ZODB

Base de Datos de Objetos

Universidad Tecnológica Nacional
Facultad Regional Paraná

Jerónimo L. Barraco Mármol
JeroBarraco@Yahoo.com.ar
http://Nande.com.ar

Introducción:

Las bases de datos de objetos son un sistema de administración de base de datos que permiten el modelado y la creación de datos como objetos. Esto incluye algún soporte para clases de objetos, herencia de propiedades de clases y métodos.

La finalidad principal de este trabajo es analizar el estado del arte en la base de datos orientada a objetos ZODB (Zope Object Data Base).

Para el correcto desarrollo y comprensión de los temas, dada la naturaleza de los mismos, se requieren conocimientos básicos generales sobre sistemas de bases de datos, programación orientada a objetos y el lenguaje de programación Python.

La principal motivación de este trabajo es que, a pesar de la aparente novedad de los sistemas de bases de datos de objetos, éstos han estado funcionando hace mucho tiempo a nivel empresarial, sin embargo, hay relativamente poco conocimiento de su utilización.

Por ello se pretende verificar las ventajas planteadas e identificar nuevas, en contraposición a los sistemas de bases de datos relacionales, comprobar el estado de adopción de la tecnología y analizar el posible impacto de su utilización en las metodologías de desarrollo actuales.

Para lograr este fin, se procederá con una introducción sobre los tipos más conocidos de bases de datos, a fin de entender las particularidades de las mismas. Luego, una rápida comparación entre las bases de datos más utilizadas en la actualizad con las bases de datos orientadas a objetos, a fin de comprender mejor su comportamiento general.

Seguidamente se estudiará el objeto principal del documento, ZODB, explicando sus características principales y como se debe proceder en un caso básico y típico de trabajo. Explicando sus mayores funcionalidades como, instalación; acciones básicas de creación, modificación y eliminación de datos; transacciones; historicidad y BLOBs. Completando con la descripción de algunas utilidades particulares de ZODB.

Para finalizar, una breve conclusión sobre el impacto de ZODB sobre las tareas de desarrollo precisadas en la actualidad.

Dado que ZODB está ligado por naturaleza al lenguaje de programación Python, se intentará abrir el panorama de bases de datos de objetos en la actualidad con una explicación básica y rápida de una solución similar la cual está desarrollada para dos de los lenguajes más utilizados actualmente para el desarrollo profesional de software.

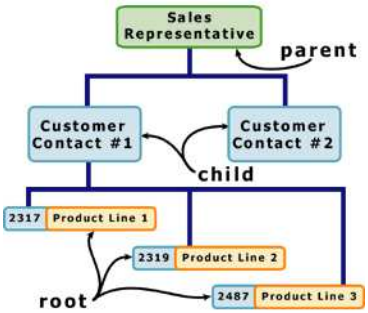
Bases de datos

Una base de datos es una colección de datos organizados para uno o más propósitos, usualmente en un formato digital. Los datos están típicamente organizados para modelar los aspectos relevantes de la realidad. Por ejemplo, la disponibilidad de las habitaciones de un hotel, de manera que permita realizar procesos requiriendo esta información (por ejemplo, encontrar un hotel disponible).

Tipos de bases de datos

Existen cuatro mayores tipos de estructuras en los sistemas gestores de bases de datos: Jerárquico, de red, relacionales y orientadas a objetos. También existen otros modelos menos pertinentes a este análisis como: Modelo semiestructurado, asociativo y Entidad – Atributo - Valor.

Modelo Jerárquico



El modelo de datos jerárquico organiza los datos en una estructura de árbol.

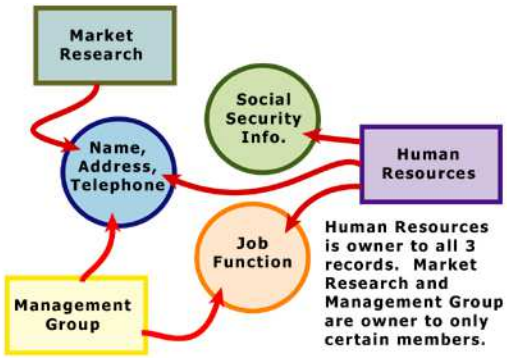
Existe una jerarquía de padre e hijo entre los datos. Esta estructura implica que en un registro puede existir información repetida, generalmente dentro de los segmentos de datos hijos. Los datos están agrupados en una serie de registros, que tienen un conjunto de valores.

En una base de datos jerárquica, la relación padre-hijo es de uno a muchos. Esto restringe los datos hijos a un solo padre. Las DBMSs jerárquicas se hicieron populares a finales de 1960, con la introducción del Information Management System (IMS) de IBM hasta el 1970.

Las bases de datos jerárquicas, comúnmente usadas en mainframes, han existido por un largo tiempo. Es uno de los métodos más antiguos de organizar y almacenar datos, y aún es utilizado por algunas organizaciones, por ejemplo, para reservaciones de viajes.

La ventaja de las bases de datos jerárquicas es que pueden ser accedidas y actualizadas rápidamente gracias a que su estructura de árbol y las relaciones entre registros son definidas con anterioridad. De igual manera, esto trae sus desventajas. Este tipo de estructuras de base de datos solo permite que cada hijo tenga un solo padre, y las relaciones o vínculos entre hijos no están permitidos. Además las bases de datos jerárquicas son tan rígidas en su diseño que agregar un nuevo campo o registro requiere que se redefina la base de datos por completo.

Modelo de Red



La popularidad del modelo de datos de red coincide con la popularidad del modelo jerárquico. Algunos datos son modelados de forma más natural con más de un padre por hijo. De esta manera, el modelo de red permite modelar relaciones muchos-a-muchos.

En 1971, la Conferencia en Lenguajes de Sistemas de Datos

(Conference on Data Systems Languages (CODASYL)) definió formalmente el modelo de datos de red. El modelo de red CODASYL está basado en el modelo matemático de Set.

Un set consiste de un registro dueño, un nombre, y un registro miembro. Un registro miembro puede tener dicho rol en más de un set, de ahí nace la posibilidad del concepto de multi-padre. Un registro dueño puede también ser miembro o dueño en otro set.

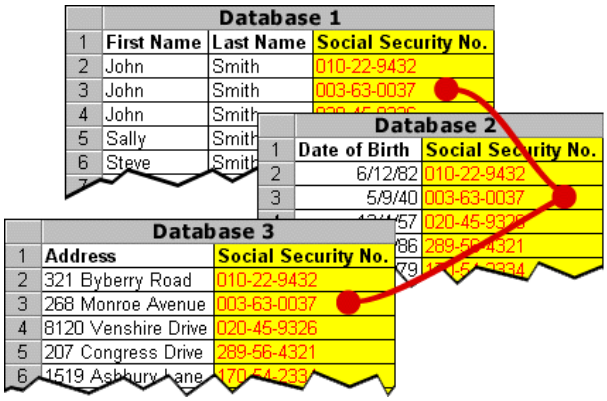
El modelo de datos es una sencilla red, y los enlaces y registros de intersección (llamados registros de conjunción IDMS) pueden existir, así como sets entre ellos. De esta manera, la red completa de relaciones es representada por varios pares de sets; en cada set algunos (un) registro es el dueño y uno o más registros son miembros. Usualmente un set define una relación 1:M, aunque 1:1 está permitido.

Las bases de datos de red son similares a las jerárquicas al tener una estructura jerárquica. Pero tienen unas diferencias clave. En vez de parecer un árbol de cabeza, una base de datos de red se parece más a una telaraña o red interconectada de registros. En las bases de datos de red, los hijos son llamados miembros y los padres son llamados dueños. La diferencia más importante es que cada hijo o miembro puede tener más de un padre (o dueño).

Como en las bases de datos jerárquicas, las bases de datos de red son principalmente usadas en mainframes. Dado que se pueden lograr más conexiones entre los tipos de datos, las bases de datos de red son consideradas más flexibles.

Igualmente, dos limitaciones deben ser consideradas. Al igual que con las bases de datos jerárquicas, las bases de datos de red deben ser definidas con antelación. Y también existe un límite en la cantidad de conexiones posibles entre registros.

Modelo Relacional



Las bases de datos relaciones (RDBMS - relational database management system) son bases de datos basadas en el modelo relacional desarrollado por E.F. Codd. El modelo relacional está basado en el álgebra relacional.

Una base de datos relacional permite la definición de estructuras de datos, almacenamiento y recuperación, y restricciones de integridad. En tales bases de datos los datos y las relaciones entre ellos están organizados en tablas. Una tabla es una colección de registros y cada registro en una tabla contiene los mismos campos.

Propiedades de una tabla relacional:

- Los valores son atómicos
- Cada fila es única
- Los valores en cada columna son del mismo tipo
- La secuencia de columnas es insignificante
- La secuencia de filas es insignificante
- Cada columna tiene un nombre

Las bases de datos relacionales funcionan sobre el principio de que cada tabla contiene un campo clave que identifica de manera única a cada fila, y que estos campos clave pueden ser utilizados para conectar una tabla de datos con otra. Cuando los

campos en dos tablas diferentes toman el valor del mismo set, se puede producir una operación de unión al emparejar dichos campos. Esta idea puede ser extendida a unir múltiples tablas en múltiples campos.

Dado que las relaciones son especificadas al momento de recuperar los datos, las bases de datos relacionales son clasificadas como sistemas dinámicos de administración de base de datos.

Las bases de datos relacionales se han vuelto bastante populares por dos razones principales. Primero las bases de datos relacionales pueden ser utilizadas con relativamente poco entrenamiento. Segundo, las entradas en la base de datos pueden ser redefinidas sin afectar toda la estructura. La desventaja de usar una base de datos relacional es que buscar datos puede tomar más tiempo que con los demás métodos.

Mapeo Objeto Relación (ORM)

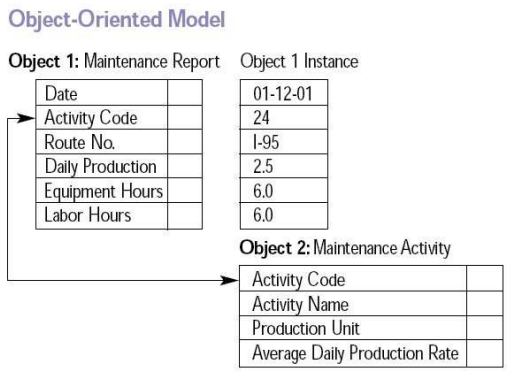
El mapeo Objeto-Relacional (Object-relational mapping (ORM)) es una técnica para convertir datos entre tipos de sistemas en programación orientada a objetos. Esto crea, en efecto, una "base de datos de objetos virtual" que puede ser utilizada dentro del lenguaje de programación.

Los manejos de datos en programación orientada a objetos son implementados generalmente manipulando objetos que son casi siempre valores no escalares.

Aún así, muchas bases de datos populares tales como los sistemas de bases de datos de lenguaje de consulta estructurado (structured query language database management systems (SQL DBMS)) solo pueden manipular valores escalares, tales como enteros y cadenas, organizados en tablas. El programador debe convertir los valores del objeto en grupos de valores más simples para poder ser almacenados en la base de datos (y convertirlos de vuelta al buscarlos), o usar solamente valores escalares dentro del programa. El mapeo objeto – relacional se utiliza para implementar la forma primeramente enunciada.

El corazón del problema es traducir las representaciones lógicas de los objetos a una forma atomizada que sea capaz de ser almacenada en la base de datos, mientras que de alguna manera se preserven las propiedades de los objetos y sus relaciones para que puedan ser recargados como un objeto cuando sea necesario. Si esta funcionalidad de almacenamiento y recuperación es implementada, se dice que los objetos son persistentes.

Modelo Orientado a Objetos



Las DBMS de objetos agregan la funcionalidad de base de datos a los lenguajes de programación objetual. Brindando mucho más que almacenamiento persistente.

Las DBMS de objetos extienden la semántica de los lenguajes de programación objetual para proveer completas capacidades de bases de datos, mientras se mantiene la compatibilidad con el lenguaje nativo.

El mayor beneficio de este enfoque es la unificación del desarrollo de la aplicación y la base de datos en el mismo modelo de datos. Como resultados, las aplicaciones requieren menos código, utilizan un modelo de datos más natural, y las bases del código

son más sencillas de mantener. Los desarrolladores de objetos pueden escribir una aplicación de bases de datos completa con un modesto esfuerzo adicional.

De acuerdo con Rao (1994), “El paradigma de las bases de datos orientadas a objetos (OODB) es la combinación de los lenguajes de programación orientados a objetos (OOPL) y los sistemas de persistencia. El poder de las OODB surge del tratamiento por igual tanto de los datos persistentes, encontrados en una base de datos, como los datos transitorios, pertenecientes a programas en ejecución.”

A diferencia de las DBMS relacionales donde las estructuras complejas deben ser aplanadas para encajar en tablas o ser unidas desde esas tablas para formar la estructura en memoria, las DBMS de objeto no tienen una carga adicional para almacenar u obtener una red o jerarquía de objetos relacionados. Este mapeo uno a uno de objetos en el lenguaje de programación a objetos de base de datos tiene dos beneficios sobre otros métodos de almacenamiento: provee un mayor rendimiento al administrar los objetos, y permite una mejor administración de interrelaciones complejas entre objetos.

Esto convierte a las DMBS de objetos muy adecuadas para aplicaciones tales como sistemas de análisis de riesgos financieros, aplicaciones de servicios de telecomunicaciones, estructuras de documentos de Internet, diseño y fabricación de sistemas, sistemas de registro de pacientes de hospitales, los cuales cuentan con relaciones complejas en los datos.

Las bases de datos orientadas a objetos tienen dos desventajas. Primero, son un poco más costosas para desarrollar. Segundo, muchas organizaciones son renuentes a abandonar o convertir las bases de datos en las que han invertido dinero en desarrollar e implementar. De igual manera, los beneficios de las bases de datos orientadas a objetos son muy atractivos.

La posibilidad de mezclar y relacionar objetos reusables provee de una increíble capacidad multimedia. Organizaciones de cuidado de la salud, por ejemplo, pueden almacenar, rastrear y obtener imágenes de escaneos CAT, rayos X, electrocardiogramas y muchas otras formas de datos cruciales.

Comparación

Relacional

En cuanto a las diferencias estructurales, dado que son dos modelos diferentes, nos encontramos con que los conceptos son distintos.

Esta es una breve comparación entre las dos nociones:

Relacional	**Orientado a objetos**
Fila/Registro Los registros son similares, dado que en ambos casos se almacenan elementos atómicos. La diferencia radica en que los registros solo pueden almacenar elementos atómicos limitados.	Objeto Los objetos pueden contener otros objetos como miembros
Fila / Definición de registro La definición de un registro es siempre dependiente de una base de datos. Las definiciones de un registro no siempre mapean directamente sobre el tipo del lenguaje de programación y debe siempre ser convertido desde y hacia la base de datos.	Clase Las clases describen los objetos con los que se trabaja, y son mucho más fáciles de entender para el usuario. Las clases también encapsulan la lógica de negocio junto con los datos, manteniendo todo convenientemente en el mismo lugar. En el caso de objetos es más seguro, dado que la traducción de un tipo a otro suele causar pérdida de transformación o sobrecarga en el rendimiento.
Tablas Las tablas y las colecciones son similares porque ambos contienen muchos registros/objetos. Ambos suelen tener estructuras de índices para acceso más rápido. Aún así, las tablas poseen una estructura más rígida en cuanto todas las filas en una tabla deben poseer los mismos campos.	Colección Una colección puede contener objetos de diferentes clases utilizando herencia (o polimorfismo).
Campo El campo en una base de datos relacional siempre es atómico (String o número) y siempre almacenado físicamente.	Miembro Los miembros pueden estar estructurados, por ejemplo, pueden ser otros objetos, y también pueden ser computados mediante un método.
Consulta Una consulta relacional solo puede especificar una tabla como resultado, lo que significa que todos los registros deben ser del mismo tipo y el resultado es unidimensional. Pero las consultas relacionales permiten la creación de definiciones de registros dinámicos usando "*select*" y "*join*".	Filtro Dado que los tipos dinámicos no tendrían una clase correspondiente los *"select"* y *"join"* no suelen ser soportados en modelos orientados a objeto, por lo que los filtros se corresponden a la cláusula *"where"* de una consulta relacional.

Clave Los registro de diferentes tablas en una base de datos relacional se combinan usando claves y *join*. Si bien esto es un mecanismo muy general, es lento, y las claves son difíciles de mantener.	Enlace Los objetos en las bases de datos usan enlaces, que son mucho más rápidas que las búsquedas de claves y pueden ser mantenidas por el objeto automáticamente. Esto previene datos corruptos. Los enlaces también son conceptos manejados en Java y Python (referencia de objetos) y C++ (punteros) haciéndolo más fácil de manejar para un programador.
Unión Los registros en una base de datos relacional son obtenidos mediantes operaciones de unión (*join*). Las uniones tienen por desventaja el hecho que se vuelven muy lentos cuando se manipulan varias tablas. Para dos tablas ocurren m*n combinaciones y por cada tabla extra involucrada, esta cifra debe ser multiplicada por el tamaño de la tabla.	Navegación Por otro lado la navegación es un solo paso muy rápido. Solo consiste en acceder a la propiedad del objeto en cuestión.

Actualmente las bases de datos relacionales implementan una forma de proveer patrones de comportamiento a los datos almacenados, esto es, darles una forma de manejarlos como si fuesen objetos utilizando Procedimientos Almacenados.

La desventaja de los procedimientos almacenados radica en que si bien simplifican y estandarizan el comportamiento de los datos, no provee una forma de vincular los datos con el comportamiento, es decir, no hay manera de hacer explícita la correspondencia entre ciertos procedimientos y las tablas con las cuales trabaja.

Además el lenguaje varía de implementación a implementación, y requiere que el programador conozca no solo otro lenguaje adicional sino que pueda desenvolverse, conocer y mantener dos modelos diferentes.

En el caso de las bases de datos objetuales, el comportamiento está fuertemente ligado al código de la clase. Trayendo todos los beneficios que actualmente se conocen de la programación orientada a objetos:

- Encapsulamiento
- Buena definición de un objeto
- Autonomía en su comportamiento (es el mismo objeto quien maneja su propio estado)
- Permite plantear una interfaz de métodos disponibles. De sencilla comprensión sin tener que conocer el funcionamiento interno del mismo.
- Se utiliza el mismo lenguaje en el programa para la lógica de negocios de los objetos y los algoritmos de validación.
- Permite herencia, no solo de datos sino también de comportamiento. Las bases de datos permiten la herencia de datos, la cual tiene la desventaja de requerir duplicar los campos de la tabla padre en la tabla hija, o mantener referencias a la tabla hija lo cual conlleva la necesidad de

manejar claves de manera más complicada. Pero no permiten una forma concisa de heredar comportamiento.

- Polimorfismo

ZODB no implementa un sistema de *constraints* como los encontrados en una base de datos relacional. Esto se debe en parte a su diseño transparente. Y por otra parte porque lo más natural en un entorno de programación orientada a objetos es manejar la validación de datos dentro de la lógica del objeto, ya sea utilizando *Properties* o mediante la correcta definición de la interfaz de comportamiento del objeto mediante los correspondientes métodos.

En el caso de los tipos de datos, las bases relacionales requieren que todos los registros para una misma definición de registros (tabla) contengan exactamente la misma cantidad de miembros y de exactamente los mismos tipos. Como se explicó anteriormente, ZODB permite el uso de miembros heterogéneos. Esto no solamente es poderoso sino algo que es muy utilizado en lenguajes que soportan el duck-typing[1] como lo es Python. Por lo que en este caso, para un desarrollador de Python, el uso de ZODB es aún más transparente y natural, sin tener que adaptar su código a la base de datos.

ORM

Los ORMs son soluciones alternativas a los problemas y limitaciones de las bases de datos relacionales.

Lo cual no significa que eliminan los problemas, simplemente se encargan de ellos con la mínima participación del programador.

El intento por lidiar con los problemas de traducción de base de datos relacionales a un entorno orientado a objetos, trae aparejado nuevos problemas.

Esto es bien conocido en el mundo de los ORM como diferencia de impedancia Objeto-Relacional. Entre estos problemas encontramos:

- Encapsulamiento
- Accesibilidad
- Interfaz, clase, herencia y polimorfismo
- Mapeo a conceptos relacionales
- Diferencias de tipos de datos
- Diferencias estructurales y de integridad
- Diferencias manipulativas
- Diferencias transaccionales

Todas estas diferencias hacen que un ORM sea un componente de software altamente complejo, con la tarea no trivial de encargarse de toda la base de datos.

Por lo que el funcionamiento suele ser de un rendimiento mucho menor al del uso de una base de datos relacional en forma directa.

[1] El Glosario de Python define el duck typing como: Estilo de programación Python que determina el tipo de un objeto a través de la inspección de sus métodos o su conjunto de atributos en lugar de emplear la relación con algún tipo de objeto ("Si luce como un pato y suena como un pato, debe ser un pato") Enfatizando las interfaces sobre los tipos específicos, el código bien diseñado mejora su flexibilidad al permitir sustitución polimórfica. El duck typing evita las pruebas con type() o isinstance(). En lugar de eso, prefiere el estilo de programación "Es mas fácil pedir perdón que pedir permiso".

Por otro lado tampoco libera al programador de la necesidad de conocer los conceptos mínimos de una base de datos relacional así como la implicancia de ciertas decisiones a la hora de ajustar el funcionamiento del ORM.

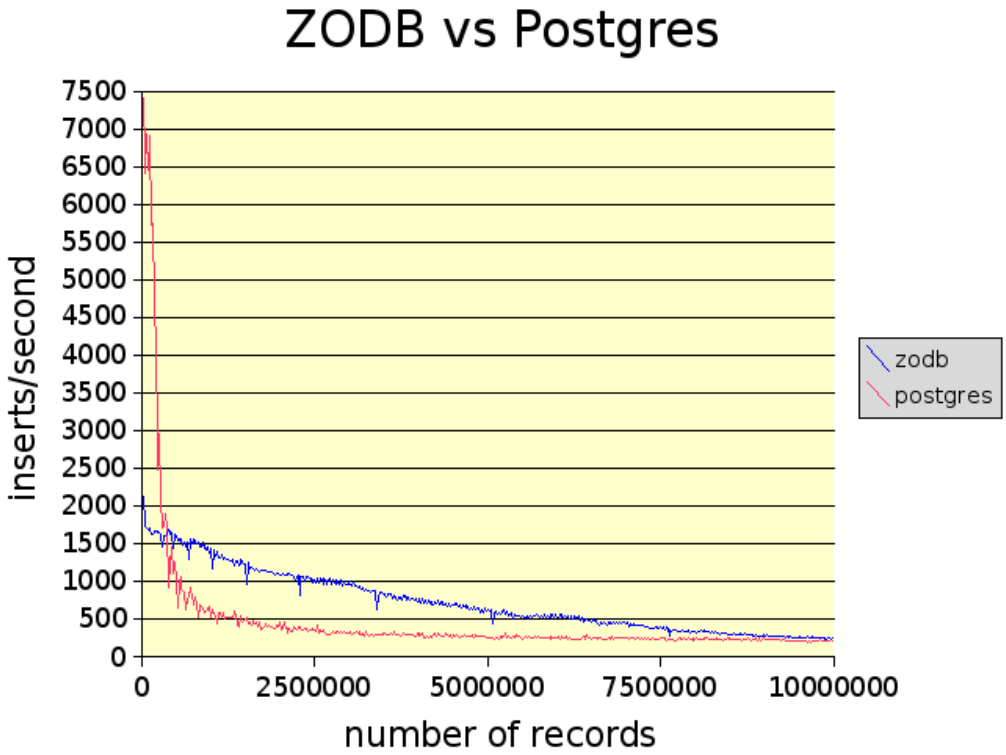
Otras Ventajas

En la práctica, las bases de datos de objetos tienen muchas ventajas sobre las bases de datos relacionales. Típicamente:

- Buen rendimiento transaccional
- Manejo de estructuras de datos complejas
- Facilidad de uso
- Capacidades de búsqueda flexibles
- Interfaz de acceso a los datos estándar
- Pueden ser usados para aplicaciones de inteligencia de negocios

En cuanto a rendimiento se han hecho muchas comparaciones tanto en ámbito de laboratorio como en ámbito de producción.

A continuación se muestra un ejemplo del rendimiento de ZODB comparado con una base de datos relacional conocida.



Esta prueba muestra que ZODB es más rápida al insertar nuevos objetos que Postgres.

En cuanto a los tiempos de búsqueda de los objetos en ZODB:

Cantidad de objetos	Tiempo promedio de búsqueda
100000	0.00000311
1000000	0.00000648
10000000	0.00230820

De la tabla anterior se infiere claramente que las búsquedas en un BTree con 10 millones de objetos son muy veloces, alrededor de 2 milisegundos. En una tabla Postgres con 10 millones de registros el tiempo de búsqueda promedio es de 14 milisegundos.

Como ilustración se tiene la siguiente cita del lanzamiento de prensa de Versant, referente a la aplicación para reservas y seguimiento de asientos de Air France.

“No es la complejidad de los datos sino la complejidad de las interrelaciones entre los diferentes puntos de datos que la solución debe manejar”, dijo Steve Clampett, senior vicepresidente de SDT. “Usando una base de datos relacional, una simple consulta podía requerir mas de 30 uniones (joins) Incluso Oracle era una orden de magnitud más lenta que Versant las pruebas de esta aplicación”

Las pruebas de rendimiento en Air France revelan que la aplicación de Disponibilidad de Proceso se desarrolla a un nivel de consistencia igual o mejor que los requerimientos de la industria. Por ejemplo, el tráfico de transacciones de ventas de un día reciente alcanzó más de 207,000 transacciones. De dicho número, 99.48% de las transacciones se realizaron en menos de 30 milisegundos.

Zope Object Data Base (ZODB)

La base de datos de objetos de Zope (ZODB) es una base de datos orientada a objetos para almacenar objetos de Python de manera transparente y persistente. Se incluye como parte del servidor de aplicaciones Web Zope, pero puede ser utilizada independientemente.

ZODB es una base de datos para Python madura, con cientos de miles de sistemas corriendo sobre ella actualmente.

Creada por Jim Fulton de Zope Corporation a fines de los ’90. Comenzó como el simple Sistema de Objetos Persistentes (POS) durante el desarrollo de Principia (que luego se convirtió a Zope)

ZODB toma un enfoque minimalista comparado con otros sistemas denominados “Bases de datos”.

Provee persistencia y transacciones, pero no provee seguridad, indexación ni utilidades de búsqueda.

Existen paquetes de terceros que proveen estos y otros servicios para ZODB.

Características

La naturaleza dinámica de Python permite desarrollar aplicaciones rápidamente, evitando los tiempos de compilado y las declaraciones de tipos requeridas por otros lenguajes.

ZODB ofrece beneficios similares: los desarrolladores que utilicen ZODB pueden almacenar sus objetivos transparentemente sin ningún mapeo pesado y engorroso de objetos a tablas de una base de datos relacional. Y ya no necesitan preocuparse por descomponer complejos objetos para encajar en un modelo relacional o de sistema de archivos.

Entre sus características principales encontramos:

- Transacciones
- Historial, deshacer
- Transparencia
- Almacenamientos intercambiables
- Control de concurrencia multiversión (MVCC)
- Escalabilidad a través de la red (utilizando ZEO).

Familiaridad

ZODB ofrece un ambiente familiar para los desarrolladores de Python. No solo almacena Objetos de Python de manera nativa, también utiliza un mecanismo de serialización conocido (el modulo pickle de la biblioteca estándar de Python). Así también ZODB se encuentra escrito en Python mismo, por lo que no requiere de herramientas especiales para ser inspeccionado.

Simplicidad

ZODB es una base de datos jerárquica. Existe un objeto raíz, inicializado cuando se crea la base de datos. El objeto raíz es utilizado como un diccionario Python y puede contener otros objetos. Para almacenar un objeto en la base de datos, solo basta con asignarlo a una nueva clave en su contenedor.

Listo para producción

ZODB ha estado presente por más de 10 años y ha sido puesto a trabajar en múltiples ambientes de producción.

Transparencia

Para hacer que una instancia de una clase sea automáticamente persistente, basta con que herede de la clase base *Persistent.*

ZODB luego se encarga de guardar estos objetos en cuanto son modificados. Algunos objetos no persistentes también pueden ser almacenados fácilmente en ZODB pero en algunos casos se debe advertir a ZODB cuando éstos cambian.

Soporte de Transacciones

Las transacciones son una serie de cambios sobre la base de datos que necesitan ser aplicadas como una unidad. O todos los cambios se aplican (commit), o no se aplica ninguno (rollback).

Este concepto es el mismo que en algunos sistemas de bases de datos relacionales. Los sistemas transaccionales se aseguran que la base de datos nunca esté en estado de inconsistencia al implementar las cuatro propiedades conocidas como ACID:

- Atomicidad
- Concurrencia
- Aislamiento
- Durabilidad

Puntos de guardado

Dado que los cambios hechos durante una transacción se mantienen en memoria hasta que la transacción es aceptada, el uso de la memoria puede subir demasiado. Los puntos de guardado permiten aceptar parte de la transacción antes de terminar, así los cambios son escritos a la base de datos y la memoria es liberada de manera progresiva.

Los cambios en el punto de guardado no son aceptados hasta que toda la transacción ha finalizado, para que si es abortada, todos los puntos de guardado son restaurados (rollback)

Deshacer

ZODB provee un mecanismo muy simple para restaurar (rollback) cualquier transacción aceptada. Esta característica es posible porque ZODB mantiene rastro del estado de la base de datos antes y después de cada transacción. Esto hace posible deshacer cambios en una transacción, aún si otras transacciones han sido aceptadas después.

Por supuesto si los objetos incluidos en la transacción que necesitamos deshacer han cambiado en la última transacción, no será posible deshacer debido a los requerimientos de consistencia.

Historia

Dado que se conserva cada transacción en la base de datos, es posible ver la historia del estado de un objeto en las transacciones anteriores y compararlas con su estado actual. Esto les permite a los desarrolladores implementar rápidamente un sistema sencillo de versionado.

BLOBS

Los objetos binarios grandes (Binary large objects (BLOBS)), tales como imágenes o documentos de oficina, no precisan de las posibilidades de versionado que ofrece ZODB.

De hecho si fuesen manipulados como atributos comunes de los objetos, los BLOBS incrementarían rápidamente el tamaño de la base de datos y generalmente harían las cosas mucho más lentas. Por ello ZODB utiliza un almacenamiento especial para los BLOBS, lo cual hace muy fácil manipular grandes archivos de hasta varios cientos de megabytes sin problemas de rendimiento.

Cacheo en memoria

Cada vez que se lee un objeto de la base de datos, se mantiene en un cache LRU dentro de la memoria. Subsecuentes accesos a este objeto consume menos recursos y tiempo. ZODB maneja el cache de manera transparente y automáticamente retira los objetos que no han sido accedidos por largo tiempo. El tamaño de la cache puede ser configurado, para que los servidores con más memoria puedan beneficiarse de esto.

Comprimir

ZOBD preserva todas las versiones de los objetos almacenados. Esto significa que la base de datos crece con cada modificación y puede alcanzar grandes tamaños, que pueden alentar el sistema y consumir mas espacio del necesario. ZODB permite purgar viejas revisiones de los objetos almacenados mediante un procedimiento conocido como Compresión (o Empacado (Packing)).

El procedimiento de compresión es suficientemente flexible como para permitir remover solo los objetos más antiguos que cierta cantidad de días, manteniendo así las versiones más nuevas.

Almacenamientos Intercambiables

Por defecto, ZODB almacena la base de datos en un solo archivo. El programa que administra esto es llamado un Almacén de Archivo (file storage).

De igual manera, ZODB está construido de tal manera que otros almacenamientos pueden ser conectados sin la necesidad de modificar el código. Esto puede ser utilizado para almacenar los datos de ZODB en otros formatos.

Escalabilidad

Zope Enterprise Objects (ZEO) es un almacenamiento de red para ZODB. Utilizando ZEO, un gran número de clientes de ZODB pueden conectarse a la misma ZODB. ZEO puede ser utilizado para proveer escalabilidad dado que la carga puede ser distribuida en varios clientes ZEO en vez de uno solo.

Método de funcionamiento

Instalación

En el caso de usar Windows se puede instalar utilizando **easy_install** el cual permite administrar paquetes de Python descargados automáticamente desde la red. Forma parte del paquete **setuptools** (**http://pypi.python.org/pypi/setuptools**)

```
1. easy_install ZODB3
```

De contar con el sistema operativo basado en Debian puede utilizarse el administrador de paquetes del sistema.

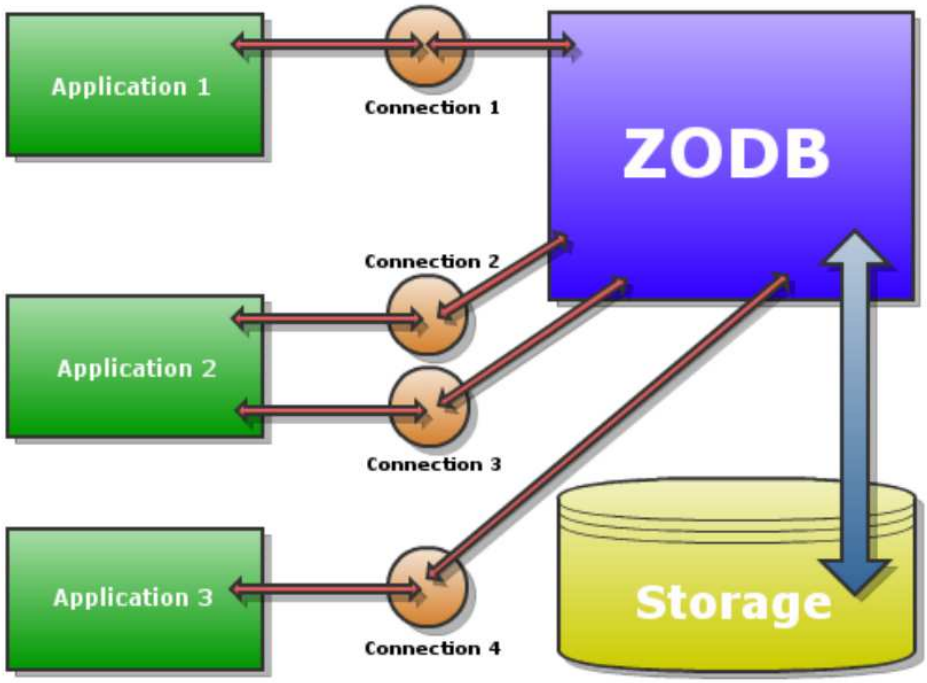
```
1. apt-get install python-zodb
```

Conectando con la base de datos

ZODB define algunos conceptos con los que deberemos trabajar.

- Un *storage* se ocupa de los detalles de bajo nivel para almacenar y recuperar objetos. El mas utilizado es *FileStorage* que utiliza un solo archivo para almacenar los objetos. Existen otros tipos, como *RelStorage* y *DirectoryStorage* los cuales utilizan una base de datos relacional y una estructura de directorios respectivamente.
- Una base de datos es un proceso que administra un pool de conexiones. Es implementada por la clase *DB*.
- Una conexión provee acceso a los objetos dentro de una base de datos y se encarga de utilizar cache para un acceso más rápido. Una aplicación multi-hilos puede abrir una conexión por hilo y cada conexión tendrá su propio cache y podrá modificar objetos independientemente de otros hilos.
- El objeto raíz de la base de datos puede ser accedido mediante una conexión y actúa como un diccionario Python.

El siguiente diagrama ilustra sus interacciones.



Para cada caso se creará primero un archivo .py donde se guardará el código que se cree.

Para poder conectarse a una base de datos ZODB se debe proceder de esta manera.

En este caso crearemos un archivo **mizodb.py** donde almacenaremos una clase que permita abrir y cerrar de manera sencilla nuestra base de datos con el nombre que le sea indicado.

```
1.  # mizodb.py
2.  from ZODB import FileStorage, DB
3.  import transaction
4.
5.  class MiZODB(object):
6.      def __init__(self, archivo):
7.          self.storage = FileStorage.FileStorage(archivo)
8.          self.db = DB(self.storage)
9.          self.conexion = self.db.open()
10.         self.raiz = self.conexion.root()
11.     def close(self):
12.         self.conexion.close()
13.         self.db.close()
14.         self.storage.close()
15.
```

Almacenando datos simples

ZODB permite almacenar cualquier objeto de Python.
El siguiente código muestra como almacenar valores simples

```
1. # simple.py - escribe algunos valores simples
2. from mizodb import MiZODB, transaction
3.
4. db = MiZODB('Data.fs')
5. dbroot = db.raiz
6. dbroot['numero'] = 3
7. dbroot['string'] = 'Gift'
8. dbroot['lista'] = [1, 2, 3, 5, 7, 12]
9. dbroot['diccionario'] = { 1918: 'Red Sox', 1919: 'Reds' }
10. dbroot['anidado'] = {
11.     1918: [ ('Red Sox', 4), ('Cubs', 2) ],
12.     1919: [ ('Reds', 5), ('White Sox', 3) ],
13. }
14. transaction.commit()
15. db.close()
```

Leyendo datos simples

```
1. # leer_simple.py Muestra lo que hay en la base de datos
2. from mizodb import MiZODB
3. db = MiZODB('./Data.fs')
4. dbroot = db.raiz
5. for key in dbroot.keys():
6.     print key + ':', dbroot[key]
7. db.close()
```

ZODB siempre sabe cuando se asigna un nuevo valor a una clave. Por lo que un cambio a la base de datos anterior será detectado automáticamente y será persistido

```
1. dbroot['string'] = 'Otra cosa'
2. transaction.commit()
```

En el caso de listas y diccionarios, si se desea modificar uno en vez de reemplazarlo por completo, ZODB no podrá ver el cambio automáticamente. Esto es una cualidad de la mutabilidad.

Si se desea modificar uno de estos objetos se debe notificar a la base de datos que ha cambiado asignando el atributo *_p_changed* a la raíz de la base de datos.

```
1. diccionario = dbroot['diccionario']
2. diccionario[1920] = 'Indians'
3. db._p_changed = 1
4. transaction.commit()
```

Eliminar un dato

Eliminar un objeto es tan sencillo como hacer

```
1. del dbroot['numero']
2. transaction.commit()
```

Debe notarse que nada hubiese sucedido en la base de datos en ninguno de los ejemplos anteriores si no se hubiese hecho la llamada a *transaction.commit()*.

Al igual que en una base de datos relacional, solo mediante la llamada a *commit* se logra que los cambios aparezcan de manera atómica en la base de datos.

Abortando una transacción

Una cualidad importante de un sistema transaccional es permitir abortar la transacción en caso de un error. Esta acción lleva la base de datos al último estado de consistencia en que se encontraba antes de comenzar los cambios. En el caso de ZODB abortar una transacción es un proceso muy sencillo, luego del cual los objetos vuelven automáticamente a su estado correspondiente.

Con este script se demuestra su funcionamiento:

```
1. # abort_transaction.py
2. from mizodb import MiZODB, transaction
3. from modelo import Host
4.
5. db = MiZODB('./Data.fs')
6. dbroot = db.raiz
7. host1 = dbroot['www.example.com']
8. print "Ip actual:" , host1.ip
9.
10. host1.ip = "192.168.7.3"
11. print "Ip modificada:", host1.ip
12.
13. transaction.abort()
14.
15. print "Valor luego de abortar la transacción:", host1.ip
16. db.close()
```

La salida producida es similar a esta.

```
usuario@debian:~/zodb$ python abort_transaction.py
Ip actual: 192.168.7.2
Ip modificada: 192.168.7.3
Valor luego de abortar la trasaccion: 192.168.7.2
usuario@debian:~/zodb$
```

Probando de esta manera la sencillez y transparencia con que funciona el manejo de transacciones.

Trabajando con Objetos

Por supuesto que se no es deseable trabajar con un bosque de creciente complejidad de estructuras de datos como listas, tuplas y diccionarios. Por el contrario se deseará trabajar con objetos de Python cuyos atributos son persistidos.

Crearemos un pequeño archivo Python que demuestre este funcionamiento.

Para ello la clase a persistir debe heredar de la clase "*Persistent*".

Dado que Python permite multi-herencia, el hecho de requerir heredar de "*Persistent*" de ninguna manera evita que las clases a construir no puedan heredar de otras clases.

Un modelo

```
1. # modelo.py - Simple modelo
2. from persistent import Persistent
3. class Host(Persistent):
4.     def __init__(self, hostname, ip, interfaces):
5.         self.hostname = hostname
6.         self.ip = ip
7.         self.interfaces = interfaces
```

Como se puede apreciar, la única diferencia entre esta clase y una clase normal de python es que hereda de la clase Persistent.

Almacenando Objetos

Ahora se pude crear instancias de esta clase y persistirlas en ZODB, de la misma manera que persistimos las estructuras simples anteriormente.

```
1. # escribir_hosts.py
2. from mizodb import MiZODB, transaction
3. from modelo import Host
4. db = MiZODB('./Data.fs')
5. dbroot = db.raiz
6.
7. host1 = Host('www.example.com', '192.168.7.2', ['eth0', 'eth1'])
8. dbroot['www.example.com'] = host1
9. host2 = Host('dns.example.com', '192.168.7.4', ['eth0', 'gige0'])
10. dbroot['dns.example.com'] = host2
11. transaction.commit()
12. db.close()
```

Leyendo Objetos

El siguiente script re-abrirá la base de datos y demostrará que el los objetos "host" han sido persistidos exitosamente (al verificar el tipo de cada elemento obtenido, ignorará el resto de objetos que puedan quedar de los primeros ejemplos)

```
1. # leer_hosts.py - show hosts in the database
2. from mizodb import MiZODB
3. from modelo import Host
4.
5. db = MiZODB('./Data.fs')
6. dbroot = db.raiz
7.
8. for key in dbroot.keys():
9.     obj = dbroot[key]
10.    if isinstance(obj, Host):
11.        print "Host:", obj.hostname
12.        print " IP address:", obj.ip, " Interfaces:", obj.interfaces
13. db.close()
```

El programa produce una salida similar a la siguiente:

```
usuario@debian:~/zodb$ python leer_hosts.py
Host: www.example.com
 IP address: 192.168.7.2  Interfaces: ['eth0', 'eth1']
Host: dns.example.com
 IP address: 192.168.7.4  Interfaces: ['eth0', 'gige0']
usuario@debian:~/zodb$ _
```

Modificando un objeto

Así como el objeto "*dbroot*" detecta cuando se ha ingresado un nuevo valor a alguna clave, los objetos *Persistent* detectarán automáticamente cuando se asigna alguno de sus atributos y los guardará en la base de datos.

Por ejemplo, el código para cambiar la dirección IP del primer host se procedería de la siguiente manera:

```
1. host = dbroot['www.example.com']
2. host.ip = '192.168.7.141'
3. transaction.commit()
```

Trabajando con Listas y Diccionarios

Si se intentara expandir la complejidad del programa incluyendo listas o diccionarios en alguna de las clases creadas, se tendría el mismo problema nombrado con anterioridad. El hecho de que dichos objetos sean mutables hacen que, por el diseño transparente de ZODB, los cambios no sean actualizados automáticamente.

Para lograrlo se debería asignar el atributo *_p_changed* al objeto que contenga la lista.

ZODB permite trabajar de forma más intuitiva y natural al proveer de objetos específicos que pueden reemplazar directamente a las listas y diccionarios.

En este ejemplo podemos ver el funcionamiento básico.

```
1.  # lista_y_diccionario.py
2.  from mizodb import MiZODB, transaction
3.  from persistent.list import PersistentList
4.  from persistent.mapping import PersistentMapping
5.  db = MiZODB('./Data.fs')
6.  dbroot = db.raiz
7.  print "Lista al iniciar" , dbroot['lista']
8.  print "Diccionario al iniciar", dbroot['diccionario']
9.  lista = PersistentList(dbroot['lista'])
10. dbroot['lista'] = lista
11. lista.append(42)
12. dic = PersistentMapping(dbroot['diccionario'])
13. dbroot['diccionario'] = dic
14. dic[1920] = "Red sox"
15.
16. transaction.commit()
17. print "Lista al terminar", dbroot['lista']
18. print "Diccionario al terminar", dbroot['diccionario']
19. #podemos ver que conservan las facilidades de los objetos nativos de
    python
20. lista.pop()
21. del dic[1920]
22. transaction.commit()
23. db.close()
```

La salida del programa es similar a la siguiente:

```
usuario@debian:~/zodb$ python lista_y_diccionario.py
Lista al iniciar [1, 2, 3, 5, 7, 12]
Diccionario al iniciar {1918: 'Red Sox', 1919: 'Reds'}
Lista al terminar [1, 2, 3, 5, 7, 12, 42]
Diccionario al terminar {1920: 'Red sox', 1918: 'Red Sox', 1919: 'Reds'}
usuario@debian:~/zodb$
```

Haciendolo eficiente

Al programar con ZODB, los diccionarios de Python no son siempre lo que uno necesita. El caso mas importante es cuando se desea almacenar un diccionario muy grande. Cuando un diccionario de Python es accedido, se debe decodificar (unpickle) y cargar en memoria. Si se está almacenando algo muy grande, como las entradas en una base de datos de 100,000 usuarios, decodificar un objeto tan grande resultaría lento.
Los BTree son una estructura de datos de árbol balanceado que se comportan como un mapeado pero que distribuyen las claves (key) en varios nodos. Los nodos son almacenados en orden (esto trae consecuencias importantes). De esta manera los nodos son decodificados y traídos a memoria solo al ser accedidos, de esta manera el árbol entero no necesita ocupar memoria (excepto si realmente necesita leer todas las claves)

Un objeto BTree provee todos los métodos esperables de un mapa, con algunas que extensiones que aprovechan el hecho que las claves están ordenadas.

Algunos métodos extras son *minKey*() y *maxKey*(), que devuelven el valor máximo y mínimo de las claves respectivamente. Los métodos varios para enumerar claves, valores e ítems también aceptan valores máximo y mínimo de clave como parámetro (permitiendo una búsqueda por rango).

El uso de BTree también permite acceder a diferentes funciones sobre grupos. Por ejemplo *difference*(), *union*(), e *intersection*() que devuelven el resultado de la operación de grupos diferencia, unión e intersección respectivamente sobre los elementos de BTree.

```
1.  # btree.py
2.  from mizodb import MiZODB, transaction
3.  from modelo import Host
4.  from BTrees.OOBTree import OOBTree
5.
6.  db = MiZODB('./Data.fs')
7.  dbroot = db.raiz
8.  # Asegura que exista la propiedad "hosts" en la raiz
9.  if not dbroot.has_key('hosts'):
10.     dbroot['hosts'] = OOBTree()
11.     dbroot['ultimo_host'] = 0
12.
13. hosts = dbroot['hosts']
14. ultimo = dbroot['ultimo_host']
15.
16. nuevo = Host('www.example3.com', '192.168.7.4', ['eth1', 'wlan0'])
17.
18. hosts[ultimo] = nuevo
19.
20. dbroot['ultimo_host'] = ultimo +1
21. transaction.commit()
22. db.close()
```

Nota: OOBtree es la implementación de Árboles Balanceados que utiliza Objetos como índice (key). Existen otras implementaciones de BTree que permiten utilizar valores de tipos básicos para permitir mayor rendimiento y fiabilidad. Por ejemplo, IOBTree es la implementación que permite utilizar enteros (Int) como clave y objetos como valores. Así mismo existen otras implementaciones como OIBtree y IIBtree

Herramientas

Mantenimiento

Una base de datos de ZODB es fácil de mantener, dado que no contiene un esquema (Schema) que requiera diseño o modificación, el único mantenimiento regular a realizar es una compresión periódica para evitar que consuma todo el disco.

Los administradores de base de datos ya están acostumbrados a este tipo de acciones dado que las bases de datos relacionales también aumentan de tamaño en el disco hasta que se realice un comando "*VACUUM*" periódicamente. Este comportamiento es el mismo en ZODB y básicamente es necesario para permitir realizar rollbacks en una transacción.

Existen dos maneras de realizar una compresión (packing). Si se ejecutan scripts simples que abren el archivo ".fs" directamente, como los demostrados anteriormente, entonces solo se necesita crear un pequeño script que abra el archivo ".fs" y ejecute el comando "*pack*":

```
1. db = DB(storage)
2. db.pack()
```

Dado que no es posible que dos programas puedan abrir el mismo archivo al mismo tiempo, no se puede ejecutar un script de compresión mientras otros programas están usando dicha base de datos.

En dichos casos se utiliza un servidor ZEO (descripto en la siguiente sección), y la compresión se realiza de manera aún más sencilla, sin requerir la desconexión de ningún cliente. Simplemente se utiliza el comando "*zeopack*" que viene incluido con ZODB

```
1. usuario@debian:~/zodb$ zeopack2.6 -h localhost -p 8080
```

Esto hará que se conecte al servidor ZEO y envíe una orden especial que inducirá al servidor a comprimir la base de datos.

Aún así, lo más sensato es ejecutar este comando cuando la carga de la base de datos es poca.

Zeo

Mientras que todos los ejemplos anteriores abren y modifican localmente el archivo "*Data.fs*" directamente, la mayoría de las instalaciones en producción de ZODB ejecutan sus bases de datos como un servidor. El servidor ZODB es llamado "Zope Enterprise Objects" (ZEO), y está incluido dentro del mismo código de ZODB.

Dado que solo un programa por vez puede acceder de manera segura a un mismo archivo, el servidor ZEO es la única manera de permitir conexiones de varios clientes. Esto es especialmente crítico cuando una base de datos requiere de varias interfaces (front-ends) dispuestas en una configuración de balanceo de carga.

Incluso muchas instalaciones que poseen un solo cliente a la base de datos eligen utilizar el servidor ZEO de igual manera, dado que, como se describió anteriormente, esto permite comprimir la base de datos sin tener que desconectar al cliente

La configuración de ZEO está explicada en mayor detalle en la documentación, pero un archivo de configuración básico se ejemplifica mas abajo:

```
<zeo>
    address 0.0.0.0:8090
    monitor-address 0.0.0.0:8091
</zeo>
<filestorage 1>
    path /home/usuario/zodb/Data.fs
</filestorage>
<eventlog>
    <logfile>
        path /var/tmp/zeo.log
        format %(asctime)s %(message)s
    </logfile>
</eventlog>
```

Una vez escrita esta configuración, por ejemplo en el archivo *zeo.conf*; ejecutar una instancia de ZEO es tan sencillo como:

```
usuario@debian:~/zodb$ runzeo2.6 -C ./zeo.conf
```

Para conectarse desde un cliente, es necesario realizar unas modificaciones a la conexión.

El ejemplo anterior de “MiZODB” dado anteriormente podría ser modificado en la siguiente manera

```
# mizodb_remoto.py
from ZEO.ClientStorage import ClientStorage
from ZODB import DB

class ZODBRemoto(object):
    def __init__(self, server, port):
        server_and_port = (server, port)
        self.storage = ClientStorage(server_and_port)
        self.db = DB(self.storage)
        self.connection = self.db.open()
        self.dbroot = self.connection.root()

    def close(self):
        self.connection.close()
        self.db.close()
        self.storage.close()

```

```
18. if __name__ == "__main__":
19.     mydb = ZODBRemoto('localhost', 8090)
20.     dbroot = mydb.dbroot
21.     print dbroot
```

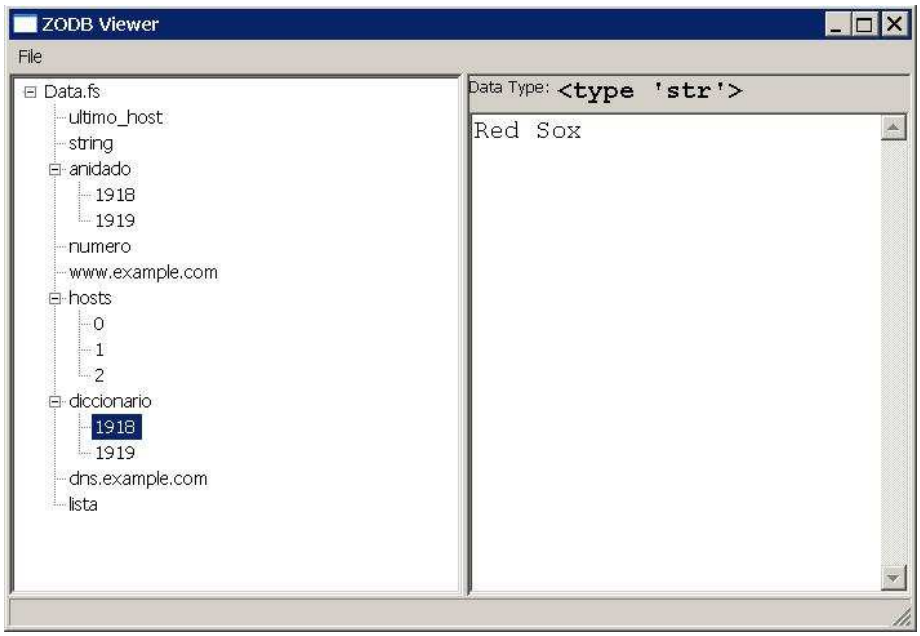
La salida del programa es similar a la siguiente:

```
usuario@debian:~/zodb$ python mizodb_remoto.py
{'ultimo_host': 3, 'string': 'Gift', 'anidado': {1918: [('Red Sox', 4), ('Cubs', 2)]
, 1919: [('Reds', 5), ('White Sox', 3)]}, 'numero': 3, 'www.example.com': <modelo.Ho
st object at 0xa279aec>, 'hosts': <BTrees.OOBTree.OOBTree object at 0xb71441dc>, 'di
ccionario': {1918: 'Red Sox', 1919: 'Reds'}, 'dns.example.com': <modelo.Host object
at 0xa279bec>, 'lista': [1, 2, 3, 5, 7, 12]}
usuario@debian:~/zodb$
```

El objeto "*dbroot*" resultante se utiliza exactamente igual que el objeto "dbroot" ilustrado anteriormente, y realizará sobre el la instancia de ZODB remota exactamente las mismas operaciones que los scripts anteriores realizaban sobre el archivo local "*Data.fs*"

ZODBViewer

Una muestra de la simpleza de ZODB conjunto con las capacidades de introspección de Python es el programa ZODBViewer que nos permite fácilmente analizar cualquier base de datos ZODB.



Historicidad

Podemos demostrar las capacidades de historicidad de ZODB con el siguiente programa:

```
1. # deshacer.py
2. import time
3. import sys
4. from mizodb import MiZODB, transaction
5.
6. db = MiZODB('./Data.fs')
7. zdb = db.db
8. dbroot = db.raiz
9.
10. if not zdb.supportsUndo():
11.     print "La base de datos no soporta deshacer"
12.     exit(-1)
13.
14. fecha_minima = time.mktime((2012, 01, 15, 10, 12, 00, 00, 00, 00))
15. lista_deshacer = []
16. for item in zdb.undoLog(0, sys.maxint):
17.     if item['time']>=fecha_minima:
18.         lista_deshacer.append(item)
19.
20. for item in lista_deshacer:
21.     tid = item['id']
22.     zdb.undo(tid)
23.     transaction.commit()
24.
25. db.close()
```

Además de esto ZODB permite algunas operaciones interesantes a la hora de investigar el historial de transacciones.

Por ejemplo, permite agregar el nombre del usuario y una nota referente a la transacción antes de aceptarla.

```
1. transaction.get().setUser('Usuario')
2. transaction.get().note('Agregar un host')
```

Esta información es visible en el historial de deshacer en la base de datos.

Conclusión

Las bases de datos relacionales han tenido el dominio del mercado durante algún tiempo, esto las ha llevado a contar con cierta madurez y fiabilidad. Sin embargo los lenguajes orientados a objetos siguen emergiendo como los lenguajes dominantes para construir aplicaciones, los beneficios de las OODBMS sobre las RDBMS se hacen más necesarios. Las investigaciones tienden a reforzar esta idea dada la cantidad de proyectos que concluyen o sostienen dichos beneficios en convertir el modelo relacional al orientado a objetos.

Aunque las bases de datos de objetos sean vistas como algo exótico, son una herramienta muy importante para los usuarios de lenguajes dinámicos orientados a objetos. La flexibilidad que proveen respecto de la estructura de objetos y su estructura jerárquica es un complemento natural a la naturaleza de los sistemas de administración de contenido.

Con ZODB, todas estas funciones son entregadas mediante una configuración cliente/servidor, con la adición de una de las soluciones de clustering, puede llevarse a un nivel de robustez necesario para las empresas.

Es remarcable también que, aunque las bases de datos relacionales han sido la elección primaria durante estos últimos años, ZODB es un proyecto que utiliza la última tecnología pero que también es muy robusto, con más de 10 años de desarrollo y usado en muchos proyectos importantes, incluyendo Plone, un Sistema de Administración de Contenido (CMS) basado en Python; y Grok un Framework de desarrollo de Aplicaciones Web de última generación.

Probablemente dado su nombre (Zope Object Data Base), se instaura el pensamiento de la necesidad de utilizar ZODB con ZOPE. Sin embargo ZODB no está necesariamente ligado a ZOPE, si bien éste último resulta un proyecto de excelente utilidad, ZODB es completamente usable por su propia cuenta.

ZODB provee una base de datos orientada a objetos que provee un alto nivel de transparencia. Las aplicaciones pueden tomar ventaja de tener todas las posibilidades de una base de datos sin muchos (o ningún) cambio en la lógica del programa.

ZODB provee de un cache de objetos que provee alto rendimiento, uso eficiente de la memoria, y protección de fuga de memoria debido a referencias circulares.

De la misma manera, la posibilidad de compartir referencia al mismo objeto en python y el manejo transparente de esto en ZODB, provee un mayor rendimiento y menor uso de la memoria.

No es necesario utilizar un método especial para acceder o consultar los objetos, ya que éstos son obtenidos y actualizados mediante interacciones normales entre objetos utilizando los mecanismos naturales del lenguaje. Trabajar con ZODB es casi tan sencillo como trabajar con diccionarios. Esto es un gran beneficio para los desarrolladores ya que quita por completo la necesidad de manejar, diseñar, implementar y mantener un modelo diferente en un paradigma también diferente. De esta manera el costo de producción de una pieza de software se reduce considerablemente, ya que ZODB interfiere mínimamente en el proceso de desarrollo.

Anexo – ODBMS en otros lenguajes

Dado que ZODB depende centralmente en el funcionamiento de serialización interno de Python (pickle), por ello su funcionamiento está exclusivamente ligado a Python. Esto hace que su funcionamiento sea natural al lenguaje pero a su vez queda ligado a Python por completo.

Esto no significa que no existan ODBMS para otros lenguajes de igual o similar funcionalidad.

Si bien el listado es amplio y variado, para este trabajo se ha considerado uno de los lenguajes orientado a objetos más populares actualmente, Java.

Dentro de las opciones para este lenguaje se ha escogido la implementación de mayores prestaciones y robustez así como menor restricción de su uso, Db4objects.

Db4objects

Db4O es la versión open source de la base de datos de objetos desarrollada por Versant. Por lo tanto su licencia de uso es GPL, permitiendo el uso de una licencia Comercial para casos de productos que lo requieran. A su vez, está respaldada por una compañía con experiencia en el campo.

Además de todas las propiedades comunes de una base de datos de objetos, provee varios mecanismos para consultar los objetos almacenados.

Una de sus mayores ventajas es que no requiere de una Api adicional para su funcionamiento. De igual manera que ZODB, haciendo uso de *reflection*, db4objects permite almacenar objetos nativos de Java con poco o ningún cambio.

Otra de sus ventajas es la posibilidad de poder utilizar la misma librería en .Net haciendo uso de la misma Api.

Crear e insertar

Un ejemplo de cómo crear una base de datos y almacenar una instancia de un objeto requiere unas pocas líneas.

Dado que db4objects utiliza un sistema completamente nativo (reflection) para almacenar los objetos, es posible almacenar objetos de cualquier tipo o clase, sin requerir modificación alguna, esto incluye listas, clases con herencia y atributos complejos. La definición de la clase no necesita nada en especial para poder ser persistente. Solo los métodos "*getters*" y "*setters*". Aunque para algunos casos se pueden definir propiedades y métodos especiales para mejorar el rendimiento de db4objects.

```
1.  public void store(Engine engine){
2.       ObjectContainer db =
3.          Db4o.openFile("car.yap");
4.       db.set(engine);
5.       db.commit();
6.       db.close();
7.     }
```

Recuperar

Para recuperar un objeto de la base de datos, db4o provee de varios métodos, uno de ellos es “Query by Example”. El cual utiliza una instancia prototipo para seleccionar objetos almacenados que sean similares.

Por ejemplo, para seleccionar un objeto de la clase piloto, según el nombre, podemos usar el siguiente código.

```
1. Pilot proto = new Pilot("Michael Schumacher", 0);
2. ObjectSet result = db.queryByExample(proto);
3. for (Object o: result){
4.     System.out.println(o);
5. }
```

Por otro lado, para obtener la lista completa de objetos de la misma clase solo se necesita una línea de código.

```
1. ObjectSet result = db.queryByExample(Pilot.class);
```

Actualizar

Para actualizar solo se debe volver a indicar a la base de datos que debe almacenar el objeto modificado. Por ejemplo, de la siguiente manera:

```
1. ObjectSet result = db.queryByExample(new Pilot("Michael Schumacher",
   0));
2. Pilot found = (Pilot) result.next();
3. found.addPoints(11);
4. db.store(found);
```

Eliminación

Para quitar objetos de la base de datos solo es necesario utilizar el método *delete*() de la base de datos.

```
1. ObjectSet result = db.queryByExample(new Pilot("Michael Schumacher",
   0));
2. Pilot found = (Pilot) result.next();
3. db.delete(found);
4. System.out.println("Deleted " + found);
```

Otros métodos de búsqueda

Db4o también provee dos métodos extras para buscar elementos almacenados, que son más complejos pero permiten una mayor flexibilidad.

Uno de ellos es “SODA” que permite la escritura de sentencias muy precisas de manera rápida.

```
1. Query query = db.query();
2. query.constrain(Pilot.class);
3. Query pointQuery=query.descend("points");
4. query.descend("name").constrain("Rubens Barrichello")
5.     .or(pointQuery.constrain(99).greater()
6.         .and(pointQuery.constrain(199).smaller()));
7. ObjectSet result=query.execute();
```

Por último también permite lo que es conocido como “Native Query”, los cuales utilizan los mismos objetos, propiedades y métodos para seleccionar las instancias de una base de datos. Esto funciona de manera similar a la metodología de filtrado para ZODB (o usando la función *filter* de python).

```
1. List <Pilot> result = db.query(new Predicate<Pilot>() {
2.     public boolean match(Pilot pilot) {
3.         return pilot.getPoints() > 99
4.             && pilot.getPoints() < 199
5.             || pilot.getName().equals("Rubens Barrichello");
6.     }
7. });
```

Como se puede ver, esto permite además la inclusión de código arbitrario, por lo que es más flexible y potente.

Desventajas

Db4objects es una versión reducida de la base de datos comercial de Versant, por lo tanto no posee todas las capacidades ni el mismo el mismo rendimiento.

Algunas de sus desventajas son las siguientes.

Uno de los problemas, es que db4objects es incapaz de realizar un live rollback, esto es, cancelar una transacción y volver el estado de los objetos ya cargados a su estado anterior. Todos los objetos cargados deben ser descartados y deben recargarse.

Otra de las desventajas se debe que a diferencia de ZODB, donde las clases a persistir heredan de una clase de ZODB, al trabajar sobre objetos nativos de Java, db4objects no es notificado cuando se accede o modifica alguna de las propiedades de un objeto. Esto imposibilita que db4objects actualice automáticamente los objetos modificados. Como se mostró anteriormente es necesario dar la orden “*save*” a la base de datos con el objeto modificado.

El problema se repite en cascada para los atributos de una instancia modificada. Db4objects no puede identificar que objeto requiere ser persistido (de manera automática y previa a la orden *save*) por lo que al guardar un objeto, esto puede desencadenar la persistencia de todo el árbol de objetos que se encuentran por debajo (update depth).

Por otro lado, aunque la situación es diferente, el mismo problema se presenta al intentar recuperar (activate) un objeto de la base de datos. Al contrario que en ZOBD, db4objects no puede cargar los objetos, ni los elementos de una lista, sólo al momento de ser accedidos, por lo tanto, cargar un objeto trae como consecuencia cargar todo el árbol de objetos a partir de este. Lo que podría significar cargar todo el árbol de objetos o gran parte de éste.

Indefectiblemente esto acarrea grandes problemas de persistencia. Aunque aún con estos problemas db4objects no es mucho más lenta que una base de datos relacional, el sistema incluye maneras de solventar o aminorar este efecto y mejorar así el rendimiento.

La forma más flexible para solucionar el problema de Update Depth es utilizar el método *save* con un parámetro extra que indica cuantos niveles se deben persistir.

```
1. db.store(object, depth)
```

Para el caso de la activación de objetos (activate), una manera flexible es utilizando la configuración global de db4objects, ajustando para cada clase, los niveles de persistencia. De la manera ejemplificada a continuación.

```
1. EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
2. config.common().objectClass(Pilot.class).minimumActivationDepth(1);
3. config.common().objectClass(Pilot.class).maximumActivationDepth(2);
```

Alternativas

Una buena alternativa a db4objects en el caso de no requerir licencia GPL es ObjectDB que dispone de una sencilla Api usando JPA estándar. Esto requiere una mínima modificación a la declaración de las clases, pero está exenta de los problemas anteriormente mencionados por db4objects. Así mismo cuenta con un rendimiento superior, y un conjunto de herramientas que ayudan al desarrollo y mantenimiento.

Está disponible con algunas limitaciones para uso no comercial, y para uso comercial debe pagarse una licencia.

Bibliografía

Christine Weiss. *Selecting a Database Management System (DBMS) – A Practical and Quasi-Technical Analysis of Relational Database Management Systems (RDBMS) and Object Database Management Systems (ODBMS)*. University of Colorado at Colorado Springs

McClure, Steve. *Object Database vs. Object-Relational Databases.* IDC Bulletin #14821E: Agosto 1997

Dr. Andrew E. Wade, Ph.D. *Hitting the relational wall.* Objectivity Inc.

Robin Bloor. *The Failure of Relational Database, The Rise of Object Technology and the Need for the Hybrid Database.* Baroudi Bloor, 2004. **http://www.intersystems.com/cache/whitepapers/pdf/baroudi_bloor.pdf**

Pascal Hauser. *Review of db4o from db4objects.* University of Applied Sciences Rapperswil, Suiza.

Scott W. Ambler. *Mapping objects to relational databases.* IBM. Julio 2000. **http://www.ibm.com/developerworks/webservices/library/ws-mapping-to-rdb/**

Ryan Somma. *The O/R Problem: Mapping Between Relational and Object-Oriented Methodologies.*

Kybele Research Group. *Aggregation and Composition in Object – Relational Database Design*. España (Madrid): Rey Juan Carlos University.

Noah Gift; Brandon Rhodesmill. *Example-driven ZODB*. [s.l.] developerWorks (IBM), Abril 2008.

Jim Fulton. *Introduction to the Zope Object Database*. Digital Creations.

A.M. Kuchling. *ZODB/ZEO Programming Guide*. Enero 2005.

Carlos de la Guardia et al. *ZODB Book Documentation*. Agosto 2011.

Otras fuentes de información utilizada

ZOPE
http://www.zope.org/

The Pylons Project
http://www.pylonsproject.org/

Db4Objects (Versant)
http://www.db4o.com/

Versant Object Database
http://www.versant.com/products/Versant_Database_Engine.aspx

Object Database Management Systems
http://www.odbms.org/

Plone
http://plone.org/

Grok
http://grok.zope.org/

ObjectDB
http://www.objectdb.com/

ZODB Documentation
http://zodbdocs.blogspot.com

Zope Wiki
http://wiki.zope.org/ZODB/guide/index.html

Running a ZEO server
http://wiki.zope.org/ZODB/howto.html

Related Modules – ZODB v3.10.3 documentation
http://www.zodb.org/documentation/guide/modules.html

The official ZODBWiki
http://wiki.zope.org/ZODB/Documentation

FileStorageBackup
http://wiki.zope.org/ZODB/FileStorageBackup

UnixSpace
http://unixspace.com/context/databases.html

Learn the cutting-edge Grok Python Web framework by following this tutorial
http://grok.zope.org/tutorial.html

A simple ZODB-based Grok Web application example
http://grok.zope.org/documentation/tutorial/adder-an-adding-machine-sample-application/tutorial-all-pages

Website for Plone, a Python, ZODB-based Content Management System
http://grok.zope.org/documentation/tutorial/adder-an-adding-machine-sample-application/tutorial-all-pages

ZODB ZEO Raid
http://grok.zope.org/documentation/tutorial/adder-an-adding-machine-sample-application/tutorial-all-pages

Jim Fulton's paper on ZODB.
http://www.python.org/workshops/2000-01/proceedings/papers/fulton/zodb3.html

Zodb undo and conflict resolution
http://lonetwin.net/20100828/hacks-you-can-live-without/zodb-undo-and-conflict-resolution/

Advanced ZODB for Python Programmers
http://www.zodb.org/documentation/articles/ZODB2.html

ZODB Benchmarks revisited — Upfront Systems
http://www.upfrontsystems.co.za/Members/roche/where-im-calling-from/zodb-benchmarks-revisited

ZODB vs Relational Database: a simple benchmark
http://pyinsci.blogspot.com/2007/09/zodb-vs-relational-database-simple.html

The Failure of Relational Database, The Rise of Object Technology and the Need for the Hybrid Database
http://www.intersystems.com/cache/whitepapers/hybrid.html

ODBMS v. RDBMS, Choosing an ODBMS, FAQs - Objectivity/DB
http://www.objectivity.com/pages/objectivity/faq.asp

Object vs Relational Database Concepts
http://www.qint.de/joria/doc/objectsvsrecords.html

Python Glossary
http://docs.python.org/glossary.html#term-duck-typing
http://docs.python.org/glossary.html#term-eafp