Melta: Prototipo de una base de objetos en Python *

Ernesto Bossi†
Universidad Tecnológica Nacional - FRBA
Medrano 951
Buenos Aires, Argentina
bossi.ernestog@gmail.com

ABSTRACTO

Melta es un prototipo, que tiene como objetivo implementar una base de datos orientada a objetos, utilizando el modelo de una propuesta para una nueva estandarización, de este tipo de bases de datos [9]; y haciendo modificaciones, extensiones sobre él. Aportando de esta manera, con una nueva implementación de este tipo de motores de base de datos en la comunidad. En la sección 1, se introducirá la problemática y los fundamentos de porque se implemetará Melta. En la sección 2 y 3, se tratará la arquitectura principal y el modelo de objtos que existirá en la base de datos. En la sección 4, se profundarizará en lo que es el lenguaje de AOQL. En la sección 5, se detallará brevemente la persistencia, capítulo 6 las transacciones y concurrencia, y en la sección 7 las conclusiones y una propuesta de futuras mejoras y nuevas funcionalidades al sistema.

1. INTRODUCCIÓN

Con el crecimiento de base de datos no clasicos (base de datos no-SQL), especialmente en el contexto de rápido crecimiento de internet, el problema de procesamiento de datos en sistemas basados en lenguajes orientados a objetos, radica en el procesamiento previo que tienen los datos persistidos desde que se consultan a una base de dato relacional, hasta su transformación en objetos. Estos pasos de procesamiento implican el uso de herramientas, a veces complejas como los ORM's, que son utilizadas para resolver lo que se conoce como *impedance mismatch*. Las bases de datos orientadas a objetos resuelven estros problemas, y se presentan otros, tales como la sincronizcación entre los objetos del dominio o sistema y el de la base de datos. En general las bases de datos orientadas a objetos han sido especialmente populares en lenguajes como Smalltalk, en donde los objetos estan vivos mientras el sistema este en ejecución, esa ventaja permite tener un modelo de objetos a persistir en una etapa intermedia. Esto no se presenta en otros lenguajes de programación. El objetivo de este paper será el de proponer un prototipo para un lenguaje dinámico y e interpretado. Para ello se modelo un ecosistema, que será llamado *schema*, en el que existirán los objetos a persistir propios de un sistema. En la siguiente sección se tratará en más detalle como es la arquitectura general de este prototipo. La decisión del lenguaje en el que se implementará dicho portotipo fue Python, ya que permite un rápido modelado del sistema, y se podrá optimizarse partes del mismo utilizando los bindings de C.

De esta manerá se podrá analizar, si es viable continuar con el desarrollo del sistema en dicho lenguaje o utilizar otro que posea mejor soporte para paralelismo. Con respecto a lo último, Java o Scala, resultan como posibles candidatos para una implementación a futuro.

2. ARQUITECTURA

El modelo de este prototipo, será cliente/servidor, en donde el cliente consistirá en un módulo que permitirá la conversión de objetos Python/Melta, un procesador de AOQL,una interfaz que mantendrá la relación entre un objeto Python y el id correspondiente al objeto en la base de datos (se profundizará en este tema más adelante en esta sección), y un conector que permitirá conectarse con la base de datos. Si bien el formato de los dos tipos de objetos no es el mismo, los objetos de la base de datos, tienen conocimiento de la clase a la que pertenece el objeto, su jerarquía, y el estado que tiene. El metamodelo de objetos de Melta, se describirá en la sección 3 de este artículo.

El modelo de los objetos que existirán en el entorno Python en ejecución, y que estará del lado del cliente, tendrá una tabla que relacionará cada uno de los objetos que hayan sido agregados a la base de datos, con el id del objeto Melta, que existirá del lado del servidor. De esta manera se conocerá la relación entre ambos objetos en el cliente y el servidor. Esta relación existirá por el tiempo que el entorno del dominio esté en ejecución una vez terminado la misma, los objetos que ya no se utilicen en la base de datos y estén presentes en el *schema*, serán persistidos y borrados de este, hasta que sean requeridos nuevamente. Como se menciono previamente, como el formato de los objetos del dominio de Python y de Melta no son compatibles, se tendrá un conversor de objetos para poder hacer el pasaje del estado de los objetos de un lado al otro.

*ODBMS en desarrollo
†Ing. de la UTN FRBA.

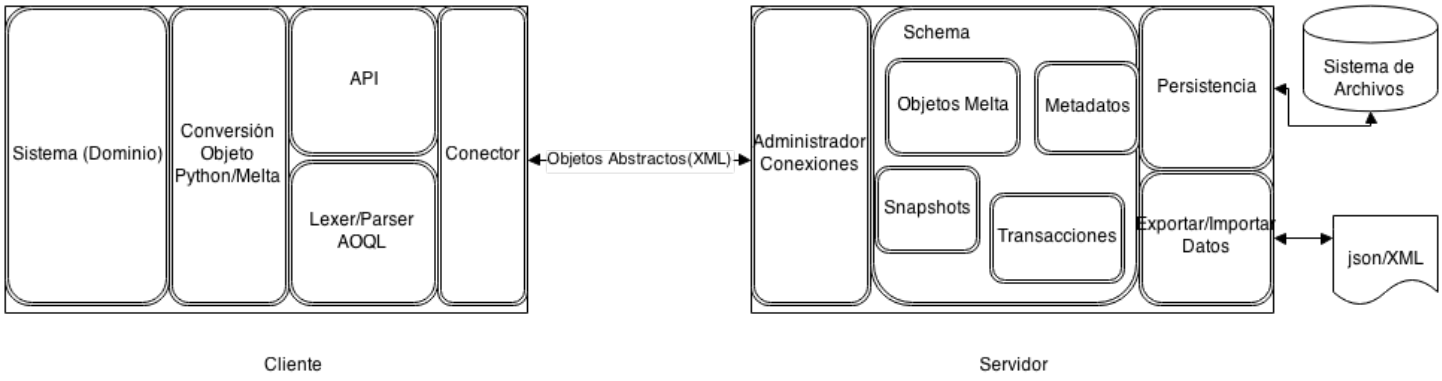

Figure 1: Arquitectura General de Melta

El cliente realiza operaciones de consulta y actualización sobre el servidor, lo puede realizar mediante una interfaz, implementada en el mismo lenguaje que el cliente o en un lenguaje abstracto llamado AOQL. Cuando se requiera de objetos persistidos, se hará una consulta al servidor, en caso de que sea mediante el lenguaje AOQL el mismo se preprocesará para que luego sea interpretado por medio de una API, en el caso de la interfaz, esta interactuará directamente con esta API. La API de consulta hara el requiermineto formal al servidor, este procesará las operaciones, y en caso que sea valida, se procesará, y se restaurarán aquellos objetos que no esten en el *schema* y que estén involucrados, luego se devolverá al cliente los objetos o el resultado de dicha operación. Se tratará mas exhaustivamente esto en la sección 4.

El Servidor consistirá basicamente en un módulo que estará escuchando nuevas conexiones, intercambiando datos con los clientes, tendrá un entorno, llamado *schema*, donde vivirán los objetos de la base de datos, y un modulo que se encargará de realizar la persistencia, y reconstruir las instancias, clases y metadatos del disco al schema y consecuentemente servirlos al cliente.

El servidor maneja la sincronización de las operaciones de lectura/escritura, mediante transacciones, que administra el servidor, si bien son disparadas por el cliente, el servidor mantendra la información sobre las transacciones que se estan llevando a cabo en el servidor. Se tratará con mas detalle este tema, en la sección de concurrencia y transaccionabilidad (sección 6). Otro componente del servidor es el de que pueda persistir o exportar la base de datos completa a otros formatos, tales como json o xml.

La comunicación entre el cliente y el servidor se realiza, en esta primera versión, serializando los objetos en XML, ya que permite de una manera simple intercambiar instrucciones entre estos a través de una red de comunicaciones. En una próxima versión se tratará de implementar un protocolo específico para serializar los objetos y operaciones que el servidor deberá manejar. Con respecto al XML, se presentó el problema de que pueda existir una posible colisión de nombres, por ejemplo, puede existir una Clase teléfono y una instancia de otra clase que tenga un estado teléfono. Para resolver esto, se utilizan los namespaces de XML [11], que proveen una manera de calificar nombres de atributos y elementos en documentos XML, asociandolos con namespaces identificados por referencias de *Unified Resource Identifier* URI.

3. METAMODELO DE OBJETOS

Este paper esta basado en gran parte en el paper de la OMG [9], en el que se podrá hacer referencia para conocer más sobre esta sección. El modelo de objetos de Melta se diferencia del de los objetos nativos de Python, en disasociar los objetos con estado en un objeto compuesto, y sus estados como otros objetos atómicos, que pasarían a ser inmutables, o sea, ante el cambio, un objeto atómico no se actualiza sino que se crea uno nuevo y se desecha el anterior. Esto permitirá que pueda darse soporte en el futuro a objetos versionados, y facilita la restauración de un estado que esta asociado a una transacción ante un fallo de sincronización o *rollback* de la transacción. Dichos objetos, poseen un ID, único, generado en la creación inicial del objeto que se da cuando una instancia o valor es ingresado al *schema*. Este ID, permitirá identificar unívocamente al objeto en el *schema*, asi como la asociación entre ID de objeto de Python/Melta.

Los objetos que se encontrarán en el *schema* o ecosistema de Melta en el servidor serán de estos tres tipos:

1. **Objeto Atómico**: <*i,n,v*> Son objetos, donde i es el ID del objeto, n es el nombre que tendrá el objeto, que será el nombre del atributo de la instancia de Python, y v el valor asociado al objeto.

2. **Objeto Compuesto**: <*i,n,T*> Son objetos donde i es el ID, n el nombre de la instancia de Python y T, una lista de objetos atómicos o de referencia, que componen el objeto.

3. **Objeto Referencia**: <*i,n,r*> donde i es el ID del objeto, n el nombre de la instancia referida de Python y r, es el objeto Melta al que se hace referencia. Este tipo de objetos se generan, cuando una instancia tiene en su estado una referencia a otra instancia y representan este estado.

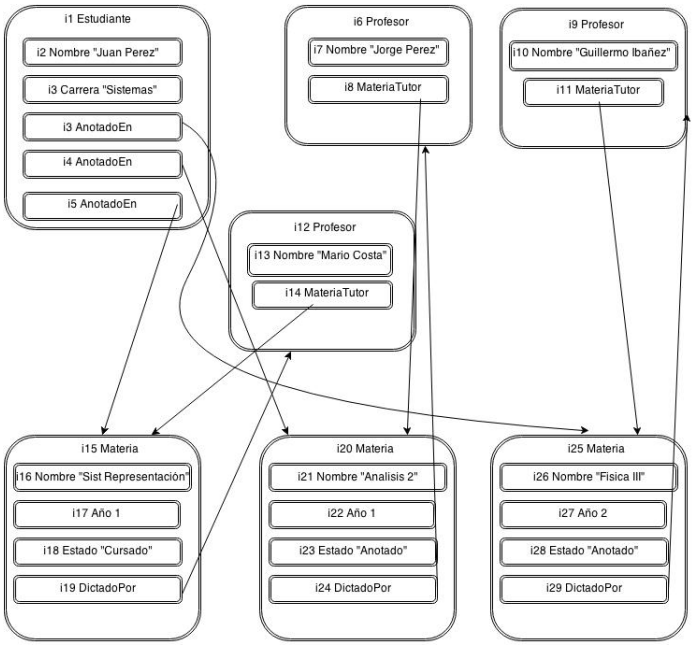

Figure 2: Ejemplo de un schema de objetos en Melta.

4. **Objeto Clase**: $<i,c,M,A>$ donde i es el ID del objeto, c es el nombre de la Clase python, M es una lista de métodos que posee la clase y A es una lista ordenada con los ancestros de la clase que se representa en objeto. Las clases poseen invariantes, tales como los métodos de instancia que lo componen, el código que compone cada uno de los métodos, se serializa en un *string* formateado, para respetar en el caso de Python, por ejemplo, la identación. Cuando se desee regenerar la clase nuevamente al levantar el sistema, se evaluarán estos *strings* con el nombre del método almacenado y sus argumentos. Para los métodos de clase, se ha optado por tener un objeto que sea Metaclase.

5. **Objeto MetaClase**: $<i,c,C,MC,A>$ donde i es el ID del objeto, c el nombre de la Metaclase Python, C es el objeto clase de la base de datos asociada a dicho objeto Metaclase, MC es una lista de métodos de clase que suministrará al objeto clase C, y A es una lista de los ancestros del objeto Metaclase y son otros objetos Metaclase. Notese que la jerarquía de ancestros de C es la misma que la del objeto en cuestión, solo que el del este último son de objetos metaclase y no de objetos clase (Figura 3).

Como se menciono en la sección anterior, dichos objetos se encontrarán contenidos en el *schema*, y estos representarán a aquellos objetos que se han recuperado de la base de datos y están en estado de transición durante el tiempo y que pueden variar de acuerdo al flujo del sistema de dominio (Figura 2).

Finalmente, todo objeto posee metadata asociado a el, en donde se define datos tales como la fecha de creación, modificación, si esta asociado a un *schema*, la clase a la que pertenece, y si posee transacciones asociadas, entre otras cosas.

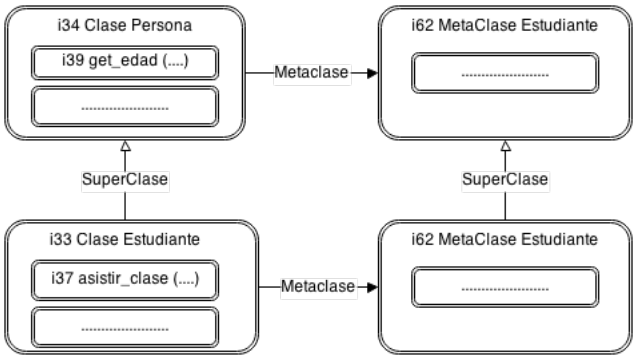

Figure 3: Ejemplo de la jerarquía de clases y metaclases.

3.1 Estructura del Schema

El *schema*, se compone por todos los objetos y clases que se hayan agregado a la base de datos, y además de eso, soporta lo que es la herencia dinámica. En el caso en el que tenemos un objeto de una clase que cambia luego su clase asociada por otro de la misma jerarquía, la base de datos debería evitar una reificación del objeto, descartando el viejo, y creando uno nuevo antes, a partir del estado del anterior, ya que se perdería la identidad del mismo.

Por ejemplo, si tenemos una clase Estudiante, que hereda de una clase Persona, y un objeto de este tipo (Estudiante); se tendría una instancia de este tipo con el estado propio de esta clase y otra instancia del tipo Persona con el estado que posee esta clase. En este caso el objeto del tipo Estudiante tendra una referencia al objeto de la clase Persona, que contiene parte de su estado. Esto rompe el principio de sustitución de Liskov, ya que no se podrá utilizar el objeto Estudiante, cuando se use en cambio el objeto Persona, ya que en este caso, Estudiante no sería más un subtipo de Persona.Esto se resuelve por lo explicado en [9], mediante la implementación de roles, que hacen uso de la composición. Bajo este esquema, una aplicación puede definir subclases a partir de una clase, como en el caso de Estudiante-Persona, y cuando se necesite un objeto del rol Estudiante, el objeto, de la base de datos, de esta clase puede ser creado y ser asociado con el objeto de la superclase y ser disasociado, cuando ya no este actuando en este rol.

Ya explicada la herencia del *schema*, se puede definir formalmente al mismo como una tupla $<O,C,R,CC,OC,OO>$, donde O representa a un hash o diccionario de objetos, del tipo {i : o}, donde i es el ID del objeto, y o es la referencia del objeto en si mismo; C es una lista de las clases presentes en el ecosistema, R son los objetos raices del *schema*, estos objetos, son aquellos que pertenecen a instancias de objetos Python. CC es una tabla de hash o dicccionario, del tipo {c1 : A}, que relaciona una clase *c1* con su lista de ancestros *A*. OC es un diccionario que relaciona lo que es un objeto con su clase. Notese que CC y OC, son datos que poseen estos objetos en sus respectivas estructuras de metadatos, y solo se vuelven a tener en el *schema*, como tabla maestra. OO define una relación de lo

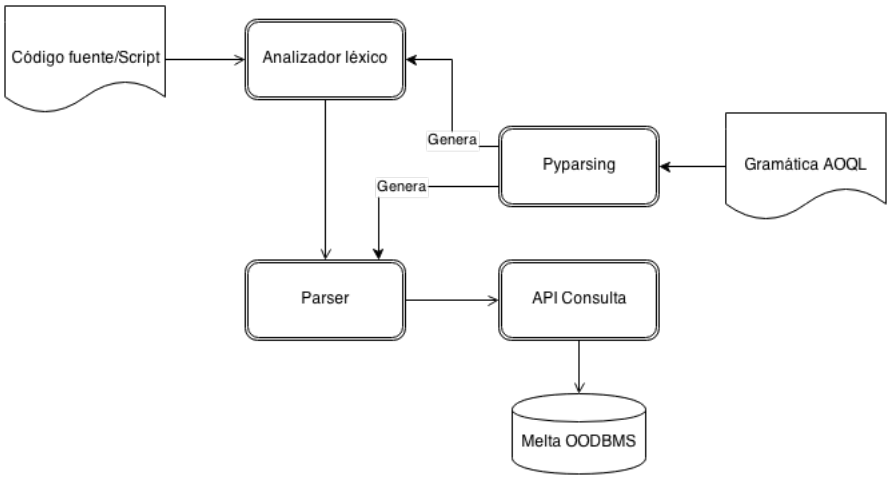

Figure 5: Estructura del procesamiento del lenguaje de AOQL.

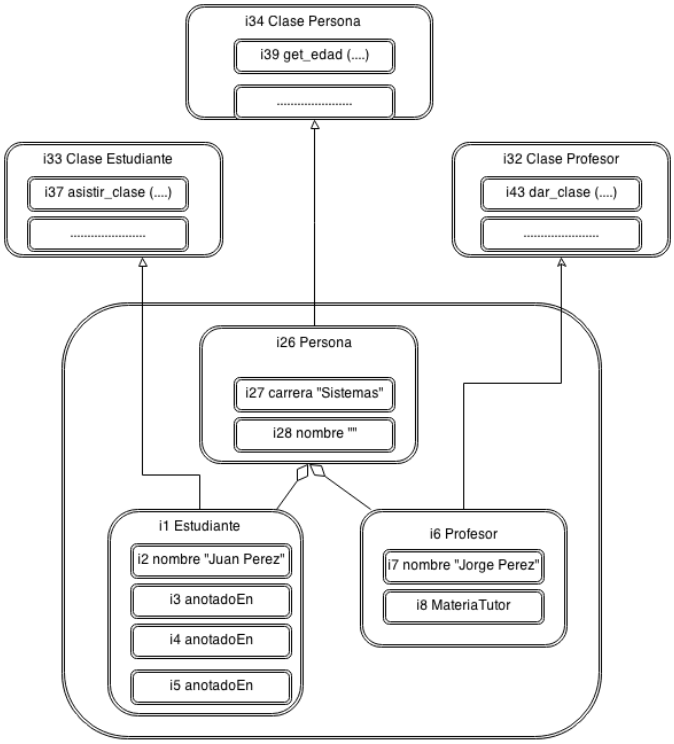

Figure 4: Herencia dinámica por medio de Roles.

que es la herencia dinámica entre objetos y roles. Por ejemplo si {i1:i2} ∈ OO, el objeto rol identificado como i1 esta heredado dinámicamente con el objeto identificado como i2. Un ejemplo de esto se ve en la Figura 4

Con respecto a lo que es la actualización del sistema, el mismo se resolverá de acuerdo al mecanismo de transacciones, como se mecionó en la sección anterior.

4. LENGUAJE DE CONSULTA AOQL

El lenguaje de consulta de AOQL (*Abstract Object Query Language*), es vital para el acceso y persistencia de los datos que utilizará el sistema a lo largo de su ejecución. El mismo podrá ser utilizado en una consola, que actuará como cliente, como en el proyecto en el que se esta utilizando la base de datos. Este lenguaje de consulta definirá las operaciones necesarias para el uso de un cliente con la base de datos.

Por otra parte el cliente no solo tendrá la posibilidad de comunicarse por medio de un lenguaje de consulta externo, sino que se ofrecerá la posibilidad de que puedan accederse por medio de sentencias en el mismo lenguaje de programación, que el sistema de dominio. De esta manera los programadores podrán hacer consultas sin tener que recurrrir a archivos externos donde esten las consultas, o a tener incluso las mismas mezcladas con el lenguaje de programación concreto del dominio. También se puede ver como no se necesitará de un parser previo para traducir la sintaxis del lenguaje AOQL, a las instrucciones Python propias de la API de consulta, en Python, que consultará a la base de datos.

Se puede ver como ambas maneras de acceder a los datos, son coincidentes con lo que son los DSL's (*Domain Specific Language*) externos e internos. En el que el lenguaje AOQL es un DSL externo, que requerirá una gramática asociada, ya que es independiente del lenguaje en el que se implementará la base de datos y el lenguaje en donde existirá el sistema que lo utilizará; y a su vez también tendrá un analizador léxico y un *parser*, que puedan procesar el lenguaje

Table 1: Gramática de AOQL v1

Regla gramaticales	Nota
query ::= literal	lista L
query ::= name	lista N
query ::= operadorAlgUnario query	operadores algebráicos unarios
operadorAlgUnario ::= count | min | sum | max | $-$ | $sqrt$ | not | avg |	
query ::= query operadorAlgBinario query	
operadorAlgBinario ::== $\|<\|>\|+\|-\|*\|/\|$ and | or | $intersect$ | ...	
query ::= query SinOperadorAlg query	Operador no algebráico.
SinOperadorAlg ::= where | . | $join$ | $\forall$ | $\exists query$ ::= $\forall queryquery$ | $\exists queryquery$	Sintáxis alternativa de los identificadores.
query ::= query as name Name	Definición
query ::= query group as name	Agrupamiento
query ::= if query then query	Condicional
query ::= if query then query else query	Otra sintaxis de Condicional
querySeq ::= query | $query, querySeq$	Lista o secuencia de queries
query ::= struct(querySeq) | $(querySeq)$	Structure constructor
query ::= list( ) | $list(querySeq)$	Constructor de Lista
query ::= set( ) | $set(querySeq)$	Constructor de Set
query ::= (role$_name$)$query$	Casteo dinámico para retornar un rol en particular de un dataSet.
query ::= query has role role$_name$	consulta de existencia del rol.

y traducirlo a instrucciones de la API (figura 5). En cambio las consultas hechas con comandos en un lenguaje de programación, en este caso será Python ya que se desarrollo el cliente en dicho lenguaje, directamente harán referencia a estas instrucciones de la API, y solo será un azucar sintáctico dicho DSL interno.

En un futuro debería definirse un pequeño motor de optimización de consultas para minimizar la cantidad de operaciones hacia el servidor y de esta manera tener un menor tiempo de espera, en obtener datos o resultados por parte del cliente.

4.1 Gramática del lenguaje AOQL

El lenguaje de AOQL, en su primera versión estará conformada por la gramática descripta en la tabla 1.

Dicha tabla estaba basada en el trabajo en [9], con algunas pequeñas modificaciones. En el futuro, se podría pensar una extensión de los tipos de colecciones para soportar otros lenguajes de programación que hagan uso de este motor. En cuanto a la implementación, se implemento el lenguaje utilizando una herramienta que permite generar un analizador léxico y un *parser* a partir de la gramática definida en la tabla. Luego de eso, se armo un intérprete directo que hace uso de la API de consulta contra la base de datos.

5. PERSISTENCIA

Melta posee un modulo de persistencia en donde se almacenarán físicamente los objetos pertenecientes al *schema*, una vez que se haya hecho un *commit* en laa transacción. En la primera versión solamente se persistirán los cambios detectados en el sistema hacia el *schema*, solo cuando esto suceda.

Los objetos en Melta, estan identificados por un ID único de 128bits de longitud, generado cuando se introduce un objeto en la base de datos, y el mismo tiene un formato similar al de UUID, definido en la RFC4122 **??**. Este identificador unívoco se utilizará en el archivo donde se almacenarán a aquellos objetos que no estén siendo utilizados durante la ejecución. Cuando se solicite un objeto, por parte de un cliente, que no exista actualmente en el schema, se procederá a leerlo del archivo en donde esta almacenado, y un módulo de extracción convertirá el objeto serializado en formato Melta, asimismo, se extraerá la metadata del mismo que tabien ha sido serializado. Cuando se reconstruye el objeto, se chequeará si la clase o rol que tiene asociado actualmente existe en el *schema*, si no existe se traerá el mismo también.

Con respecto al formato del archivo, actualmente esta en desarrollo, pero se tendrá probablemente una primera versión del mismo, con una arquitectura, en la que la persistencia de un *schema*, se haga en dos archivos separados, el primero será de extensión .mlt, donde se almacenarán todos los objetos pertenecientes al *schema*, una vez que se transaccione, el mismo estará compuesto por una cabecera de 8192 bytes, que contendrá una *binary signature*, que procurará la integridad de los datos a cierto punto (esto depende del algoritmo que se emplee para este propósito), un número de versión, la metadata propia del *schema*, cantidad de objetos persistidos, el tamaño del archivo total, una lista de punteros libres, y el puntero inicial donde comienza el primer objeto. Luego se tendrá un directorio, donde estará el puntero de cada uno de los objetos que conforman el archivo. El mismo es de una longitud fija de 8192 bytes, y en caso de que se supere dicho límite, siempre tendrá una referencia al próximo bloque del directorio, y de esta manera, se comporta como una lista secuencial de bloques.

Luego del directorio se tendrán los objetos, de longitud variable, de acuerdo a lo que ocupen, cuando se agrega un nuevo objeto se agrega al final del archivo, y cuando se elimina uno, se libera el bloque del disco, y se agrega el puntero a la lista de punteros libres. El problema que existe es análogo al que sucede con un asignador de memoria, en el que habrá fragmentación a medida que pase el tiempo, para remediar esto,

en primera medida, cada X (este valor es configurable) veces que se monte el filesystem, cuando se inicializa la base de datos, se compactará el archivo y se actualizará el directorio. Una mejora a esto sería que en vez de agregar siempre al final se tenga un algoritmo de first-fit, y de esta manera se reduzca el espacio libre fragmentado, sin embargo, se deberá hacer una compactación luego de un tiempo.

El segundo archivo contendrá los datos de indexación de los objetos, y este archivo se levantará en su totalidad cuando se inicialice el motor de base de datos, su objetivo es obtener rápidamente los objetos a restaurar por clase, y la reindexación la manejará automáticamente el modulo de persistencia y recuperación. Este archivo solo se persistirá cuando se guardan los cambios en la base de datos, y al estar levantado todo el índice en memoria, no existirá fragmentación.

Finalmente cuando una transacción global sea exitosa, se guardará los cambios en el *schema*, consecuentemente hay una manera de decir al sistema de que se desean persistir en disco lo que existe actualmente en el *schema*, y es mediante un comando llamado snapshopt, que persistirá los cambios en el disco.

6. TRANSACCIONES Y CONCURRENCIA

Melta soporta modificaciones concurrentes de objetos, por varios clientes que esten conectado al servidor, toda modficiación, del lado del cliente debe estar enmarcado en una transacción, esto provee dos características importantes:

- **Atomicidad**: Las transacciones son exitosas o falla, y al ser transacciones, no hay estado que se mantenga hasta que se haga *commit* a una transacción.
- **Control de Concurrencia**: las transacciones se ejecutan como si tuviesen acceso exlusivo a los datos, de esta manera, multiples transacciones pueden correr concurrentemente y de esta manera la lógica de la aplicacion no tiene que tomar en cuenta el acceso concurrente.

Las transacciones utilizan el principio de *Optimistic Concurrency Control*, mediante un timestamp. Los cambios que se hacen a uno o más objetos mientras los cambios no esten aplicados, mediante un *commit*, son realizado de manera independiente y del lado del cliente, que mantendrá sus objetos modificados en una cache, y de esta manera no tiene que ser bloqueados, los cambios recien se sincronizarán cuando el cliente haga un *commit* explicito. Solo se permite que una transacción aplique los cambios en un momento determinado. Si hay dos threads, o clientes, que modifican un el mismo objeto, uno de ellos aplicará los cambios primero. Cuando el segundo aplique los cambios, el cambio que aplique será inválido y se le retornará una excepción al mismo. El dominio o aplicación debería atrapar esta excepción y ejecutar de nuevo la aplicación de los cambios, cuando esto suceda, los estados de los objetos afectados, reflejan cambios hechos por las transacciones ya exitosas en el servidor.

7. CONCLUSIONES

Si bien este paper es una versión temprana de un prototipo de una base orientada a objetos, en esta primera versión se definió una primera aproximación de una arquitectura para un model cliente/servidor básico. Los próximos objetivos, para tener un prototipo más completo, son los siguientes:

- Objetos versionados.
- Replicación.
- Bases de datos federadas.
- Optimización del Motor y de sentencias AOQL.
- Acceso cruzado a otras bases de datos orientas a objetos.
- Proponer un protocolo para comunicación de objetos y operaciones entre cliente/servidor.
- Tolerancia a fallos, por interrupción del servicio causado por fallas electricas o de red.

Actualmente se esta desarrollando una implementación de este protoripo, como se mencionó en la introducción, y el objetivo es implementar la arquitectura propuesta en este documento. Los siguientes pasos involucrarían la optimización del motor y de testear el mismo con una mayor carga de datos a fin de evaluar si es viable el uso de este motor en ambientes productivos. Si bien algunas de las partes, como el metamodelo son basadas fuertemente en [9], se hicieron algunas modificaciones sobre este, y se diseño una arquitectura simple para una primera versión de este motor de base de datos.

8. AGRADECIMIENTOS

Agradecimientos en especial a la OMG, por su propuesta de un nuevo estandar, que me dio el interés para investigar y desarrollar este prototípo en esta tecnología emergente.

9. REFERENCIAS

[1] R. S. Alfred V. Aho and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 2007.

[2] M.-M. B. Andrew Gaylard, Brent Pinkney. The magma database file formats. May 2006.

[3] A. V. Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN NOTICES*, 35:26–36, 2000.

[4] T. K. S. Huan-Chao Keh. Formal specification in software reuse designs: an object-oriented database example. *Tamkang Journal of Science and Engineering*, 1(2):97–113, August 1998.

[5] A. Jodlowski. *Dynamic Object Roles in Conceptual Modeling and Databases*. Ph. D. Thesis, 2002.

[6] B. T. Klaus-Dieter Schewe. Principles of object oriented database design. March 1997.

[7] P. Leach. Rfc4122: A universally unique identifier (uuid) urn namespace. *Network Working Group*, July 2005.

[8] K. S. Michał Lentner. Odra: A next generation object-oriented environment for rapid database application development. *Lecture Notes in Computer Science*, 4690:130–140, June 2007.

[9] OMG. Next-generation object database standarization. *Mars*, September 2007.

[10] N. P. Steve Freeman. Evolving an embedded domain-specific language in java. In *OOPSLA '06 Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 855–865. ACM, March 2006.

[11] A. L. R. T. Tim Bray, Dave Hollander. Namespaces in xml 1.1 (second edition). *W3C Recommendation*, August 2006.