)`
`LOCK(seccion_critica)`
`num_lectores=num_lectores+1`
`if (num_lectores == 1)`
`  LOCK(bloqueo_escritores)`
`UNLOCK(seccion_critica)`
`UNLOCK(cola)`
	`LOCK(cola)`
	`LOCK(bloqueo_escritores)`
	`UNLOCK(cola)`

Realizar la lectura correspondiente.	Realizar la escritura

`LOCK(seccion_critica)`
`num_lectores=num_lectores-1`
`if (num_lectores == 0)`
`  UNLOCK(bloqueo_escritores)`
`UNLOCK(seccion_critica)`
	`UNLOCK(bloqueo_escritores)`

Cuadro 4.4: Algoritmo de lectores escritores equilibrados usando cerrojos

- *Bloquear para leer*: ejecutar la primera parte del lector, hasta justo antes de realizar la lectura correspondiente. Simboliza que se quiere realizar una lectura.
- *Desbloquear lectura*: ejecutar la segunda parte del lector, justo después de la lectura correspondiente. Simboliza que la lectura ha finalizado.
- *Bloquear para escribir*: ejecutar la primera parte del escritor, hasta justo antes de realizar la escritura correspondiente. Como antes, simboliza que se quiere realizar una escritura.
- *Desbloquear escritura*: ejecutar la segunda parte del escritor (un simple desbloqueo), justo después de la escritura correspondiente. Simboliza que la escritura ha finalizado.

Algunos casos con los que nos podemos encontrar son los siguientes, que además ayudarán a entender el funcionamiento de esta solución al problema lector escritor.

- **Sólo lectores**: en este caso, como pueden existir múltiples lectores accediendo de manera concurrente a los datos, cada lector aumenta en uno la variable de número de lectores; además el primer lector impide a los escritores entrar bloqueando el cerrojo de los escritores. Al acabar su lectura decrementan en uno el contador de lectores y el último lector que sale, abre el cerrojo de los escritores.
- **Aparece un escritor**:
  1. En un momento dado, aparece un escritor, junto con el resto de lectores se pone en la cola, pero mientras que los lectores salen de la cola directamente, el escritor se queda bloqueado en su cerrojo de escritores ya que, continuando con el punto anterior, hay muchos lectores accediendo.
  2. Como no ha desbloqueado el cerrojo de cola, en el preciso instante que se queda bloqueado en su cerrojo de escritor, todas las nuevas peticiones, ya sean de lectores o escritores se bloquean en el cerrojo de cola.
  3. Con el tiempo, los lectores que hay dentro terminarán su lectura, e irán decrementando el contador.

4. Cuando el último lector salga, desbloqueará el cerrojo de los escritores, y el lector liberará la cola y realizará su escritura. En este momento puede intentar acceder un lector o un escritor.
5. Cuando el escritor termine, abrirá el cerrojo de escritura.

- **Justo después del escritor, aparece un lector**

1. Cuando el escritor desbloquea la cola, entra un lector.
2. Ese lector es el primero, ya que recordemos que para que el escritor entrase no tenia que haber ningún lector.
3. El lector al ser el primero, intenta bloquear el cerrojo de escritores, pero ya esta bloqueado dado que hay un lector dentro, y se queda a la espera.
4. Como se queda a la espera, el cerrojo de cola no se desbloquea y no pueden entrar más operaciones.
5. Cuando el escritor termine abriendo dicho cerrojo de cola, el lector que había permanecido a la espera en dicho cerrojo se desbloquea dejando dicho cerrojo bloqueado para nuevos escritores.

- **Justo después del escritor, aparece otro escritor**

1. Ahora justo después del escritor, en vez de un lector, aparece otro escritor.
2. Pasa del cerrojo de la cola al cerrojo de escritores.
3. Como el cerrojo de escritores estaba bloqueado debido a que ya hay un escritor dentro, se queda a la espera de que el escritor termine la operación.
4. En dicha espera no libera el cerrojo de cola, por lo que las nuevas operaciones se ponen a la espera en dicha cola.
5. Cuando el escritor inicial termina, desbloquea el cerrojo de escritores, y el nuevo escritor comienza su operación.
6. Si justo después aparece un lector, es el caso anterior; y si aparece un escritor, es este mismo caso.

Solución aplicada al árbol B+ implementado

En el árbol que se ha implementado se ha aplicado el problema de los lectores escritores en 2 niveles:

- *A nivel de árbol*: se puede bloquear el árbol para lectura o para escritura.
- *A nivel de nodo hoja*: un nodo hoja se puede bloquear para lectura o para escritura.

La idea de aplicar esta división en dos niveles es que muchas veces solamente se necesita bloquear el nodo hoja para añadir o borrar un dato, dejando el árbol intacto; por lo que no hay que bloquear el árbol entero (millones de registros), si solo necesitamos bloquear un nodo. Si a la hora de insertar o borrar, se detecta que va a hacer falta modificar más de un nodo hoja, se desiste y se bloquea el árbol para escritura, volviendo a empezar la operación de inserción.

Veamos como se aplica esta solución a cada operación:

◇ **Operación de búsqueda indexada**

Algoritmo empleado para la búsqueda indexada:

1. Primero se bloquea el árbol para lectura.
2. Se desciende por el árbol buscando el nodo hoja correspondiente.
3. Una vez obtenemos el enlace del nodo hoja, bloqueamos dicho nodo para lectura.
4. Entre medias, dicho nodo no ha podido desaparecer ya que el árbol estaba bloqueado para lectura y su estructura no ha sido modificada, así que al menos nos aseguramos el acceso al nodo.
5. Buscamos dentro del nodo y finalizamos la operación de búsqueda de manera satisfactoria o no, dependiendo si hemos encontrado o no lo que buscábamos.
6. Desbloqueamos la lectura del nodo hoja.
7. Desbloqueamos la lectura del árbol.

◇ **Operación de búsqueda no-indexada**

Algoritmo empleado para la búsqueda no indexada:

1. Se bloquea el árbol para lectura
2. Se obtiene el puntero al primer nodo hoja, que es el nodo actual.
3. Se bloquea el nodo actual para lectura.
4. Se busca en el nodo actual. Si se encuentra lo que hemos buscado, desbloqueamos el nodo y el árbol para lectura y devolvemos el dato. Si no se encuentra, continuamos.
5. Se obtiene el puntero al nodo siguiente, y se le hace nodo actual.
6. Se desbloquea la lectura el nodo anterior.
7. Volvemos al punto 2: bloquear el nodo actual.
8. Si se llega al final de la lista de nodos hoja sin encontrar el dato deseado, se desbloquea el último nodo y el árbol de lectura y se finaliza.

◇ **Operación de modificación**

Algoritmo empleado para la modificación de los datos de un registro:

1. Se bloquea el árbol para lectura.
2. Se busca el nodo hoja que contiene el registro a modificar.
3. Se bloquea el nodo hoja para escritura.
4. Se busca el dato en el nodo hoja. Si no se encuentra, se desbloquea el nodo hoja de escritura y el árbol de lectura, y se termina la operación.
5. Si se encuentra, se modifica.
6. Se desbloquea la escritura del nodo hoja, y la lectura del árbol

◇ Operación de inserción

El algoritmo para la inserción confía en encontrarse con el mejor caso: una inserción directa sin modificaciones de la estructura; pero si detecta que va a tener que modificar dicha estructura, aborta la operación y bloquea el árbol para escritura para asegurarse el poder modificar la operación. Eso no quita que entre que desbloquea el árbol de lectura y lo bloquea para escritura los nodos cambien de contenido y la operación pudiera realizarse sin necesidad de este bloqueo; pero aun en este caso, se opta por bloquear todo el árbol. La descripción del algoritmo es la siguiente:

1. Se bloquea el árbol para lectura.
2. Se busca el nodo hoja donde insertar.
3. Se bloquea el nodo hoja para escritura.
4. Comprobamos si podemos insertar:
   a) Si la clave existe, no permitimos claves duplicadas, abortamos la operación desbloqueando el nodo hoja de escritura y la lectura del árbol.
   b) Si el nodo está lleno, abortamos la operación, desbloqueamos la escritura del nodo y la lectura del árbol. Y bloqueamos para escritura el árbol entero; en este caso no hay que hacer ninguna comprobación ya que *será la única operación que actúe en el árbol*.
5. Insertamos el dato
6. Desbloqueamos la escritura del nodo hoja y la lectura del árbol.

◇ Operación de borrado

La operación de borrado, de manera muy similar a la operación de inserción, confía en que el borrado no afecte a la estructura del árbol. Antes de borrar comprueba si el nivel de ocupación del nodo bajaría por debajo del límite, y en caso afirmativo aborta la operación y bloquea el árbol para escritura, reanudándola con la confianza de ser la única operación trabajando en el árbol. El algoritmo empleado para el borrado es el siguiente:

1. Se bloquea el árbol para lectura.
2. Se busca el nodo hoja con el registro a borrar.
3. Se bloquea el nodo hoja para escritura.
4. Se comprueba si es posible el borrado:
   a) Si la clave no existe, se aborta la operación ya que no hay nada que borrar, y se desbloquea la escritura del nodo y la escritura del árbol.
   b) Si al borrar un elemento del nodo hoja, la capacidad del nodo está por debajo del límite mínimo de ocupación de un nodo, se aborta la operación desbloqueando la escritura del nodo y la lectura del árbol. Más tarde se bloquea el árbol para escritura y se repite, teniendo la certeza de que es la única operación que se esta realizando en el árbol.
5. Si es posible, se borra el elemento del nodo.
6. Se desbloquea la escritura del nodo y la lectura del árbol.

Durante el borrado, sólo se producen fusiones y balanceos entre nodos si el árbol está bloqueado para escritura, ya que tanto en un balanceo como en una fusión, hace falta actualizar los nodos superiores, y para eso se necesita disponer del árbol de manera exclusiva.

4.2.6. Implementación del árbol B+

La implementación del árbol B+ la componen varios ficheros, cada uno con su funcionalidad.

- `arbolbmas.parmacs.c` Código del árbol B+
- `arbolbmas.h`: Cabecera para usar con otros módulos con la declaración del interfaz y de los tipos de datos usados.
- `readwrite.parmacs.c readwrite.parmacs.h`: Implementación del problema lector escritor, usado por el código del árbol B+.
- `arbolbmas-priv.h`: Macros y definiciones internas del árbol.

Problema del lector escritor

Este módulo resuelve el problema del lector-escritor, para ello define una estructura de sincronización y una serie de funciones para actuar sobre dicha estructura. Comenzaremos con las estructuras y tipos de datos:

▷ **struct struct_rw**

Estructura que define un sistema de bloqueo lector/escritor

- `LOCKDEC(cola)` Cerrojo que actúa como cola a la hora de encolar las operaciones de lectura o de escritura que se vayan solicitando.
- `LOCKDEC(seccion_critica)` Cerrojo que protege la variable controla el número de lectores actuales: num_lectores.
- `LOCKDEC(bloqueo_escritores)` Cerrojo utilizado para dar paso a un solo escritor.
- `int num_lectores;` Variable que controla el número actual de lectores.

▷ **rw_data**

Se define el tipo de dato rw_data de la siguiente manera:

```
typedef struct struct_rw rw_data;
```

Las funciones para el problema lector escritor y que utilizan la estructura anterior, son las siguientes:

▷ **rw_init**

- Declaración: `rw_data *rw_init(rw_data *nuevo)`
- Descripción: Inicializa un identificador lector/escritor.
  - Inicializa los 3 semáforos.

- Pone a 0 el contador de lectores.

- Parámetros:

  - `rw_data *nuevo` (entrada): Estructura lector/escritor a inicializar.

▷ **rw_read_ini**

- Declaración: `void rw_read_ini(rw_data *bloq)`

- Descripción: Añade un lector a la estructura lector/escritor.

  - Si el lector puede leer, la función devuelve el control y anota el lector extra.
  - Si el lector no puede leer debido a que hay un escritor trabajando, la función no devuelve el control y se queda a la espera de poder realizar la lectura.

- Parámetros:

  - `rw_data *bloq` (entrada): puntero a un identificador lector/escritor sobre el que trabajar.

▷ **rw_read_fin**

- Declaración: `void rw_read_fin(rw_data *bloq)`

- Descripción: Indica la finalización de una operación de lectura, por lo que resta un lector al identificador lector/escritor.

- Parámetros:

  - `rw_data *bloq` (entrada): puntero a un identificador lector/escritor sobre el que trabajar.

▷ **rw_write_ini**

- Declaración: `void rw_write_ini(rw_data *bloq)`

- Descripción: Añade un escritor al identificador lector/escritor, iniciando una operación de escritura.

  - Si al intentar añadir un escritor hay lectores u otro escritor, la función no devuelve el control y se mantiene a la espera.
  - En el momento que no exista ningún lector ni escritor, la función devuelve el control.

- Parámetros:

  - `rw_data *bloq` (entrada): puntero a un identificador lector/escritor sobre el que trabajar.

▷ **rw_write_fin**

- Declaración: `void rw_write_fin(rw_data *bloq)`
- Descripción: Finaliza una operación de escritura que ha comenzado.
- Parámetros:
  - `rw_data *bloq` (entrada): puntero a un identificador lector/escritor sobre el que trabajar.

Árbol B+

El interfaz para comunicarse con el árbol B+ es una fiel representación de la teoría; contiene las operaciones de búsqueda indexada y no indexada, modificación, inserción y borrado. Todas estas operaciones se hacen en base a la estructura que identifica a un árbol.

Aun así el árbol tiene unos parámetros que definen un poco su funcionamiento:

▷ **NODE_SIZE**

Para hacer un árbol de orden n este valor tiene que contener $n - 1$. Es la capacidad de un nodo: el número de claves y enlaces de un nodo hoja, y el número de claves de un nodo interno; NODE_SIZE+1 es el número de enlaces del nodo interno. El valor mínimo es 3.

▷ **NODE_SIZE_MIN**

Mínima cantidad de datos en un nodo.

- Ayuda a ajustar la altura del nodo de manera más o menos rápida. Según el valor de este valor mínimo: a un valor del mínimo más pequeño, más tarde se realizará el ajuste.
- Se debe de cumplir la regla: $min \leq (max)/2$, para que el nodo pueda dividirse por la mitad cuando se llena y no incumpla el mínimo.

Las estructuras y tipos de datos utilizadas en la comunicación con el árbol:

▷ **struct struct_ABMNodo**

Nodo del árbol, tanto interno como hoja:

- `unsigned char atributos;` Atributos del nodo. Actualmente sólo almacena si el nodo es interno o es hoja.
- `unsigned short ocupados;` Entradas ocupadas en el nodo.
- `unsigned int keysize;` Tamaño de las claves de este nodo. Este campo es necesario para que el nodo sea autosuficiente y se pueda operar con él sin necesidad de consultar la estructura principal de árbol.
- `rw_data bloq;` Estado de sincronización del nodo.
- `union enlaces` Unión para diferenciar tipos.

- `struct struct_ABMNodo *interno[NODE_SIZE+1];` Si el nodo es interno, los enlaces de ese nodo son a otros nodos
- `void *externo[NODE_SIZE];` Si el nodo es hoja, los enlaces son a los datos.

- `struct struct_ABMNodo *link;` Enlace a otro nodo.
  - Si el nodo es interno, es un enlace al nodo padre.
  - Si el nodo es hoja, es un enlace al nodo siguiente.

▷ **struct struct_Arbol**

Estructura que almacena información del árbol.

- `struct struct_ABMNodo *raiz;` Puntero al nodo raíz del árbol.
- `struct struct_ABMNodo *listaHojas;` Puntero a la hoja más a la izquierda del árbol.
- `unsigned int keysize;` Tamaño en bytes de la clave.
- `unsigned int regsize;` Tamaño en bytes del registro completo.
- `int (*comparar)(void *,void *);` Función de comparación de 2 claves a y b, que devuelve:
  - -1 si $a < b$
  - 0 si $a = b$
  - 1 si $a > b$
- `rw_data bloq;` Estado de sincronización del árbol.

▷ **struct struct_iterador**

Iterador para búsquedas secuenciales sin usar un campo clave.

- `struct struct_ABMNodo *actual;` Nodo en el que nos encontramos.
- `unsigned int pos;` Posición dentro del nodo actual en la que nos encontramos.
- `unsigned int regsize;` Tamaño de los registros a devolver.
- `int(*encaja)(void *,void *);` Función que dado un registro comprueba si cumple ciertas condiciones. Dados un registro a y unos datos, esta función devuelve 1 si dicho registro cumple las condiciones que implementa la función, o 0 si no las cumple.
- `void *datos;` Datos extra para la función encaja.
- `rw_data *bloq;` Estructura de sincronización del árbol.

▷ **ABMNodo**

Se define el tipo de dato ABMNodo como:

```
typedef struct struct_ABMNodo ABMNodo;
```

▷ **Arbol**

Se define el tipo de dato Arbol como:
```
typedef struct struct_Arbol Arbol;
```

▷ **cmpFunc**

Tipo de dato que identifica un puntero a una función de comparación que compara dos registros: a y b; de tal manera que:

- Si $a > b$: devuelve 1.
- Si $a = b$: devuelve 0.
- Si $a < b$: devuelve -1.

Se define el tipo de dato cmpFunc como:
```
typedef int(*cmpFunc)(void *,void*);
```

▷ **iterador**

Se define el tipo de dato iterador como:
```
typedef struct struct_iterador iterador
```

▷ **itFunc**

Tipo de dato para la función de búsqueda del iterador. Dado un registro devuelve 0 o 1 si dicho registro encaja o no con las especificaciones indicadas en la función respectivamente.

- Primer parámetro de entrada: registro siguiente.
- Segundo parámetro de entrada: datos extra indicados al crear el iterador.

Se define el tipo de dato itFunc como:
```
typedef int(*itFunc)(void *,void *);
```

▷ **doFunc**

Tipo de dato para la función de modificación de datos del árbol. Dados dos registros: uno el usado para la búsqueda y otro el que se ha encontrado en el árbol; realiza una operación con el fin, de actualizar el registro encontrado usando los datos del registro dado para la búsqueda.

Se define el tipo de dato doFunc como:
```
typedef void(*doFunc)(void *,void *);
```
Por último, vamos a ver las *funciones* que hacen de interfaz con el árbol:

▷ **abm_nuevo**

- Declaración: `Arbol *abm_nuevo(int regsize,int keysize,cmpFunc comparar)`
- Descripción:Crea un árbol B+ nuevo.

- Reserva la memoria necesaria para la estructura que identifica al árbol.
- La inicializa a un estado que indique que el árbol está vacío: con sólo un nodo que es hoja y vacío.

- Parámetros:
  - `int regsize` (entrada): Tamaño en bytes del registro a usar.
  - `int keysize` (entrada): Tamaño de la clave del registro.
  - `cmpFunc comparar` (entrada): Función que compara dos claves de dos registros. De tal manera que si al 1er parámetro le denominamos A y al segundo B, al llamar a la función comparar(A,B), esta devolverá:
    - -1 si $A < B$
    - 0 si $A = B$
    - 1 si $A > B$
- Devuelve: Un puntero al nuevo árbol creado.

▷ **abm_destruir**

- Declaración: `void abm_destruir(Arbol *arb)`
- Descripción: (función sincronizada) Destruye un árbol borrando todo su contenido, lo que incluye:
  - Los registros asociados.
  - Los nodos que lo componen.
  - La estructura que identifica al árbol.
- Parámetros:
  - `Arbol *arb` (entrada): Puntero al árbol a destruir.

▷ **abm_insertar**

- Declaración: `int abm_insertar(Arbol *arb,void *registro)`
- Descripción: (función sincronizada) Inserta un registro en el árbol. Para insertar un registro en el árbol:
  - Primero se localiza el nodo donde insertarlo.
  - Si el nodo en cuestión está lleno hay que dividirlo, lo que puede encadenar más de una división.
  - Si no, se reserva memoria compartida para el registro, y se inserta.
- Parámetros:
  - `Arbol *arb` (entrada): árbol con el que trabajar.
  - `void *registro` (entrada): Zona de memoria con el registro a insertar. Después de la inserción, esta memoria se puede liberar, ya que se hace copia del contenido.

- Devuelve: Un entero con el estado de la operación.
  - `ABM_ERROR` En caso de error.
  - `ABM_DUP` En caso de que el registro ya existiera.
  - `ABM_OK` Si el registro ha sido insertado satisfactoriamente.

▷ **abm_buscar**

- Declaración: `int abm_buscar (Arbol *arb,void *registro)`
- Descripción: (función sincronizada) Busca un registro dada una clave. Utiliza el parámetro registro tanto para entrada: es ahí donde se indica la clave a buscar, como para salida: en caso de encontrar el registro, los datos van a parar a esa estructura.
- Parámetros:
  - `Arbol *arb` (entrada): árbol donde realizar la operación.
  - `void *registro` (entrada y salida): Contiene la clave por la que buscar, y almacena el resultado en caso de que la búsqueda de resultados.
- Devuelve: Un entero indicando si la búsqueda ha tenido o no éxito.
  - `ABM_OK` Se ha encontrado lo que se buscaba.
  - `ABM_404` No se ha encontrado el registro buscado.

▷ **abm_modificar**

- Declaración: `int abm_modificar(Arbol *arb,void *registro,doFunc hacer)`
- Descripción: (función sincronizada) Modifica un registro dada su clave. Modifica el contenido de un registro buscándolo a partir de su clave. Es importante señalar, que NO SE DEBE cambiar la clave del registro con esta operación, pues los resultados NO se reflejarán en el árbol y el registro sólo se podrá acceder usando la clave anterior.
- Parámetros:
  - `Arbol *arb` (entrada): árbol donde realizar la operación.
  - `void *registro` (entrada): Registro que contiene tanto la clave como los nuevos datos a insertar, esto último en el caso de que hacer sea NULL.
  - `doFunc hacer` (entrada): Función que contiene una operación a realizar cuando se encuentre el registro a modificar. Este parámetro puede ser NULL, y en este caso se cambiaran los datos del registro encontrado a los del parámetro "registro"; si no es null, se utilizará la clave del parámetro registro, y cuando se encuentre el dato, se aplicará la función.
- Devuelve: Un entero indicando el resultado de la operación.
  - `ARB_OK` Se ha encontrado y modificado el registro
  - `ABM_404` Valor no encontrado, por lo que no se pudo modificar.

▷ **abm_borrar**

- Declaración: `int abm_borrar (Arbol *arb,void *clave)`
- Descripción: (función sincronizada) Borra un registro del árbol. El borrado de un registro del árbol es una de las operaciones más costosas debido a que puede implicar desde el simple borrado del registro, balanceo entre hojas, hasta borrado de nodos y reajuste del árbol.
- Parámetros:
  - `Arbol *arb` (entrada): árbol donde realizar las operaciones.
  - `void *clave` (entrada): Clave del registro a borrar.
- Devuelve: Un entero indicando el resultado de la operación.
  - `ABM_OK` La operación se ha realizado sin incidencias.

▷ **abm_it_init**

- Declaración: `void abm_it_init (iterador *it,Arbol *arb,itFunc encaja,void *datos)`
- Descripción: (función sincronizada) Inicializa un nuevo iterador. Dado un árbol, inicializa un iterador para recorrer el árbol secuencialmente a través de sus nodos hoja en busca de registros que encajen en las condiciones que aplicará la función "encaja".
- Parámetros:
  - `iterador *it` (entrada): Iterador a inicializar.
  - `Arbol *arb` (entrada): árbol que recorrer.
  - `itFunc encaja` (entrada): Función de encaje. Su función consiste en que dado un registro devuelve 1 si cumple ciertas condiciones que tiene la función internamente, o 0 si no las cumple..
  - `void *datos` (entrada): Información extra para que la función encaja trabaje.

▷ **abm_it_fin**

- Declaración: `void abm_it_fin(iterador *it)`
- Descripción: (función sincronizada) Pone fin a una iteración por los nodos hoja del árbol. Es necesario ejecutar esta función después de terminar el recorrido para liberar el bloqueo de lectura del árbol.
- Parámetros:
  - `iterador *it` (entrada): Iterador sobre el que trabajar.

▷ **abm_iterar**

- Declaración: `int abm_iterar(iterador *itr,void *registro)`
- Descripción: (función sincronizada) Realiza una iteración entre los nodos hoja del árbol. Busca solamente un registro con el que la función "encaja" esté de acuerdo, una vez encontrado lo devuelve, pudiendo llamar más tarde a esta función para continuar.
- Parámetros:
  - `iterador *itr` (entrada): Iterador sobre el que trabajar y que contiene el estado actual de iteración.
  - `void *registro` (salida): Zona de memoria donde guardar el registro encontrado.
- Devuelve: Un entero indicando el estado de la operación.
  - `ABM_OK` Se ha encontrado un registro.
  - `ABM_FIN` Se ha llegado al final de la lista de nodos hoja y no se ha encontrado ningún registro más. Se debe llamar a abm_it_fin para finalizar la iteración.

Peculiaridades de la implementación

Al igual que en la lista simplemente enlazada, una de las peculiaridades es la de admitir registros genéricos, y en este caso dado que hace falta un campo clave (o unos campos clave), también se permiten claves genéricas.

Si un nodo, tanto interno como hoja, contiene las claves además de los enlaces, dado que las claves son de tamaño arbitrario (dependen del registro), cuando se reserva memoria para la estructura del nodo, también se añade la memoria necesaria para las claves:

```
nuevoNodo=(ABMNodo *)G_MALLOC(sizeof(ABMNodo) + NODE_SIZE*keysize)
```

Esto implica manejo de punteros para establecer y obtener una clave. Para simplificar estas operaciones se ha hecho uso de macros del preprocesador de C.

- Obtener una clave:
  ```
  #define getNodeKey(nodo,pos) ((void
  *)(nodo+1))+((pos)*getKeySize(nodo))
  ```
- Establecer una clave:
  ```
  #define setNodeKey(nodo,pos,clave)
  memcpy(getNodeKey(nodo,pos),clave,getKeySize(nodo))
  ```
- Macro auxiliar de obtener el tamaño de la clave de un nodo:
  ```
  #define getKeySize(nodo) nodo->keysize
  ```

4.2.7. Banco de pruebas del subsistema

Este subsistema es la base del benchmark TPC-C, un fallo en cualquiera de sus funciones puede suponer a alto nivel un error indetectable o una corrupción de las estructuras que acabe dando un error fatal. Por lo que se ha hecho un esfuerzo especial en probar todas las funcionalidades de estos dos métodos de almacenamiento.

Lista Enlazada

Es el sistema de almacenamiento más simple con solo dos operaciones: añadir al final y borrar del principio. Las pruebas por las que ha pasado y ha superado son las siguientes.

- Insertar números de manera secuencial.
- Recuperar dichos números en el mismo orden que se insertaron borrando y recuperando por el principio.
- Destruir la lista y con ello liberar la memoria.

Árbol B+

Para el árbol, dado que es más complejo y tiene más operaciones se han realizado muchas más pruebas, sobre todo basándose en inserciones y borrados aleatorios ya que representan mejor un funcionamiento real. Las pruebas a las que ha sido sometido y ha superado son:

- Iterar en un árbol vacío no debe ocasionar errores.
- No se debe de poder insertar un valor duplicado.
- Rellenar con una secuencia de números aleatoria: dado un rango de números, barajarlos e insertarlos.
- Buscar un dato concreto.
- Modificar un dato concreto.
- Modificar utilizando una función de modificación.
- Iterar buscando pares.
- Iterar buscando múltiplos de 7.
- Volcado de la estructura en memoria a modo texto para observar de manera visual que se cumplen todas las propiedades del árbol.
- Borrado de dicho rango de números, pero con otro barajeo distinto al de inserción. Después de borrado, se comprueba de manera automática algunas condiciones del árbol:
  - La lista de nodos recorre todos los nodos.
  - Se puede llegar a todos los datos.
  - Las claves conservan el orden.

Aplicaciones de prueba

Ambas pruebas han sido implementadas en programas de prueba; esto facilita el comprobar la validez de las estructuras tras pequeños cambios y correcciones. Los programas que comprueban las estructuras son los siguientes.

- `abm_test` que proviene de `abm_test.parmacs.c`. Realiza las pruebas indicadas anteriormente al árbol B+.
- `le_test` que proviene de `le_test.c`. Realiza las pruebas indicadas anteriormente a la lista enlazada.

4.3. Subsistema generador

El subsistema generador es el encargado de obtener dos tipos de datos:

- El poblado inicial para el subsistema de almacenamiento.
- La carga que las terminales van a enviar a los servidores.

Para generar la carga, implementan las especificaciones que se pueden encontrar a lo largo del apartado 3.1 (pag. 43). Durante el análisis del benchmark TPC-C se especificaron:

- Las diferentes tablas donde almacenar datos.
- La descripción exacta de cada registro.
- Qué cantidades hacen falta de cada registro.
- Cómo generar dichas cantidades.
- Cuántas y cómo repartir las diferentes transacciones para que se cumpla el porcentaje asignado a cada una.

Los ficheros que integran este subsistema, junto con su funcionalidad son los siguientes:

- `registros.h` Especificaciones de los registros: atributos, formato para escribirlos en un fichero y formato para recuperarles de un fichero.
- `basicgen.c basicgen.h` Dado que ciertos datos se generan de manera aleatoria con unas especificaciones, dicha generación se repite en muchas ocasiones, por lo que aquí se han agrupado las funciones de generación de datos más utilizadas.
- `generador.h` Configura el generador con las cardinalidades de poblado de las diferentes tablas, así como en que ficheros volcar los datos.
- `generador.c` Programa independiente que al ejecutarse genera un *experimento*, consistente en: una carga de trabajo y un poblado.

4.3.1. Diseño e implementación de los registros

Dado que el lenguaje de programación es C, hay que adaptar los tipos de atributos especificados en el análisis a tipos de datos lo más sencillos posibles que podamos encontrar en C. Se han resumido los siguientes tipos de datos, con sus equivalencias en el lenguaje C:

- Identificadores de cualquier tipo: `uint32_t atrib;`
- Texto tamaño variable, N: `char atrib[N+1];`
- Texto tamaño fijo, N: `char atrib[N+1];`
- Números hasta 9 dígitos: `uint32_t atrib;`
- Números hasta 19 dígitos: `uint64_t atrib;`
- Fecha y hora: `time_t atrib;`

Para los números con decimales, se les ha normalizado multiplicándoles por el número de ceros necesarios para que no se utilicen decimales. Por ejemplo, para la unidad monetaria, que en nuestro caso son los euros, dado que existen céntimos de euro, se ha multiplicado dicha unidad por 100. Esto, a la hora de realizar las operaciones, se tiene en cuenta, y se ha modificado la lógica de negocio para que se trabaje con céntimos de euro en vez de euros. Para el resto de medidas que también se han modificado, como pueden ser descuentos u otros porcentajes, también se ha modificado la manera de operar para no utilizar decimales.

En cada registro se ha dispuesto una constante que indica el tamaño de la clave, para hacer más fácil su implementación. Otras constantes que se pueden encontrar en los registros son las relacionadas con la lectura y escritura en disco, para usar con las funciones *fprintf* y *fscanf*, de la biblioteca estándar de entrada y salida en C.

Veamos la composición y características de los diferentes registros. Las constantes que definen los formatos de entrada y salida no se han incluido debido a su ilegibilidad, pero se ha incluido una referencia a su posición en el fichero `registros.h`.

▷ Registro Almacén

- Tamaño de la clave de los registro del tipo *almacén*:
  `#define REGALMACEN_KEYSIZE sizeof(struct {uint32_t w_id;})`
- Cadena para el volcado del registro *almacén* vía *printf:
  `#define DUMPSTRING_ALMACEN` (línea 25)
- Cadena para la lectura del registro *almacén* vía *scanf:
  `#define READSTRING_ALMACEN` (línea 27)
- Parámetros a usar con la cadena de volcado, partiendo de un registro *almacén* (alm):
  `#define DUMPPARAM_ALMACEN(alm)` (línea 29)
- Parámetros a usar con la cadena de lectura, partiendo de un registro *almacén* (alm):
  `#define READPARAM_ALMACEN(alm)` (línea 32)
- Registro equivalente a una tupla de la tabla Almacén:
  `struct struct_RegAlmacen`:
  - `uint32_t w_id;`
  - `char w_name[11];`
  - `char w_street_1[21];`
  - `char w_street_2[21];`
  - `char w_city[21];`
  - `char w_state[3];`
  - `char w_zip[10];`
  - `uint32_t w_tax;`
  - `uint64_t w_ytd;`

▷ **Registro Zona**

- Tamaño de la clave de los registro del tipo *zona*:
  `#define REGZONA_KEYSIZE sizeof(struct {uint32_t d_id;})`
- Cadena para el volcado del registro *zona* vía *printf:
  `#define DUMPSTRING_ZONA` (línea 53)
- Cadena para la lectura del registro *zona* vía *scanf:
  `#define READSTRING_ZONA` (línea 55)
- Parámetros a usar con la cadena de volcado, partiendo de un registro *zona* (zon):
  `#define DUMPPARAM_ZONA(zon)` (línea 57)
- Parámetros a usar con la cadena de lectura, partiendo de un registro *zona* (zon):
  `#define READPARAM_ZONA(zon)` (línea 60)
- Registro equivalente a una tupla de la tabla Zona:
  `struct struct_RegZona`:
  - `uint32_t d_id;`
  - `uint32_t d_w_id;`
  - `char d_name[11];`
  - `char d_street_1[21];`
  - `char d_street_2[21];`
  - `char d_city[21];`
  - `char d_state[3];`
  - `char d_zip[9];`
  - `uint32_t d_tax;`
  - `uint64_t d_ytd;`
  - `uint32_t d_next_o_id;`

▷ **Registro Cliente**

- Tamaño de la clave de los registro del tipo *cliente*:
  `#define REGCLIENTE_KEYSIZE sizeof(struct {uint32_t c_id; uint32_t c_d_id; uint32_t c_w_id;})`
- Cadena para el volcado del registro *cliente* vía *printf:
  `#define DUMPSTRING_CLIENTE` (línea 83)
- Cadena para la lectura del registro *cliente* vía *scanf:
  `#define READSTRING_CLIENTE` (línea 85)
- Parámetros a usar con la cadena de volcado, partiendo de un registro *cliente* (cl):
  `#define DUMPPARAM_CLIENTE(cl)` (línea 87)
- Parámetros a usar con la cadena de lectura, partiendo de un registro *cliente* (cl):
  `#define READPARAM_CLIENTE(cl)` (línea 92)

- Registro equivalente a una tupla de la tabla Cliente:
  `struct struct_RegCliente`
  - `uint32_t c_id;` (Campo clave)
  - `uint32_t c_d_id;` (Campo clave)
  - `uint32_t c_w_id;` (Campo clave)
  - `char c_first[17];`
  - `char c_middle[3];`
  - `char c_last[17];`
  - `char c_street_1[21];`
  - `char c_street_2[21];`
  - `char c_city[21];`
  - `char c_state[3];`
  - `char c_zip[10];`
  - `char c_phone [17];`
  - `time_t c_since;`
  - `char c_credit[3];`
  - `uint64_t c_credit_lim;`
  - `uint32_t c_discount;`
  - `int64_t c_balance;`
  - `uint64_t c_ytd_payment;`
  - `uint32_t c_payment_cnt;`
  - `uint32_t c_delivery_cnt;`
  - `char c_data[501];`

▷ **Registro Histórico**

- Tamaño de la clave de los registro del tipo *histórico*:
  `#define REGHISTORICO_KEYSIZE 0`
- Cadena para el volcado del registro *histórico* vía *printf:
  `#define DUMPSTRING_HISTORICO` (línea 126)
- Cadena para la lectura del registro *histórico* vía *scanf:
  `#define READSTRING_HISTORICO` (línea 127)
- Parámetros a usar con la cadena de volcado, partiendo de un registro *histórico* (his):
  `#define DUMPPARAM_HISTORICO(his)` (línea 128)
- Parámetros a usar con la cadena de lectura, partiendo de un registro *histórico* (his):
  `#define READPARAM_HISTORICO(his)` (línea 130)
- Registro equivalente a una tupla de la tabla Histórico:
  `struct struct_RegHistorico`

- `uint32_t h_c_id;`
- `uint32_t h_c_d_id;`
- `uint32_t h_c_w_id;`
- `uint32_t h_d_id;`
- `uint32_t h_w_id;`
- `time_t h_date;`
- `uint32_t h_amount;`
- `char h_data[25];`

▷ **Registro NuevoPedido**

- Tamaño de la clave de los registro del tipo *nuevo pedido*:
  `#define REGNUEVOPEDIDO_KEYSIZE sizeof(struct {uint32_t no_o_id; uint32_t no_d_id; uint32_t no_w_id;})`
- Cadena para el volcado del registro *NuevoPedido* vía *printf:
  `#define DUMPSTRING_NUEVOPEDIDO` (línea 149)
- Cadena para la lectura del registro *NuevoPedido* vía *scanf:
  `#define READSTRING_NUEVOPEDIDO DUMPSTRING_NUEVOPEDIDO`
- Parámetros a usar con la cadena de volcado, partiendo de un registro *NuevoPedido* (np):
  `#define DUMPPARAM_NUEVOPEDIDO(np)` (línea 151)
- Parámetros a usar con la cadena de lectura, partiendo de un registro *NuevoPedido* (np):
  `#define READPARAM_NUEVOPEDIDO(np)` (línea 152)
- Registro equivalente a una tupla de la tabla NuevoPedido:
  `struct struct_RegNuevoPedido`
  - `uint32_t no_o_id;` (Campo clave)
  - `uint32_t no_d_id;` (Campo clave)
  - `uint32_t no_w_id;` (Campo clave)

▷ **Registro Pedido**

- Tamaño de la clave de los registro del tipo *pedido*:
  `#define REGPEDIDO_KEYSIZE sizeof(struct {uint32_t o_id; uint32_t o_d_id; uint32_t o_w_id;})`
- Cadena para el volcado del registro *pedido* vía *printf:
  `#define DUMPSTRING_PEDIDO` (línea 165)
- Cadena para la lectura del registro *pedido* vía *scanf:
  `#define READSTRING_PEDIDO DUMPSTRING_PEDIDO`
- Parámetros a usar con la cadena de volcado, partiendo de un registro *pedido* (ped):
  `#define DUMPPARAM_PEDIDO(ped)` (línea 167)

- Parámetros a usar con la cadena de lectura, partiendo de un registro *pedido* (ped): `#define READPARAM_PEDIDO(ped)` (línea 169)
- Registro equivalente a una tupla de la tabla Pedido: `struct struct_RegPedido`
  - `uint32_t o_id;` (Campo clave)
  - `uint32_t o_d_id;` (Campo clave)
  - `uint32_t o_w_id;` (Campo clave)
  - `uint32_t o_c_id;`
  - `time_t o_entry_d;`
  - `uint32_t o_carrier_id;`
  - `uint32_t o_ol_cnt;`
  - `uint32_t o_all_local;`

▷ **Registro LíneaPedido**

- Tamaño de la clave de los registro del tipo *LineaPedido*: `#define REGLINEAPEDIDO_KEYSIZE sizeof(struct {uint32_t ol_i_id; uint32_t ol_d_id; uint32_t ol_w_id; uint32_t ol_number;})`
- Cadena para el volcado del registro *LíneaPedido* vía *printf: `#define DUMPSTRING_LINEAPEDIDO` (línea 188)
- Cadena para la lectura del registro *LíneaPedido* vía *scanf: `#define READSTRING_LINEAPEDIDO` (línea 189)
- Parámetros a usar con la cadena de volcado, partiendo de un registro *LíneaPedido* (lp): `#define DUMPPARAM_LINEAPEDIDO(lp)` (línea 190)
- Parámetros a usar con la cadena de lectura, partiendo de un registro *LíneaPedido* (lp): `#define READPARAM_LINEAPEDIDO(lp)` (línea 193)
- Registro equivalente a una tupla de la tabla LíneaPedido: `struct struct_RegLineaPedido`
  - `uint32_t ol_o_id;` (Campo clave)
  - `uint32_t ol_d_id;` (Campo clave)
  - `uint32_t ol_w_id;` (Campo clave)
  - `uint32_t ol_number;` (Campo clave)
  - `uint32_t ol_i_id;`
  - `uint32_t ol_supply_w_id;`
  - `time_t ol_delivery_d;`
  - `uint32_t ol_quantity;`
  - `uint32_t ol_amount;`
  - `char ol_dist_info[25];`

▷ **Registro Existencias**

- Tamaño de la clave de los registros del tipo *existencias*:
  `#define REGEXISTENCIAS_KEYSIZE sizeof(struct {uint32_t s_id; uint32_t s_w_id;})`
- Cadena para el volcado del registro *existencias* vía *printf:
  `#define DUMPSTRING_EXISTENCIAS` (linea 215)
- Cadena para la lectura del registro *existencias* vía *scanf:
  `#define READSTRING_EXISTENCIAS` (línea 216)
- Parámetros a usar con la cadena de volcado, partiendo de un registro *existencias* (ex):
  `#define DUMPPARAM_EXISTENCIAS(ex)` (línea 217)
- Parámetros a usar con la cadena de lectura, partiendo de un registro *existencias* (ex):
  `#define READPARAM_EXISTENCIAS(ex)` (línea 220)
- Registro equivalente a una tupla de la tabla Existencias:
  `struct struct_RegExistencias`
  - `uint32_t s_i_id;` (Campo clave)
  - `uint32_t s_w_id;` (Campo clave)
  - `uint32_t s_quantity;`
  - `char s_dist_01[25];`
  - `char s_dist_02[25];`
  - `char s_dist_03[25];`
  - `char s_dist_04[25];`
  - `char s_dist_05[25];`
  - `char s_dist_06[25];`
  - `char s_dist_07[25];`
  - `char s_dist_08[25];`
  - `char s_dist_09[25];`
  - `char s_dist_10[25];`
  - `uint32_t s_ytd;`
  - `uint32_t s_order_cnt;`
  - `uint32_t s_remote_cnt;`
  - `char s_data[51];`

▷ **Registro Producto**

- Tamaño de la clave de los registros del tipo *producto*:
  `#define REGPRODUCTO_KEYSIZE sizeof(struct {uint32_t i_id;})`
- Cadena para el volcado del registro *producto* vía *printf:
  `#define DUMPSTRING_PRODUCTO` (línea 250)

- Cadena para la lectura del registro *producto* vía *scanf:
  `#define READSTRING_PRODUCTO` (línea 251)
- Parámetros a usar con la cadena de volcado, partiendo de un registro *producto* (prod):
  `#define DUMPPARAM_PRODUCTO(prod)` (línea 252)
- Parámetros a usar con la cadena de lectura, partiendo de un registro *producto* (prod):
  `#define READPARAM_PRODUCTO(prod)` (línea 253)
- Registro equivalente a una tupla de la tabla Productos:
  `struct struct_RegProducto`
  - `uint32_t i_id;` (Campo clave)
  - `uint32_t i_im_id;`
  - `char i_name[25];`
  - `uint32_t i_price;`
  - `char i_data[51];`

Tipos de datos

Para un uso más simple y una implementación más clara, con los registros actuales se han definido los siguientes tipos de datos.

▷ RegAlmacen

Se define el registro de la tabla de almacenes RegAlmacen como:
```
typedef struct struct_RegAlmacen RegAlmacen;
```

▷ RegZona

Se define el registro de la tabla de zonas RegZona como:
```
typedef struct struct_RegZona RegZona;
```

▷ RegCliente

Se define el registro de la tabla de clientes RegCliente como:
```
typedef struct struct_RegCliente RegCliente;
```

▷ RegHistorico

Se define el registro de la tabla del histórico de pedidos RegHistorico como:
```
typedef struct struct_RegHistorico RegHistorico;
```

▷ RegNuevoPedido

Se define el registro de la tabla de nuevos pedidos RegNuevoPedido como:
```
typedef struct struct_RegNuevoPedido RegNuevoPedido;
```

▷ **RegPedido**

Se define el registro de la tabla de pedidos como:
`typedef struct struct_RegPedido RegPedido;`

▷ **RegLineaPedido**

Se define el registro de la tabla de líneas de pedido RegLineaPedido como:
`typedef struct struct_RegLineaPedido RegLineaPedido;`

▷ **RegExistencias**

Se define el registro de la tabla de existencias como:
`typedef struct struct_RegExistencias RegExistencias;`

▷ **RegProducto**

Se define el registro de la tabla de productos RegProducto como:
`typedef struct struct_RegProducto RegProducto;`

4.3.2. Generadores básicos

A la hora de cumplir con los requisitos del análisis en cuanto a la generación de ciertos campos, se utilizan las funciones del módulo `basicgen.c`. Estos generadores básicos se usan tanto para generar el poblado como para generar la carga de trabajo de los servidores. Los generadores básicos que se han implementado son los siguientes.

▷ **gen_last**

- Declaración: `void gen_last(char *last,int num)`
- Descripción: Genera un apellido según las reglas del benchmark TPC-C. Los apellidos se crean a partir de un número de 3 dígitos, donde cada dígito equivale a un vocablo. Los 3 vocablos concatenados forman el apellido.
- Parámetros:
  - `char *last` (salida): Lugar donde almacenar el apellido generado.
  - `int num` (entrada): Número de 3 dígitos con el cual generar el apellido..

▷ **gen_a_string**

- Declaración: `void gen_a_string(char *destino,int a, int b)`
- Descripción: Genera una cadena de caracteres aleatorios. Dado un número mínimo de caracteres y un número máximo, se crea una cadena con caracteres aleatorios entre esas dos posiciones, ambas inclusive. Aunque tiene ciertas restricciones:
  - No empieza las cadenas con espacios.
  - No termina las cadenas con espacios.

- Parámetros:
  - `char *destino` (salida): Lugar donde colocar la cadena generada.
  - `int a` (entrada): Número mínimo de caracteres.
  - `int b` (entrada): Número máximo de caracteres.

▷ **gen_n_string**

- Declaración: `void gen_n_string(char *destino,uint32_t a,uint32_t b)`
- Descripción: Genera una cadena de dígitos aleatorios. Con un mínimo y un máximo de longitud, genera una cadena de dígitos decimales aleatorios con una longitud aleatoria entre los dos límites, ambos incluidos.
- Parámetros:
  - `char *destino` (salida): Lugar donde colocar la cadena generada.
  - `uint32_t a` (entrada): Número mínimo de dígitos.
  - `uint32_t b` (entrada): Número máximo de dígitos.

▷ **gen_zip**

- Declaración: `void gen_zip (char *destino)`
- Descripción: Genera un código postal. Según los requisitos, un código postal se crea concatenando a una cadena aleatoria de 4 dígitos, la cadena "1111".
- Parámetros:
  - `char *destino` (salida): Lugar donde colocar el código postal generado.

▷ **gen_number**

- Declaración: `uint32_t gen_number(uint32_t a,uint32_t b)`
- Descripción: Genera un número entero positivo entre dos valores, ambos incluidos en los posibles valores de salida.
- Parámetros:
  - `uint32_t a` (entrada): Valor mínimo.
  - `uint32_t b` (entrada): Valor máximo.
- Devuelve: Un entero sin signo generado aleatoriamente.

▷ **gen_NURand**

- Declaración: `#define gen_NURand(A,x,y) ((((gen_number(0,A) | gen_number(x,y))+CValue) % (y-x+1))+x)`
- Descripción: Macro utilizada para generar un número aleatorio mediante una distribución no uniforme. Necesita de la variable CValue para funcionar correctamente
- Parámetros:
  - `A` (entrada): Constante de trabajo cuyo valor es:
    - Para el rango [0...999], 255
    - Para el rango [0...3.000], 1.023
    - Para el rango [0...100.000], 8.191
  - `x` (entrada): Valor mínimo.
  - `y` (entrada): Valor máximo.
- Devuelve: un número aleatorio no uniforme.

Y por último, en `basicgen.h`, tenemos algunos parámetros con los que variar el funcionamiento de los generadores básicos. Dichos parámetros son:

▷ **MAPA_NUMEROS**

Cadena de caracteres que indica con que dígitos se va a trabajar a la hora de interpretar un número y convertirlo en una cadena de caracteres. Se define como:

```
#define MAPA_NUMEROS "0123456789"
```

Y su longitud, para acelerar el funcionamiento, se indica de esta manera:

```
#define MAPANUM_LEN 10
```

▷ **MAPA_CARACTERES**

Cuando se generan cadenas de caracteres aleatorios, los caracteres a incluir en esa cadena generada se obtienen de la cadena MAPA_CARACTERES. Se define como:

```
#define MAPA_CARACTERES
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
-_."
```

Y su longitud, para acelerar el funcionamiento, se indica de la siguiente manera:

```
#define MAPACAR_LEN 66
```

4.3.3. Terminales simuladas

Por un lado está el poblado de datos, cuya generación es muy sencilla (ver apartado 3.3); pero por otro está la carga de trabajo. La carga de trabajo son los datos que necesita cada transacción para empezar a funcionar; si se revisan las especificaciones de cada transacción, se verá que primero se reunen unos cuantos datos iniciales, que son facilitados por el usuario de la terminal, y con dichos datos se lanza la transacción.

Para simular esta entrada/salida, los datos que las terminales enviarían al servidor para que realizase una transacción se van a guardar en un fichero de carga (uno por cada servidor); y el módulo encargado de fabricar esos ficheros es el que codifica el fichero `terminal.c`

Interfaz público

El generador, por cada servidor que le pidan que genere unos datos, creará un fichero donde guardar dichos datos y le pedirá al simulador de terminales que genere la carga de n transacciones en dicho fichero. Esto lo hace a través de la función: terminal_tpcc.

▷ **terminal_tpcc**

- Declaración: `void terminal_tpcc(FILE *fich,uint64_t numTran);`
- Descripción: Genera carga de trabajo. Dado un fichero de destino y un número de transacciones para las que generar carga, va barajando una baraja de transacciones (segundo método para conseguir la proporción de transacciones, pag. 65), y va generando de manera aleatoria la carga para cada transacción.
- Parámetros:
  - `FILE *fich` (entrada): puntero al descriptor del fichero utilizado para almacenar la carga de trabajo.
  - `uint64_t numTran` (entrada): Número de transacciones para generar carga.

Interfaz privado

Internamente, después de seleccionar el tipo de transacción para la cual se va a generar carga, acude a una función especializada en generar carga de cada tipo de transacción, pasándole como parámetro el fichero donde debe volcar la carga. Dichas funciones, de uso interno de terminal_tpcc son:

- `void term_nuevo_pedido (FILE *);` Genera los datos iniciales para una transacción de nuevo pedido.
- `void term_pago (FILE *);` Genera los datos iniciales para una transacción de pago.
- `void term_estado_pedido (FILE *);` Genera los datos iniciales para una transacción de estado de un pedido.
- `void term_envio (FILE *);` Genera los datos iniciales para una transacción de envío.
- `void term_nivel_existencias(FILE *);`

Formato de la salida

Todas estas funciones, insertan una sola linea de texto ascii con campos separados por tabuladores que contiene los datos necesarios para realizar la transacción, dicha línea va precedida de un número que identifica a cada tipo de transacción:

- 0 para nuevo pedido.
- 1 para pago.
- 2 para estado de un pedido.

- 3 para envío.
- 4 para nivel de existencias.

Además, la función de nuevo pedido, dado que necesita indicar los datos de las líneas de pedido, inserta tantas líneas como líneas de pedido; pero dichas líneas no tienen un número delante que las identifique. Al leer los datos para una transacción de nuevo pedido, se lee primero la línea marcada con un 0, y en dicha línea se encuentra el número de líneas de pedido, que son las que se extraerán del fichero.

Los formatos de entrada y de salida se definen en el fichero `terminal.h`, son para las funciones fprintf (salida) y fscanf(entrada); son los siguientes:

- Formatos de salida para los datos que necesitan las diferentes transacciones para ejecutarse:
  - Transacción de nuevo pedido:
    `#define DS_TERM_NUEVOPEDIDO "0\t %u\t %u\t %u\t %u\n"`
  - Transacción de nuevo pedido, cada línea de pedido:
    `#define DS_TERM_LINEAPEDIDO "\t %u\t %u\t %u\n"`
  - Transacción de pago:
    `#define DS_TERM_PAGO`
    `"1\t %u\t %u\t %u\t %s\t %u\t %u\t %u\t %u\t\n"`
  - Transacción de estado de un pedido:
    `#define DS_TERM_ESTADOPEDIDO "2\t %u\t %u\t %u\t %s\n"`
  - Transacción de envío:
    `#define DS_TERM_ENVIO "3\t %u\t %u\t %u\n"`
  - Transacción de nivel de existencias:
    `#define DS_TERM_NIVELEXISTENCIAS "4\t %u\t %u\t %u\n"`
- Para la lectura desde fichero (entrada) de los datos que necesitan las diferentes transacciones:
  - Transacción de nuevo pedido:
    `#define RS_TERM_NUEVOPEDIDO " %u\t %u\t %u\t %u\n"`
  - Transacción de nuevo pedido, cada línea de pedido:
    `#define RS_TERM_LINEAPEDIDO "\t %u\t %u\t %u\n"`
  - Transacción de pago:
    `#define RS_TERM_PAGO " %u\t %u\t %u\t %s\t %u\t %u\t %u\t %u\t\n"`
  - Transacción de estado de pedido:
    `#define RS_TERM_ESTADOPEDIDO " %u\t %u\t %u\t %s\n"`
  - Transacción de envío:
    `#define RS_TERM_ENVIO " %u\t %u\t %u\n"`
  - Transacción de nivel de existencias:
    `#define RS_TERM_NIVELEXISTENCIAS " %u\t %u\t %u\n"`

4.3.4. Generador de carga y poblado

Por último, la aplicación generador, que se utiliza para obtener el poblado y la carga de datos. Su interfaz con el usuario es a través de opciones indicadas en la línea de órdenes y devuelve su trabajo en forma de ficheros. A la salida que se genera se le llama *experimento*.

Parámetros a la hora de generar un experimento

Los siguientes parámetros se pueden indicar en la línea de órdenes a la hora de ejecutar el generador para variar su funcionamiento.

- `-h` Muestra un mensaje de ayuda con los parámetros y sus valores por defecto.
- `-c` No genera el poblado de la base de datos. Esto es útil cuando sólo se quiere cambiar la carga del sistema; por ejemplo: añadir más transacciones o cambiar el número de terminales, usando el mismo poblado.
- `-a número` Por defecto se genera el poblado para un almacén, si se desean más almacenes, no hay más que indicar aquí el número total de almacenes para los que generar el poblado.
- `-t número` Por defecto se generan 100 transacciones; si se desean más, se indica en este número. El máximo de transacciones que se pueden indicar es $2^{64} = 18446744073709551615$.
- `-s número` El número de procesos servidor para los que generar carga. El número de transacciones total se dividirá entre el número de servidores, y la cantidad resultante será la carga total para cada servidor.

Por último, y como parámetro final, se debe especificar un directorio donde depositar la carga generada; este último parámetro es obligatorio. Para conocer más sobre ejemplos y el modo de uso de este programa, acudir al apéndice B.

Ficheros del experimento

Una vez tenemos los parámetros y se ha ejecutado el generador, obtendremos en el directorio indicado una serie de ficheros, cada uno con un contenido diferente. Veamos cual es su nombre y su aplicación:

- Ficheros acabados en: *_productos.txt, _almacenes.txt, _existencias.txt, _zonas.txt, _clientes.txt, _histórico.txt, _pedidos.txt, _lineaspedido.txt y _nuevospedidos.txt* ; son los ficheros de poblado de la base de datos. Su formato es texto ascii con campos separados por tabuladores que tienen el formato que se indicó cuando se especificaron los registros.
- Ficheros acabados en: *_cargaNUM.txt*; son los ficheros de carga de cada servidor. Su formato es también en texto ascii con campos separados por tabuladores.
- Fichero acabado en: *_constantes.txt*; almacena el número de servidores para los que se ha generado carga, y también el número de transacciones totales a ejecutar entre todos los servidores.

.

Por defecto, cuando se genera un experimento con el mismo nombre de uno ya existente, se sobreescriben los ficheros de poblado y se eliminan los ficheros anteriores de carga antes de ser generados de nuevo. Al aplicar la opción `-c`, los ficheros de poblado se mantienen pero los ficheros de carga anteriores son eliminados antes de generar los nuevos.

4.3.5. Banco de pruebas del subsistema

Las pruebas del sistema generador se han centrado sobre todo en comprobar cada una de las cadenas de escritura a ficheros y las de lectura, para poder asegurar que lo mismo que se lee es lo que se ha escrito. No se han olvidado las pruebas de generación aunque dichas pruebas no se pudieron automatizar.

El banco de pruebas para el subsistema de generación, consta de las siguientes pruebas que ha superado.

- Para cada tipo de registro (almacén, zona, ...), se utilizó un programa que dado un poblado generado, lo lee del disco y lo vuelve a escribir en otro fichero. Después de su ejecución, se comprueba que ambos ficheros son iguales, y así se puede afirmar que lo mismo que se escribe es lo mismo que se lee.
- Para cada fichero de poblado generador, con la utilidad `wc -l`, se han contado la cantidad de líneas generadas, para comprobar si la cardinalidad pedida es la misma que la cardinalidad generada.
- Para los ficheros de carga se utilizó otro programa que mostraba por pantalla los datos que iba leyendo, de tal manera que se podía comprobar que lo escrito en disco y lo leído era exactamente igual.
- Para cada generador, se comprobó que realmente su salida era aleatoria entre los rangos que se le indicaban.

No es un banco de pruebas muy extenso, pero con esto se puede asegurar una correcta generación de datos, que más tarde serán utilizados por el programa de medida del rendimiento.

4.4. Subsistema TPC-C

Este subsistema es el encargado de ejecutar las pruebas de rendimiento para obtener una medición de dicho rendimiento, la unidad utilizada es el *tpmC*, que consiste en medir el número de transacciones de *nuevo pedido* que se realizan por segundo; dicho número estará estrechamente relacionado con la rapidez del sistema en resolver el resto de transacciones para poder atender a las de nuevo pedido.

Al igual que el sistema generador, este subsistema se maneja a través de un ejecutable que interactúa con el usuario. La secuencia de su función principal es la siguiente:

1. Depurar los parámetros de entrada: nombre del experimento y la ruta donde está ubicado; para poder acceder a los ficheros de carga y poblado.
2. Poblar la base de datos con los ficheros de poblado. Esta tarea no se tiene en cuenta para la medición.
3. Inicializar los sistemas de comunicación y sincronización con los servidores:
    - Una zona de memoria compartida donde los servidores irán poniendo sus estadísticas y que cuando finalicen será leída por el programa principal para hacer un resumen y presentar los resultados de rendimiento.
    - Una barrera para que todos los servidores comiencen a la vez.

- Un índice compartido para que cada servidor tenga un identificador único, partiendo del 0, y así puedan leer correctamente su fichero de carga. Para esto también hace falta un cerrojo que controle el acceso concurrente a dicho índice.

4. Lanzar los servidores, cuidando de que todos comiencen simultáneamente, y esperar a que finalicen sus tareas.
5. Cuando todos hayan finalizado, preparar un fichero con las estadísticas.

Los ficheros que componen este subsistema, junto con su funcionalidad, son los siguientes:

- `tpcc.parmacs.c` Programa principal que también implementa la funcionalidad básica del servidor.
- `transacciones.c` Implementa el trabajo que tiene que realizar un servidor para satisfacer cada transacción.
- `cargador.c` Realiza el poblado inicial de la base de datos, inicializando todas las estructuras de almacenamiento necesarias.

4.4.1. Lectura y carga del poblado

Para que los servidores comienzan a trabajar, necesitan que las estructuras de almacenamiento estén: creadas y pobladas. De esta tarea se encarga la función `cargador`, que rellena una estructura llamada `BBDD` con los árboles y las listas necesarias.

Estructuras de datos

Ya que los servidores necesitan trabajar con 9 almacenes, se ha dispuesto una estructura de datos que sirve como directorio para acceder a los mismos. La estructura se llama `struct_BBDD`. Esta estructura también proporciona un par de datos al programa principal, como son el número de servidores y el número de transacciones totales.

▷ struct struct_BBDD

Directorio de estructuras de almacenamiento de datos.

- `Arbol *productos;` Estructura de almacenamiento para los productos.
- `Arbol *almacenes;` Estructura de almacenamiento para los almacenes.
- `Arbol *existencias;` Estructura de almacenamiento para las existencias.
- `Arbol *zonas;` Estructura de almacenamiento para las zonas.
- `Arbol *clientes;` Estructura de almacenamiento para los clientes.
- `ListaEnlazada *historico;` Estructura de almacenamiento para el histórico de pedidos.
- `Arbol *pedidos;` Estructura de almacenamiento para los pedidos.
- `Arbol *lineaspedido;` Estructura de almacenamiento para las lineas de pedido.

- `Arbol *nuevospedidos;` Estructura de almacenamiento para los nuevos pedidos.
- `uint32_t numServ;` Número de servidores a ejecutar.
- `uint64_t numTran;` Número total de transacciones a ejecutar entre todos los servidores.

▷ **BBDD**

El tipo de dato BBDD se define como:
`typedef struct struct_BBDD BBDD;`
El programa principal pide al *cargador* que rellene esa estructura, el cargador lee los ficheros de poblado y constantes, y la rellena; más tarde el programa principal la utilizará para saber el número de servidores que lanzar y comprobar si se han ejecutado todas las transacciones; los servidores la utilizarán para acceder a los datos.

Interfaz de uso

La función de carga se declara de la siguiente manera:

▷ **cargador**

- Declaración: `void cargador(BBDD *bd,char *nombre_base)`
- Descripción: Carga el poblado de la base de datos y las constantes de funcionamiento. Utiliza las definiciones de `terminal.h` para conocer el formato de lectura de los ficheros de poblado.
- Parámetros:
  - `BBDD *bd` (salida): Directorio de estructuras de almacenamiento a inicializar y poblar.
  - `char *nombre_base` (entrada): Dado que los experimentos están en un directorio, y todos los ficheros de un experimento, tienen antepuesto el nombre del mismo. Esta cadena es la ruta común de acceso a los ficheros de un experimento. Por ejemplo, el experimento "uno" en el directorio "experimentos", hace que esta cadena tenga el valor: `experimentos/uno_`. Donde _ es el separador usado en todos los ficheros para separar el tipo de fichero del nombre del experimento.

Y el algoritmo que ejecuta es el descrito a continuación:

1. Para cada estructura de almacenamiento:
   a) Abrir el fichero de poblado.
   b) Crear la estructura de almacenamiento.
   c) Para cada línea del fichero de poblado, añadirla a la estructura de almacenamiento.
   d) Cerrar el fichero y pasar a la siguiente estructura.
2. Por último, cargar los parámetros de número de servidores y número de transacciones.

4.4.2. Servidor de transacciones

El servidor de transacciones es la función encargada de ejecutar la carga de transacciones de un fichero de carga, y de ir anotando que transacciones ha ejecutado. Para esto el servidor requiere del módulo `transacciones.c` que implementa la *lógica de negocio* del benchmark tpc-c para cada transacción.

Función servidor

La organización de estas tareas la realiza la función *servidor* localizada en el fichero `tpcc.parmacs.c`.

▷ **servidor**

- Declaración: `void servidor(void)`
- Descripción: Procesa transacciones a partir de un fichero de carga y anota cuántas transacciones ha realizado y de qué tipo son.
- Parámetros:
  - Ninguno. Utiliza variables globales del servidor
- Variables globales que utiliza.
  - `struct semaforos *sems;` Cerrojo de acceso al índice de servidores.
  - `int *indice;` Índice para asignar a cada servidor un identificador.
  - `uint64_t **estadisticas` Vector de vectores, donde existe un vector para almacenar la cantidad de transacciones de cada tipo por cada proceso.
  - `BBDD principal;` Directorio de acceso a las estructuras de almacenamiento.
  - `char nombre_base[512];` Base de la ruta de acceso al fichero de carga.
  - `struct barreras *bars;` Barrera de inicialización, para que todos los servidores comiencen a la vez.

El trabajo que realiza es parecido a la función de carga de poblado, pero en este caso sólo lee de su fichero de carga de trabajo.

1. Obtiene su índice de servidor, gracias al cerrojo de índice y a la variable compartida *índice*.
2. Accede al fichero de carga, mediante la ruta parcial *nombre_base*.
3. Utiliza la barrera para esperar que todos los servidores restantes hayan hecho los dos pasos anteriores.
4. Hasta finalizar el fichero de carga:
   a) Lee un número del fichero de carga, que es el tipo de transacción a realizar.
   b) Ejecuta dicha transacción, gracias al modulo `transacciones.c`
   c) Anota la transacción realizada.
5. Cierra el fichero de carga y finaliza.

Transacciones

El esfuerzo que realiza el sistema de pruebas, mediante este benchmark, está implementado en 5 transacciones. Dichas transacciones, recordemos, simulan la carga de trabajo de la base de datos de una empresa donde se realizan: pedidos, envíos, consultas de estado y stock, y pagos.

Para esta implementación se han *traducido* las especificaciones del apartado 3.2 (pag. 50), al lenguaje C, utilizando para el acceso a los datos, el interfaz del subsistema de almacenamiento.

Los 5 tipos de transacciones están implementados en el fichero `transacciones.c`, mediante 5 funciones.

▷ nuevo_pedido

- Declaración: `void nuevo_pedido(BBDD *bd,FILE *fich)`
- Descripción: Ejecuta la transacción de nuevo pedido, definida en el apartado 3.2.2 (pag. 52).
- Parámetros:
  - `BBDD *bd` (entrada): Directorio de acceso a las estructuras de almacenamiento.
  - `FILE *fich` (entrada): puntero al fichero de carga. A partir de este puntero se pueden leer los datos de entrada de la transacción.

▷ pago

- Declaración: `void pago(BBDD *bd,FILE *fich)`
- Descripción: Ejecuta la transacción de pago, definida en el apartado 3.2.3 (pag. 54).
- Parámetros:
  - `BBDD *bd` (entrada): Directorio de acceso a las estructuras de almacenamiento.
  - `FILE *fich` (entrada): puntero al fichero de carga. A partir de este puntero se pueden leer los datos de entrada de la transacción.

▷ estado_pedido

- Declaración: `void estado_pedido(BBDD *bd,FILE *fich)`
- Descripción: Ejecuta la transacción de consulta de estado de un pedido, definida en el apartado 3.2.4 (pag. 55).
- Parámetros:
  - `BBDD *bd` (entrada): Directorio de acceso a las estructuras de almacenamiento.
  - `FILE *fich` (entrada): puntero al fichero de carga. A partir de este puntero se pueden leer los datos de entrada de la transacción.

▷ **envío**

- Declaración: `void envío(BBDD *bd,FILE *fich);`
- Descripción: Ejecuta la transacción de envío de pedidos, definida en el apartado 3.2.5 (pag. 56).
- Parámetros:
  - `BBDD *bd` (entrada): Directorio de acceso a las estructuras de almacenamiento.
  - `FILE *fich` (entrada): puntero al fichero de carga. A partir de este puntero se pueden leer los datos de entrada de la transacción.

▷ **nivel_existencias**

- Declaración: `void nivel_existencias(BBDD *bd,FILE *fich)`
- Descripción: Ejecuta la transacción de consulta del nivel de existencias, definida en el apartado 3.2.6 (pag. 57).
- Parámetros:
  - `BBDD *bd` (entrada): Directorio de acceso a las estructuras de almacenamiento.
  - `FILE *fich` (entrada): puntero al fichero de carga. A partir de este puntero se pueden leer los datos de entrada de la transacción.

4.4.3. Programa tpcc

Como último paso y culminación de todos los subsistemas y módulos, se encuentra la aplicación tpcc, implementada en el fichero `tpcc.parmacs.c`. Esta aplicación ejecuta la función principal descrita al principio de esta sección sobre el subsistema TPC-C.

Se ha comentado el algoritmo de trabajo de dicha función, pero no se ha detallado el interfaz de comunicación con el usuario. Dicho interfaz es muy sencillo, ya que la aplicación tpcc sólo requiere dos parámetros para funcionar:

- Nombre del experimento: es decir, el prefijo de los ficheros del experimento.
- Directorio de trabajo del experimento: del cual leer los ficheros de carga de trabajo y de poblado, y en el cual generar un fichero de estadísticas.

Ejemplo:

```
./tpcc prueba1 mispruebas
```

Después de la ejecución del programa, en el directorio *mispruebas*, se habrá generado un fichero con las estadísticas. Las estadísticas se calculan con el siguiente algoritmo.

1. Se espera a que finalicen los procesos.
2. Una vez finalizados los servidores se comienza la realización de las estadísticas.
3. La zona de memoria para estadística se compone de tantos vectores de estadísticas como procesos servidores se hayan lanzado.

4. Un vector de estadísticas es un vector de 5 enteros de 64 bits que contabiliza la cantidad de cada uno de los 5 tipos de transacciones.
5. Se contabilizan todas las estadísticas: para cada vector de estadísticas se suman sus 5 valores y todas esas sumas se agregan a un contador de total de transacciones.
6. Se utiliza el vector 0 para hacer las estadísticas totales por tipo: se recorren todos los vectores y se suman todos los del tipo 0, tipo 1, ...
7. Se imprime a un fichero una estadística de porcentajes de transacciones por tipo.
8. Se imprime, además, un total de transacciones por segundo.
9. Y por último, se imprime el total de transacciones de nuevo pedido por minuto, el *tpmC*.

4.4.4. Banco de pruebas del subsistema

Dado que en este subsistema se prueba la aplicación final, para comprobar el correcto funcionamiento de todos los módulos hay que verificar casi todas las condiciones del análisis. En la mayoría de casos esta revisión no se ha podido automatizar, por lo que las revisiones manuales han sido relativamente pocas.

La lista de pruebas que se ha realizado al subsistema y que ha superado han sido las siguientes:

- Comprobar que todas las transacciones obtienen los datos correctamente de las estructuras de almacenamiento. Esta comprobación se realiza volcando las acciones de las transacciones por pantalla.
- Comprobar que cada transacción realiza su trabajo, esto se realiza de dos maneras:
  - De manera directa se muestra por pantalla cada una de las acciones de la transacción. Si por ejemplo, se realiza una búsqueda, se muestran los datos encontrados; cálculos realizados y otra información de lo que está sucediendo en un instante dado.
  - De manera indirecta, algunas transacciones almacenan datos que son más tarde utilizados por otras. Por ejemplo, un nuevo pedido, con el paso del tiempo, será enviado, luego la transacción de envío necesitará esos datos y se verán reflejados en la pantalla.
- Comprobación del sistema de manejo de parámetros de entrada. Debido a que se forma una ruta con los dos parámetros de entrada, hay que asegurar que se accede a todos los ficheros.

Capítulo 5

Conclusiones

Las conclusiones que se pueden obtener de cualquier desarrollo vienen condicionadas por los parámetros o acciones que se deseen observar, o por el conocimiento previo de la materia, que puede llevar a pasar por alto logros ya conocidos.

De la manera más objetiva posible, y siempre con los dos condicionantes anteriores, las conclusiones que se han obtenido en este proyecto son las siguientes.

5.1. Benchmark TPC-C

El benchmark TPC-C, como especificación, es completo y cumple todas las características que se le exigen a un buen análisis de rendimiento. A la hora de implementar todos los requerimientos hay que tener un especial cuidado en seguir las especificaciones correctamente, debido a que son muchos los requisitos y es muy fácil equivocarse.

Al ser muchos los requisitos, no todos han sido implementados; ya se comentó que los retardos producidos por el tiempo que el *usuario simulado* tarda en introducir los datos en la terminal no se tenían en cuenta. Luego no se han podido cumplir todos los requisitos del benchmark TPC-C.

La lista completa de requisitos a los que se ha hecho alusión y no se han alcanzado es la siguiente:

- Tiempos de retardo simulados.
- Transacciones fallidas: todas se anotan y no se comprueba la porcentaje de transacciones que fallan.
- Propiedades ACID de las transacciones: se intenta que los cambios consistentes, aislados, persistentes y las operaciones sean atómicas. Pero no se ha hecho un estudio detallado del cumplimiento de estas propiedades.
- No aprovecharse del modo de funcionamiento del benchmark: en esta implementación se ha aprovechado la situación de tal manera que se usa una lista enlazada para la tabla histórico sabiendo que sólo se van a añadir entradas sin clave al final.

5.2. Árboles de búsqueda

A la hora de buscar estructuras que permitan la búsqueda partiendo de una clave, se pueden encontrar múltiples alternativas: desde listas, tablas hash, índices multinivel estáticos, etc. [1]. A la hora de buscar una solución para el problema de implementar un sistema de almacenamiento para el benchmark TPC-C, se optó por una estructura muy empleada en implementaciones de índices: el árbol B+.

El problema de una estructura como el árbol B+, es que tiene gran cantidad de detalles en sus operaciones. Se puede encontrar mucha información en libros e Internet, pero en muy pocos lugares, por no decir ninguno, se detalla completamente una implementación que realmente funcione. Sólo incluyen una definición de su estructura, y una descripción superficial de la operación más sencilla: la inserción. El resto de detalles necesarios se dejan sin definir, *para que el lector trabaje*, etc.

Presentar un diseño completo y una implementación funcional de un árbol B+, ha sido uno de los mayores problemas, sobre todo debido a que el árbol es la base del almacenamiento del benchmark, y el funcionamiento del benchmark depende mucho del funcionamiento del árbol. Para este diseño se han tenido que realizar muchas actuaciones del tipo: prueba y error.

Otro problema es el que relaciona la estructura de datos con el lenguaje de programación. El lenguaje C no es uno de los más indicados para implementar estructuras de alto nivel, y mucho menos genéricas; aunque dada la necesidad de portar más tarde la aplicación a RSIM, es el único lenguaje disponible.

Como *conclusión* sobre los árboles B+, se puede obtener que son buenas estructuras de búsqueda, y que lo han demostrado en este benchmark, ya que después de su implementación, su uso y rendimiento ha sido satisfactorio.

Como contrapartida, y dado el lenguaje de programación, su implementación ha sido muy costosa en cuanto al tiempo empleado, y ha estado plagada de errores en casi todas las etapas del desarrollo; sobre todo debido a que ciertos aspectos del diseño no se *conocían* de antemano ya que no estaban documentados en casi ningún sitio, y parte de la implementación se realizó por "prueba y error".

5.3. Sincronización

La programación concurrente es siempre un aspecto ignorado en la programación de aplicaciones. Durante la carrera se tocan ciertos aspectos en la asignatura de *Sistemas operativos*, pero hasta que no se aplican dichos conocimientos en una aplicación real, no se tiene conciencia de todos los problemas que ello supone.

Si bien los paradigmas típicos del *lector-escritor*, o el *productor-consumidor*, son bien conocidos; su aplicación en un sistema concreto es el verdadero reto del analista y/o programador.

En este caso el lenguaje C, no es que provea de un sistema de sincronización, sino que se depende de la funcionalidad del sistema operativo. Para intentar abstraerse un poco de esto, se ha utilizado la colección de macros PARMACS.

Durante el desarrollo de esta aplicación, sobre todo a la hora de sincronizar las lecturas y escrituras en el subsistema de almacenamiento, se ha obtenido una experiencia muy valiosa en sistemas concurrentes.

Por otro lado, y centrándonos en la implementación de la sincronización del árbol B+ y del problema de los lectores escritores. La implementación de una solución al problema de lectores escritores no es algo complicado, el problema, como se comento anteriormente, es "sembrar" el código realizado para un solo proceso con dicha implementación, y que realmente se solucione el problema. Que

funcione no es la solución al problema, ya que en la programación concurrente, si no se parte de un diseño correcto, la solución puede funcionar N veces, pero no la vez $N + 1$.

Como *conclusión* en este aspecto, el sincronizar una estructura de datos de manera correcta y transparente a los procesos que la utilizan, es un trabajo tedioso que requiere múltiples revisiones y varios puntos de vista para lograr tener en cuenta todas las situaciones posibles.

5.4. Medidas de rendimiento

Uno de los objetivos de está implementación, es disponer de una aplicación que nos provea de una medida de rendimiento de un sistema concreto. Lo más normal en estos casos es buscar una aplicación ya construida, ejecutarla y obtener dicha medida; pero como ya se ha visto en los fundamentos teóricos, una medida de rendimiento, y por lo tanto un benchmark requieren cumplir ciertas normas.

Sin conocer nada sobre las medidas de rendimiento, los benchmark y los análisis de rendimiento, los resultados a los que se puede llegar después de leer unas especificaciones como las del TPC-C pueden ser totalmente erróneos. Las especificaciones del benchmark TPC-C están construidas en base a estas reglas, intentan cumplirlas, y por ello es muy necesario tener estos conceptos claros antes de entender el análisis.

Una vez se sepa qué se busca con una medida de rendimiento, sus fallos más comunes, las reglas, conocer algunas medidas existentes, etc. Se puede dar un paso más e intentar comprender el benchmark TPC-C para más tarde implementarlo.

La *conclusión* de este apartado, es que es necesario un conocimiento previo de la teoría de medidas de rendimiento, antes de analizar un benchmark para poder extraer correctamente todos los puntos, y no implementar un sistema con errores desde el principio.

5.5. Conclusión final

Los objetivos que se plantearon en un principio de han logrado, aunque con algunas restricciones como se comentó para el benchmark TPC-C. Se ha logrado:

- Obtener una aplicación con la que realizar medidas de rendimiento. Esta implementación es ya un benchmark más a añadir a la biblioteca de benchmarks disponibles tanto para el grupo de Arquitectura y Tecnología de Computadores, como para el resto de interesados en medidas de rendimiento.
- Utilizar y ampliar los conocimientos obtenidos a lo largo de la carrera. Se han afrontado problemas, y mediante soluciones ya conocidas o investigación de aquellos problemas no conocidos, se ha construido la solución.

Pero la implementación de las especificaciones del benchmark TPC-C es sólo el comienzo del futuro de la aplicación. Se buscaba obtener una implementación que más tarde pudiera ser ejecutada en RSIM; si bien se ha conseguido dicha implementación, queda el paso más importante: ejecutarla en RSIM. Solo ha sido el primer paso, los siguientes serán: adaptarlo a las peculiaridades de RSIM, completar la implementación ya que como se comentó hay requisitos incompletos; y por último, estudiar el comportamiento de la aplicación en RSIM, a la vez que se experimenta con los trabajos del grupo de Arquitectura y Tecnología de Computadores en cuanto a los protocolos de coherencia caché.

Apéndice A

Contenidos del CD-ROM adjunto

Por la parte interior de la tapa trasera se encuentra adjunto un CD-ROM con los contenidos expuestos en esta memoria. Para acceder a este CD-ROM es necesario un ordenador con lector de CD-ROM; no importando el sistema operativo que se use.

A.1. Acceso al CD-ROM

Acceso desde sistemas windows

1. Insertar el CD-ROM en el lector.
2. Acceder desde el icono *Mi Pc* del escritorio o desde el *Explorador de ficheros* a la unidad correspondiente.
3. Si el sistema windows lo permite, la unidad aparecerá etiquetada como *PFC-TPCC*.

Acceso desde sistemas Mac OS 10.X

1. Insertar el CD-ROM en el lector.
2. Acceder al *finder* o al escritorio.
3. Si se accede desde el finder, en la parte superior izquierda se habrá creado una nueva unidad que da acceso al CD-ROM.
4. Si se accede desde el escritorio, junto al icono de acceso al disco duro, se habrá creado otro icono de acceso al CD-ROM.
5. Por último, si se accede desde consola, ir al directorio `/Volumes/PFC-TPCC`

Acceso desde sistemas Linux/Unix

1. Insertar el CD-ROM en el lector.
2. Si el sistema linux/unix dispone de sistema de montado automático, en el escritorio gráfico o en los directorios `/mnt` o `/media` aparecerá el CD-ROM accesible.

3. Si el sistema no dispone de un sistema de montado automático, habrá que utilizar la orden mount para acceder a los contenidos del CD-ROM

4. En cualquier caso, teclear `mount` para conocer el estado de los dispositivos de almacenamiento montados en el sistema.

5. Más información sobre la orden mount tecleando `man mount`

A.2. Listado del contenido

Partiendo del directorio raíz del CD-ROM, encontramos:

- `/memoria.pdf` Una copia de esta memoria en formato PDF.
- `/fuentes/` Directorio donde se almacena el código fuente.
  - `tpcc.tar.gz` Código fuente del benchmark TPC-C.
- `/original/` Directorio donde se almacenan los originales en formato LaTeXde esta memoria.
  - `memoria.tar.gz` Originales en formato LaTeX.

Contenido del fichero `tpcc.tar.gz`

Para descomprimir y desempaquetar este fichero desde un terminal unix/linux, teclear para descomprimir en el directorio actual:
`tar -xzvf /ruta.al.fichero/tpcc.tar.gz`
Se obtendrá un directorio `tpcc/` donde se encuentra el código fuente, para más información sobre cómo compilar dicho código fuente, ver el manual de la aplicación (pag. 151).

Contenido del fichero `memoria.tar.gz`

Para descomprimir y desempaquetar este fichero desde un terminal unix/linux, teclear para descomprimir en el directorio actual:
`tar -xzvf /ruta.al.fichero/memoria.tar.gz`
Esta memoria ha sido escrita en LaTeX, y además ha sido orientada hacia la obtención de un documento PDF, por lo que es imposible utilizar la orden `latex` para generar un fichero PS.

Generación de la memoria

Para generar un fichero PDF hay que teclear, en el directorio donde se haya desempaquetado la memoria:

```
]$ pdflatex memoria
]$ bibtex memoria1
]$ bibtex memoria2
]$ pdflatex memoria
]$ pdflatex memoria
```

Manejo de ficheros PDF

La memoria se entrega en formato PDF (*Portable Document Format*); y para acceder al contenido se necesita un visor de ficheros PDF. Una lista de las diferentes posibilidades según el sistema operativo, es la siguiente:

- *Adobe Acrobat Reader*: para los sistemas operativos Microsoft Windows, IBM OS/2 Warp, Palm OS, Pocket PC, Symbian OS, Linux, Sun Solaris, IBM Aix, HP-UX y Mac OS 7.x – 10.x . Disponible en: `http://www.adobe.es/products/acrobat/readstep2.html`.
- *xpdf*: para los sistemas operativos con el sistema de ventanas X Window. Disponible en: `http://www.foolabs.com/xpdf/`.
- *kpdf*: forma parte del entorno de escritorio KDE. Disponible en `http://www.kde.org/`; más información en: `http://kpdf.kde.org/`.
- *Vista Previa*: disponible en el sistema operativo Mac OS 10.x . Más información en: `http://www.apple.com/es/macosx/features/pdf/`.

Apéndice B

Manual de la aplicación

La implementación del benchmark TPC-C se entrega en código fuente, por lo que para poder ejecutarla, hace falta compilarla. Además de las herramientas necesarias de desarrollo, esta implementación tiene otros requerimientos, no ya del software sino del hardware donde se va a ejecutar.

B.1. Requisitos mínimos

De hardware o físicos:

- Capacidad en disco: mínimo 80 MiBytes. Debido a los ficheros de carga y poblado, aproximadamente se necesitan:
  - Unos 75 MiBytes por almacén en lo referente al poblado de la base de datos.
  - Unos 750 Kibytes por cada 10.000 transacciones programadas.
- Memoria RAM: mínimo 384 MiBytes. Dado que todo el almacenamiento se produce en memoria RAM, se necesitan unos 128 MiBytes extra por cada 100.000 transacciones que se quieran procesar.

Software necesario:

- Compilador de C con preprocesador; preferiblemente *GNU C Compiler* (gcc).
- Preprocesador de macros *GNU m4*.
- Herramienta para la generación automática de ejecutables *GNU Make*
- Soporte del sistema operativo para *Hilos POSIX* (pthreads).

Sistemas donde se ha probado la aplicación:

- Principalmente en *Darwin* (Mac OS 10.4.2).
- También en linux:
  - Linux Kernel 2.6.x
  - Distribuciones probadas: gentoo y debian.

- Sólo compilación y funcionamiento del generador en Solaris 8

Incompatibilidades:

- Con el sistema operativo *FreeBSD*: el soporte de hilos POSIX es nulo.
- Con el sistema operativo *OpenBSD*: no permite el acceso a un mismo cerrojo desde dos hilos.

B.2. Instalación

Preparación del código fuente

Para la instalación de la aplicación hace falta obtener el fichero que contiene el código fuentes del CD-ROM adjunto a esta memoria (ver pag. A). Una vez tengamos el fichero `tpcc.tar.gz` se procederá a descomprimirlo y desempaquetarlo mediante la orden: $tar - xzvf/ruta.al.fichero/tpcc.tar.gz$

Ejemplo:

```
~$ tar -xzvf tpcc.tar.gz
tpcc/
tpcc/abm_test.parmacs.c
tpcc/arbolbmas-priv.h
tpcc/arbolbmas.c
tpcc/arbolbmas.h
tpcc/arbolbmas.parmacs.c
tpcc/basicgen.c
tpcc/basicgen.h
tpcc/cargador.c
tpcc/cargador.h
tpcc/comparadores.c
tpcc/comparadores.h
tpcc/debug.h
tpcc/generador.c
tpcc/generador.h
tpcc/le_test.c
tpcc/listaenlazada.parmacs.c
tpcc/listaenlazada.parmacs.h
tpcc/macros/
tpcc/macros/c.m4.posix.mutex
tpcc/Makefile
tpcc/readwrite.parmacs.c
tpcc/readwrite.parmacs.h
tpcc/registros.h
tpcc/terminal.c
tpcc/terminal.h
tpcc/tpcc.parmacs.c
tpcc/transacciones.c
tpcc/transacciones.h
~$
```

Compilación

Una vez descomprimido y desempaquetado el código fuente de la aplicación, para generar los ejecutables, se accede al directorio `tpcc` y se teclea la orden `make`. Dado que se utiliza el programa *GNU make*; y algunos sistemas que integran un make de otro fabricante, para ejecutar el *GNU make* hay que utilizar la orden `gmake`.

Ejemplo:

```
~/tpcc$ make
m4 macros/c.m4.posix.mutex readwrite.parmacs.h > readwrite.h
m4 macros/c.m4.posix.mutex listaenlazada.parmacs.h > listaenlazada.h
m4 macros/c.m4.posix.mutex tpcc.parmacs.c > tpcc.c
gcc -g -O0   -c -o tpcc.o tpcc.c
gcc -g -O0   -c -o arbolbmas.o arbolbmas.c
gcc -g -O0   -c -o cargador.o cargador.c
gcc -g -O0   -c -o comparadores.o comparadores.c
m4 macros/c.m4.posix.mutex listaenlazada.parmacs.c > listaenlazada.c
gcc -g -O0   -c -o listaenlazada.o listaenlazada.c
gcc -g -O0   -c -o transacciones.o transacciones.c
gcc -g -O0   -c -o basicgen.o basicgen.c
m4 macros/c.m4.posix.mutex readwrite.parmacs.c > readwrite.c
gcc -g -O0   -c -o readwrite.o readwrite.c
gcc -lpthread  tpcc.o arbolbmas.o cargador.o comparadores.o
listaenlazada.o transacciones.o basicgen.o readwrite.o   -o tpcc
gcc -g -O0   -c -o generador.o generador.c
gcc -g -O0   -c -o terminal.o terminal.c
gcc -lpthread  generador.o basicgen.o terminal.o   -o generador
m4 macros/c.m4.posix.mutex abm_test.parmacs.c > abm_test.c
gcc -g -O0   -c -o abm_test.o abm_test.c
gcc -lpthread  abm_test.o arbolbmas.o readwrite.o   -o abm_test
gcc -g -O0   -c -o le_test.o le_test.c
gcc -lpthread  le_test.o listaenlazada.o   -o le_test
rm tpcc.c listaenlazada.c abm_test.c readwrite.c
~/tpcc$
```

Una vez terminada la compilación del código, se habrán generado 4 ficheros ejecutables:

- `generador`: Generador de experimentos (carga y poblado).
- `tpcc`: Aplicación que realiza la prueba de rendimiento TPC-C.
- `abm_test`: Pruebas realizadas al subsistema de almacenamiento, módulo del árbol B+.
- `le_test`: Pruebas realizadas al subsistema de almacenamiento, módulo de la lista enlazada.

B.3. Ejecución

Para ejecutar una medición de rendimiento, hacen falta dos pasos:

1. Mediante el programa `generador`, crear un experimento con los ficheros de poblado y de carga.
2. Ejecutar el benchmark con la orden `tpcc`, indicándole el nombre del experimento y el directorio donde se encuentra.

B.3.1. Generador de carga y poblado

Para obtener el experimento utilizaremos el programa `generador`, si tecleamos `./generador` o `./generador -h` en el directorio donde hemos compilado los ejecutables nos saldrá el siguiente mensaje de información sobre el modo de uso del programa.

```
Modo de uso:
   ./generador [<opciones>] <dir>

Opciones:
   -h          Este mensaje.
   -c          No genera el poblado de la BD (Útil para cambiar la carga)
   -a número   Cantidad de almacenes.............(Por defecto: 1)
   -t número   Cantidad total de transacciones.....(Por defecto: 100)
   -s número   Cantidad de procesos servidor.......(Por defecto: 1)
   <dir>       Directorio destino del experimento...(Obligatorio)
```

Descripción de las opciones

- `-h` Muestra un mensaje de ayuda y termina la ejecución del programa.
- `-c` (opcional) No genera los ficheros correspondientes al poblado de la base de datos. Esta opción es útil cuando se tiene una carga concreta y deseamos cambiar: la cantidad de transacciones o la cantidad de procesos servidor; sin crear de nuevo el poblado.
- `-a <número>` (opcional, por defecto 1) Número total de almacenes en el poblado de la base de datos. Este número es la unidad básica de poblado, debido a que por cada almacén existen 10 zonas, por cada zona 3000 clientes y pedidos, etc; si aumentamos el número de almacenes, se multiplican estas últimas cardinalidades por dicho número. El número máximo de almacenes que se pueden indicar es: 4.294.967.295
- `-t <número>` (opcional, por defecto 100) Número total de transacciones a generar entre todos los servidores. El número máximo de transacciones que se pueden generar es: 18.446.744.073.709.551.615
- `-s <número>` (opcional, por defecto 1) Cantidad de servidores para los que generar carga. El número total de transacciones se divide entre el total de servidores, y el resultado es número de transacciones que generará por cada servidor. El número máximo de servidores que se pueden indicar es: 4.294.967.295

Ejemplos de uso

Generación de datos de entrada para 1 almacén y 2 servidores, dando el nombre *prueba* al experimento.

```
tpcc$ ./generador -s 2 prueba
Ejecutando ./generador - Generando carga para
        * 1 Almacenes
        * 100 Transacciones
        * 1 Almacenes
        * 100 Transacciones
        * 2 Servidores
        * Directorio de salida: prueba

-> Generando 100000 productos
-> Generando 1 almacenes
-> Generando 100000 entradas de stock para el almacén Nº 0
-> Generando 10 zonas para el almacén Nº 0
-> Generando 3000 clientes para la zona Nº 0 del almacén Nº 0
```

```
-> Generando 3000 pedidos para la zona Nº 0 del almacén Nº 0
-> Generando 900 nuevos pedidos (partiendo de 2100) para la zona Nº 0 del
almacén Nº 0
-> Generando 3000 clientes para la zona Nº 1 del almacén Nº 0
-> Generando 3000 pedidos para la zona Nº 1 del almacén Nº 0
-> Generando 900 nuevos pedidos (partiendo de 2100) para la zona Nº 1 del
almacén Nº 0
-> Generando 3000 clientes para la zona Nº 2 del almacén Nº 0
-> Generando 3000 pedidos para la zona Nº 2 del almacén Nº 0
-> Generando 900 nuevos pedidos (partiendo de 2100) para la zona Nº 2 del
almacén Nº 0
-> Generando 3000 clientes para la zona Nº 3 del almacén Nº 0
-> Generando 3000 pedidos para la zona Nº 3 del almacén Nº 0
-> Generando 900 nuevos pedidos (partiendo de 2100) para la zona Nº 3 del
almacén Nº 0
-> Generando 3000 clientes para la zona Nº 4 del almacén Nº 0
-> Generando 3000 pedidos para la zona Nº 4 del almacén Nº 0
-> Generando 900 nuevos pedidos (partiendo de 2100) para la zona Nº 4 del
almacén Nº 0
-> Generando 3000 clientes para la zona Nº 5 del almacén Nº 0
-> Generando 3000 pedidos para la zona Nº 5 del almacén Nº 0
-> Generando 900 nuevos pedidos (partiendo de 2100) para la zona Nº 5 del
almacén Nº 0
-> Generando 3000 clientes para la zona Nº 6 del almacén Nº 0
-> Generando 3000 pedidos para la zona Nº 6 del almacén Nº 0
-> Generando 900 nuevos pedidos (partiendo de 2100) para la zona Nº 6 del
almacén Nº 0
-> Generando 3000 clientes para la zona Nº 7 del almacén Nº 0
-> Generando 3000 pedidos para la zona Nº 7 del almacén Nº 0
-> Generando 900 nuevos pedidos (partiendo de 2100) para la zona Nº 7 del
almacén Nº 0
-> Generando 3000 clientes para la zona Nº 8 del almacén Nº 0
-> Generando 3000 pedidos para la zona Nº 8 del almacén Nº 0
-> Generando 900 nuevos pedidos (partiendo de 2100) para la zona Nº 8 del
almacén Nº 0
-> Generando 3000 clientes para la zona Nº 9 del almacén Nº 0
-> Generando 3000 pedidos para la zona Nº 9 del almacén Nº 0
-> Generando 900 nuevos pedidos (partiendo de 2100) para la zona Nº 9 del
almacén Nº 0
-> Generando Carga para el servidor 0 - transacciones 50
-> Generando Carga para el servidor 1 - transacciones 50
```

Basándonos en el ejemplo anterior, se conserva el poblado generado, pero se cambia el número total de transacciones a 1.000 y el número de servidores a 5.

```
tpcc$ ./generador -c -t 1000 -s 5 prueba
Ejecutando ./generador - Generando carga para
        * 0 Almacenes
        * 1000 Transacciones
        * 5 Servidores
        * Directorio de salida: prueba

-> El directorio "prueba" existe, borrando carga.
```

```
         * Borrando prueba_carga0.txt (0)
         * Borrando prueba_carga1.txt (0)
-> Generando Carga para el servidor 0 - transacciones 200
-> Generando Carga para el servidor 1 - transacciones 200
-> Generando Carga para el servidor 2 - transacciones 200
-> Generando Carga para el servidor 3 - transacciones 200
-> Generando Carga para el servidor 4 - transacciones 200
```

B.3.2. Benchmark tpcc

Una vez que dispongamos de uno o más experimentos con los que trabajar, se podrá lanzar la aplicación de medida de rendimiento `tpcc`; para que, basándose en un experimento concreto, realice una ejecución de la lógica de negocio que implementa la aplicación `tpcc`. Después de la ejecución, obtendremos un fichero con los resultados de la medición.

Parámetros de ejecución

A diferencia del generador, esta aplicación no muestra ningún mensaje por pantalla (ver apartado B.4 en pag. 157 para cambiar esto). El modo de ejecutar el benchmark es el siguiente:
`./tpcc nombre_experimento directorio_experimento`

Donde:

- `nombre_experimento` Corresponde al prefijo que llevan todos los ficheros de un experimento.
- `directorio_experimento` Lugar donde encontrar los ficheros del experimento.

Normalmente estos dos parámetros suelen ser iguales, ya que el generador cuando se le índice el nombre de un experimento, crea un directorio con ese nombre, y pone de prefijo a todos los ficheros el nombre del directorio.

Ejemplo de uso

Basándonos en los datos creados en el ejemplo anterior del generador, ejecutamos el benchmark tpcc.

```
tpcc $ ./tpcc prueba prueba
tpcc $
```

Para ver los resultados de la medición:

```
tpcc $ cat prueba/prueba_estadisticas.txt
Estadisticas generadas: Sun Sep 18 18:46:42 2005

Total transacciones: 1000 (1000 programadas)
        * Nuevo Pedido 43.3000% (433)
        * Pago 43.6000% (436)
        * Estado de Pedido 4.5000% (45)
        * Envío 4.1000% (41)
        * Nivel de Existencias 4.5000% (45)
Transacciones por segundo: 24.3902
(1000 transacciones en 41 segundos)
tpmC - 633.6585 (transacciones de nuevo pedido por minuto)
```

B.4. Configuración detallada

Dentro de todo el sistema que da lugar a la implementación benchmark TPC-C, hay parámetros para variar el comportamiento del benchmark. Estos parámetros no están accesibles mediante opciones de la línea de órdenes, sino que se encuentran establecidos en definiciones dentro de los ficheros del código fuente. Veamos que parámetros de configuración hay dentro de cada fichero

B.4.1. Salida por pantalla

Como ya se comentó en la sección 4.1.3 (pag. 90), existe un sistema de ayuda a la detección de errores que informa en pantalla sobre lo que está sucediendo en cada momento. Para activar la salida por pantalla no hay más que compilar el modulo en el que deseemos activar dicha salida con la constante `DEBUG` definida. Esto se puede hacer de manera global o local.

De manera global

Modificando el fichero `Makefile` y cambiando la definición de la variable CFLAGS. Si normalmente esta variable esta definida como `CFLAGS=-g` , ahora pasaría ser `CFLAGS=-g -DDEBUG`.

De manera local

O por cada módulo independientemente. Para poder activar la salida en un módulo concreto, hay que definir en dicho módulo la opción `DEBUG` editándolo y añadiendo `#define DEBUG` antes de la línea `#include "debug.h"`.

Ejemplo de activación

Suele ser muy útil activar la salida por pantalla en el módulo `tpcc.parmacs.c`, ya que así recibiremos información de los servidores que indican en que transacción están trabajando.

Continuando con el ejemplo de *prueba* anterior, activaremos la opción `DEBUG` en el fichero `tpcc.parmacs.c`, y teclearemos la orden `make` para recompilar dicho módulo.

```
tpcc $ vim tpcc.parmacs.c
tpcc $ make
m4 macros/c.m4.posix.mutex tpcc.parmacs.c > tpcc.c
gcc -g -O0   -c -o tpcc.o tpcc.c
En el fichero incluido de tpcc.c:8:
debug.h:21:3: aviso: #warning ¡¡Has activado DEBUG para este modulo!!
gcc -lpthread  tpcc.o arbolbmas.o cargador.o comparadores.o
listaenlazada.o transacciones.o basicgen.o readwrite.o   -o tpcc
rm tpcc.c
```

Al compilar con la opción `DEBUG` activada, el compilador nos dará un aviso; no supone ningún error, sólo es un aviso ya que en determinados módulos puede suponer un descenso importante en el rendimiento de la aplicación. Si ejecutamos ahora el benchmark, obtendremos la siguiente salida.

```
 $ ./tpcc prueba prueba
[tpcc] Inicio - Nombre base: "prueba/prueba_"
[tpcc] Inicio - Poblando la base de datos, esto llevará un buen rato
[tpcc] Numero servidores:5 Numero transacciones:1000
[0][Servidor] Índice: 0
[0][Servidor] Fichero entrada: "prueba/prueba_carga0.txt"
[1][Servidor] Índice: 1
```

```
[1][Servidor] Fichero entrada: "prueba/prueba_carga1.txt"
[2][Servidor] Índice: 2
[2][Servidor] Fichero entrada: "prueba/prueba_carga2.txt"
[3][Servidor] Índice: 3
[3][Servidor] Fichero entrada: "prueba/prueba_carga3.txt"
[4][Servidor] Índice: 4
[4][Servidor] Fichero entrada: "prueba/prueba_carga4.txt"
[4][Servidor] Comenzando (Barrera de inicio superada)
[4][Servidor] Transacción de Estado de pedido
[4][Servidor] Transacción de Nuevo pedido
...
...
...
[3][Servidor] Trabajo finalizado
[3][Servidor] Total transacciones: 200
        * Nuevo Pedido 43.5000% (87)
        * Pago 43.0000% (86)
        * Estado de Pedido 4.5000% (9)
        * Envío 4.5000% (9)
        * Nivel de Existencias 4.5000% (9)
[1][Servidor] Transacción de Pago
[1][Servidor] Transacción de Estado de pedido
[1][Servidor] Transacción de Nuevo pedido
[1][Servidor] Transacción de Pago
[1][Servidor] Transacción de Nuevo pedido
[1][Servidor] Transacción de Nuevo pedido
[1][Servidor] Trabajo finalizado
[1][Servidor] Total transacciones: 200
        * Nuevo Pedido 43.0000% (86)
        * Pago 44.0000% (88)
        * Estado de Pedido 4.5000% (9)
        * Envío 4.0000% (8)
        * Nivel de Existencias 4.5000% (9)
[tpcc] Limpiando almacén de datos
[limpiar] Destruyendo medio de almacenamiento número 0
[limpiar] Destruyendo medio de almacenamiento número 1
[limpiar] Destruyendo medio de almacenamiento número 2
[limpiar] Destruyendo medio de almacenamiento número 3
[limpiar] Destruyendo medio de almacenamiento número 4
[limpiar] Destruyendo medio de almacenamiento número 5
[limpiar] Destruyendo medio de almacenamiento número 6
[limpiar] Destruyendo medio de almacenamiento número 7
[limpiar] Destruyendo medio de almacenamiento número 8
[tpcc] Limpiando memoria compartida
```

Como se puede observar, con solo activar esta opción al más alto nivel, la cantidad de datos que se obtiene es muy alta.

B.4.2. Opciones de la aplicación tpcc

Aparte de la opción `DEBUG` existen otras dos constantes que alteran su funcionamiento. Estas constantes hay que definirlas en el fichero `tpcc.parmacs.c`.

- __ESTADISTICAS__ Por defecto activada. Su activación controla la generación de las estadísticas: si no está definida, no se generan estadísticas del rendimiento (fichero de estadísticas).
- __LIMPIAR__ Por defecto activada. En los sistemas donde PARMACS permita liberar memoria mediante G_FREE, al activar esta opción se indica que la memoria de todas las estructuras de almacenamiento sea liberada. Esta opción también es útil para comprobar si existe alguna corrupción dentro las estructuras, ya que al ordenar su destrucción si existen errores internos, no se producirán errores en la función de limpieza.

B.4.3. Opciones del generador

En el fichero `generador.h` se pueden alterar los nombres de los ficheros de salida para el poblado y las constantes, y también se puede alterar la cardinalidad de los diferentes elementos del poblado.

Para los ficheros tenemos las siguientes constantes:

- `FICHERO_PRODUCTOS` Por defecto: "productos.txt".
- `FICHERO_ALMACENES` Por defecto: "almacenes.txt".
- `FICHERO_EXISTENCIAS` Por defecto: "existencias.txt".
- `FICHERO_ZONAS` Por defecto: "zonas.txt".
- `FICHERO_CLIENTES` Por defecto: "clientes.txt".
- `FICHERO_HISTORICO` Por defecto: "historico.txt".
- `FICHERO_PEDIDOS` Por defecto: "pedidos.txt".
- `FICHERO_LINEASPEDIDO` Por defecto: "lineaspedido.txt".
- `FICHERO_NUEVOSPEDIDOS` Por defecto: "nuevospedidos.txt".
- `FICHERO_CONSTANTES` Por defecto: "constantes.txt".

Y para las cardinalidades del poblado:

- `CARD_ALMACEN` Por defecto: 1
- `CARD_ZONA` Por defecto: 10
- `CARD_CLIENTE` Por defecto: 30.000
- `CARD_PEDIDO` Por defecto: `CARD_CLIENTE`
- `CARD_NUEVOPEDIDO` Por defecto: `CARD_CLIENTE`*0.3
- `CARD_PRODUCTO` Por defecto: 100.000
- `CARD_EXISTENCIAS` Por defecto: `CARD_PRODUCTO`
- `CARD_TRANSACCION` Por defecto: 100
- `CARD_SERVIDOR` Por defecto: 1

B.4.4. Opciones del árbol B+

Como ya se comentó el árbol B+ se compone de nodos internos y hoja, y dichos nodos tienen una cierta capacidad de claves, y también un límite mínimo de claves. Se pueden alterar estos parámetros en el fichero `arbolbmas.h`.

- `NODE_SIZE` Capacidad de claves del nodo. En cuanto a los enlaces, un nodo hoja tendrá el mismo número de enlaces que de claves y un nodo interno tendrá un enlace más.
- `NODE_SIZE_MIN` Límite de capacidad mínima en claves de un nodo.

Y para estos tamaños se deben de cumplir 3 reglas:

- El tamaño mínimo de un nodo es 3.
- El límite de capacidad mínimo de un nodo es como poco 1.
- Entre el tamaño máximo y mínimo se tiene que cumplir la fórmula: $min \leq (max)/2$

Apéndice C

Notas sobre las referencias web

A lo largo de esta memoria y en la propia sección de *otras referencias*, se pueden encontrar múltiples direcciones de páginas web. Dichas direcciones son un método de encontrar más información sobre el punto en el que han sido indicadas, ya que no todo el conocimiento ni la información se encuentra en libros o artículos.

Dado que el contenido de las web cambia con el tiempo, a día **18 de Septiembre de 2005**, se han revisado todos y cada uno de los enlaces y se puede asegurar que en esta fecha contienen la información a la que se hace referencia.

Bibliografía y referencias

I. Bibliografía

[1] R. A. Elmasri and S. B. Navathe. *Fundamentos de Sistemas de Bases de Datos*. Prentice Hall, 2002.

[2] M. Flynn and K. W. Rudd. Parallel architectures. In *ACM Computing Surveys, Vol. 28, No. 1*, pages 67–70, March 1996.

[3] R. Jain. *Art of Computer Systems Performance Analysis Techniques For Experimental Design Measurements Simulation And Modeling*. John Wiley & Sons, 1991.

[4] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 181–190, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[5] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[6] D. Llanos, B. Sahelices, and A. De Dios. Reducing the Replacement Overhead on COMA Protocols for Workstation-based Architectures. *Lecture Notes in Computer Science*, pages 550–557, Sept. 2000.

[7] E. L. Lusk and R. A. Overbeek. Use of monitors in fortran: a tutorial on the barrier, self-scheduling do-loop, and askfor monitors. In *on Parallel MIMD computation: HEP supercomputer and its applications*, pages 367–411, Cambridge, MA, USA, 1985. Massachusetts Institute of Technology.

[8] R. A. Overbeek and J. Boyle. *Portable Programs for Parallel Processors*. Saunders College Publishing, Philadelphia, PA, USA, 1987.

[9] V. Pai. Rsim: An execution-driven simulator for ilp-based shared-memory multiprocessors and uniprocessors. In *3rd Workshop on Computer Architecture Education*, 1997.

[10] B. Parhami. *Introduction to Parallel Processing*. Kluwer Academic Publishers, 2002.

[11] J. Rao and K. A. Ross. Making b + -trees cache conscious in main memory. In *SIGMOD Conference*, pages 475–486, 2000.

[12] B. Sahelices, D. Llanos, and A. De Dios. Exploiting parallelism in a network of workstations. *ACM Computer Architecture News*, June 2000.

[13] B. Sahelices, D. Llanos, and A. De Dios. Simulación de Arquitecturas De Memoria en Multicomputadores. In *XI Jornadas de Paralelismo*, Sept. 2000.

[14] W. Stallings. *Sistemas Operativos*. Prentice Hall, forth edition, 2002.

[15] A. S. Tanenbaum and A. S. Woodhull. *Sistemas Operativos, Diseño e Implementación*. Prentice Hall, second edition, 1998.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

II. Otras referencias

[17] J. Bastida. Apuntes de arquitecturas avanzadas, 2004.

[18] F. P. Carpio. Apuntes de arquitecturas avanzadas, 2002.

[19] Introducción al lenguaje c. http://www.cpr2valladolid.com/tecno/cyr_01/control/lengua_C/intro.htm.

[20] M. M. Marqués. Apuntes de Ficheros y Bases de Datos. http://www3.uji.es/~mmarques/f47/apun/node27.html.

[21] Portable Applications Standards Committee of the IEEE Computer Society. http://www.open-std.org/jtc1/sc22/WG15/.

[22] V. Raatikka. Cache-conscious index structures for main-memory databases. Master's thesis, Universidad de Helsinki, 2004.

[23] Transaction processing performance council. http://www.tpc.org/.