CumuloNimbo: Una Plataforma como Servicio con Procesamiento Transaccional Altamente Escalable*

Ricardo Jiménez-Peris, Marta Patiño-Martínez, Ivan Brondino
{rjimenez,mpatino,ibrondino}@fi.upm.es

Facultad de Informática
Universidad Politécnica de Madrid

Resumen El modelo de computación en la nube (*cloud computing*) ha ganado mucha popularidad en los últimos años, prueba de ello es la cantidad de productos que distintas empresas han lanzado para ofrecer software, capacidad de procesamiento y servicios en la nube. Para una empresa el mover sus aplicaciones a la nube, con el fin de garantizar disponibilidad y escalabilidad de las mismas y un ahorro de costes, no es una tarea fácil. El principal problema es que las aplicaciones tienen que ser rediseñadas porque las plataformas de computación en la nube presentan restricciones que no tienen los entornos tradicionales. En este artículo presentamos *CumuloNimbo*, una plataforma para computación en la nube que permite la ejecución y migración de manera transparente de aplicaciones multi-capa en la nube. Una de las principales características de CumuloNimbo es la gestión de transacciones altamente escalable y coherente. El artículo describe la arquitectura del sistema, así como una evaluación de la escalabilidad del mismo.

1. Introducción

Hoy en día muchas aplicaciones se programan con una arquitectura multicapa, generalmente de tres capas, en la que se separa por un lado la capa de presentación, la capa de aplicación y por otro, la capa de almacenamiento de datos. Esta separación implica el uso de distintos sistemas para la programación de cada capa (servidor web, servidor de aplicaciones y base de datos) que se ejecutan en un entorno distribuido y que, a su vez, pueden estar cada uno de ellos replicado para tolerar fallos y escalar a medida que aumenta el número de clientes. En la actualidad muchas aplicaciones se ofrecen en la *nube* ejecutando remotamente, como por ejemplo el correo electrónico o los documentos de gmail. El único requisito para acceder a estos servicios es disponer de una conexión a internet. A este modelo se le denomina *software como servicio* (SaaS). También se han desarrollado *plataformas como servicio* (PaaS) como por ejemplo windows Azure, en las que las aplicaciones se ejecutan remotamente haciendo uso de los servidores, redes y almacenamiento del proveedor de la plataforma.

Este artículo presenta CumuloNimbo[1] [7], una plataforma como servicio (PaaS), para aplicaciones multi-capa que prevé un procesamiento transaccional ultra-escalable proporcionando el mismo nivel de consistencia y transparencia que un sistema de base de datos relacional tradicional. La mayoría de los sistemas actuales recurren al *sharding* para obtener escalabilidad en el procesamiento transaccional. *Sharding* es una técnica en la cual la base de datos se divide en varios fragmentos (particiones) que funcionan como bases de datos independientes compartiendo el esquema de la base de datos original. *Sharding* es técnicamente sencillo pero no es ni sintáctica ni semánticamente transparente. La transparencia sintáctica se pierde porque las aplicaciones tienen que ser reescritas con transacciones a las que sólo se les permite acceder a una de las particiones. La transparencia semántica se pierde, porque las propiedades ACID previamente proporcionadas

* This work was partially funded by the Spanish Research Council (MiCCIN) under CloudStorm TIN2010-19077, by the Madrid Research Foundation, Clouds project S2009/TIC-1692 (cofunded by FEDER & FSE), and CumuloNimbo project FP7-257993.

[1] http://cumulonimbo.eu/

por transacciones sobre conjuntos de datos arbitrarios se pierden. Recientemente se han propuesto alternativas al *sharding* [3],[11], pero son soluciones para estructuras de datos especializadas [3] o no han sido diseñadas para sistemas online que requieren tiempos de respuesta rápidos [11].

El objetivo de la plataforma CumuloNimbo es dar soporte a aplicaciones con procesamiento transaccional online ofreciendo las mismas garantías que las ofrecidas por las bases de datos relacionales tradicionales, es decir propiedades ACID estándar, y al mismo tiempo garantizando una escalabilidad como la de las plataformas que emplean *sharding*. CumuloNimbo proporciona el mismo entorno de programación que un servidor de aplicaciones Java EE, consiguiendo una transparencia sintáctica y semántica completa para las aplicaciones.

2. Trabajo Relacionado

Recientemente se han publicado varios trabajos que proponen protocolos para el procesamiento transaccional en entorno cloud.

En CloudTPS [12] los datos son particionados (*sharding*) para poder escalar el procesamiento de transacciones. Las claves primarias a las que accede una transacción tienen que ser suministradas al inicio de la misma. Microsoft SQL Azure [10] sigue un modelo similar en el que los datos tienen que ser particionados manualmente (*sharding*) y las transacciones no pueden acceder a más de una transacción. ElasTraS [5] es una base de datos elastic en la que las transacciones solo pueden acceder a una partición de los datos.

Google Megastore [1] proporciona transacciones serializables sobre Bigtable. Los programadores tienen que crear grupos de datos y una transacción solo puede acceder a un grupo.

Los principales problemas a la hora de construir una base de datos sobre S3 (el servicio de almacenamiento de Amazon) son presentados en [4]. Los autores argumentan que la propiedad de aislamiento de las transacciones no se puede suministrar ya que el contador necesario para implementar aislamiento *snapshot* se convierte en un cuello de botella y punto único de fallo. El diseño de CumuloNimbo evita estos dos problemas.

Deuteronomy [9] desacopla el almacenamiento del procesamiento transaccional. Las transacciones implementan todas las propiedades ACID y pueden acceder a cualquier conjunto de datos. Adopta un enfoque centralizado para el procesamiento de transacciones que puede convertirse en cuello de botella. En este caso, se pueden desplegar varios gestores de transacciones, pero cada gestor solo puede modifica datos disjuntos.

Percolator [11] es la propuesta de Google para el procesamiento transaccional de grandes cantidades de datos. Al igual que CumuloNimbo proporciona aislamiento*snapshot*. Aunque no tiene interfaz de base de datos relacional y está diseñada para realizar procesamiento batch donde el tiempo de respuesta no es la métrica principal.

CumuloNimbo proporciona la funcionalidad de una base de datos relacional, es escalable, transparente para los usuarios (no hay particionamiento) y tolerante a fallos (implementado en parte por el almacenamiento).

3. Arquitectura de CumuloNimbo

La arquitectura de CumuloNimbo consta de múltiples capas (figura 1), cada una de las cuales tiene una funcionalidad específica. Para conseguir el rendimiento deseado, cada capa puede ser escalada añadiendo más servidores. Para obtener la calidad de servicio necesaria en términos de tiempos de respuesta máximos, las capas más altas deberán procesar tantas peticiones como sea posible sin propagarlas al resto de las capas. Los principales retos son mantener las propiedades transaccionales en todas las capas y conseguir que el procesamiento de transacciones (control de concurrencia y persistencia) sean escalables.

Las instancias del servidor de aplicaciones constituyen la primera capa (*Application Server*). Actualmente, el sistema emplea Java EE, pero cualquier otra tecnología de servidor de aplicaciones podría ser integrada, como por ejemplo .NET. En nuestro prototipo actual usamos el servidor de

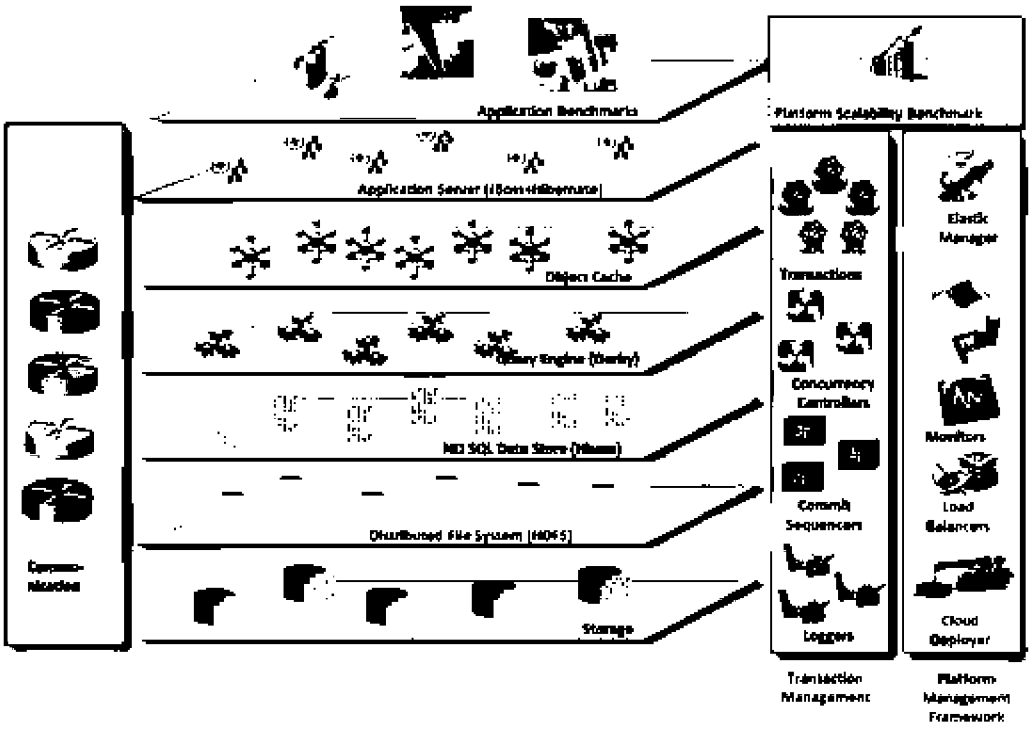

Figura 1. Arquitectura CumuloNimbo

aplicaciones JBoss[2] junto con Hibernate[3] para hacer la transformación objetos a relacional y a la inversa. El procesamiento de transacciones local de JBoss ha sido inactivado y sustituido por el gestor de transacciones de CumuloNimbo. El número de servidores de aplicaciones dependerá del consumo de CPU de la aplicación y de la cantidad de peticiones concurrentes que un único servidor de aplicaciones puede manejar.

La segunda capa está constituida por una caché distribuida de objetos (*Object Cache*). La caché es compartida por todas las instancias del servidor de aplicaciones. La caché puede alojarse en tantos nodos como sean necesarios para mantener los objetos en memoria principal, incluso la base de datos completa, si fuera necesario. Cuando se solicite el acceso a un objeto por la aplicación, Hibernate buscará primero el objeto en la caché distribuida, y sólo si no está disponible en la misma, lo solicitará a la capa de base de datos. La caché también soporta el almacenamiento de los resultados de consultas.

La tercera capa es la capa del motor de consultas SQL (*Query engine*). Esta capa es accedida si: (a) un objeto es accedido, pero el objeto no está almacenado en la caché de objetos, (b) se envía una consulta SQL que no puede ser contestada por la caché, (c) los cambios en los objetos tienen que ser almacenados cuando compromete una transacción para hacerlos persistentes. El número de motores de consultas dependerá del número de consultas complejas que son ejecutadas concurrentemente y de la cantidad de peticiones concurrentes que un motor de consultas pueda manejar. El motor de consultas empleado es el proporcionado por el sistema de gestión de bases de datos Derby[4]. Éste contiene el planificador de consultas y los componentes de ejecución y optimización de las mismas, pero toda la gestión transaccional ha sido desactivada.

La cuarta capa está constituida por el almacenamiento de datos (*No SQL Data Store*). En lugar de emplear el almacenamiento de una base de datos tradicional, éste ha sido sustituido por

[2] www.jboss.org
[3] www.hibernate.org
[4] http://db.apache.org/derby/

un almacén *no SQL* también llamado almacén clave-valor (*key-value store*). Siendo una tupla de la capa de base de datos un par clave-valor. Actualmente se emplea HBase[5] como almacenamiento de datos no SQL. HBase es una tabla hash distribuida, elástica y replicada de manera automática. Tanto las tablas de datos como los índices de la base de datos son tablas en HBase. HBase se sitúa encima de un sistema de ficheros paralelo-distribuido (*Distributed File System*), Hadoop Distributed File System (HDFS), que constituye la quinta capa y se encarga de proporcionar almacenamiento persistente a los datos.

El uso conjunto de HBase y HDFS[6] permite escalar y replicar el almacenamiento dando lugar a una capa de almacenamiento tolerante a fallos y elástica.

La gestión de transacciones se hace de manera holística y las diferentes capas colaboran para proporcionar propiedades transaccionales [6]. La gestión transaccional depende de un conjunto de componentes que proporcionan distinta funcionalidad: secuenciador de compromisos (*commit sequencers*), servidor de *snapshot* (*snapshot server*), gestores de conflictos (*conflict managers*), y *loggers*. Esta descomposición es crucial para obtener escalabilidad.

Existe una capa adicional en la plataforma encargada de tareas de gestión tales como el despliegue, monitorización, equilibrado de carga dinámico y elasticidad. Cada instancia de cada capa tiene un monitor (*Monitors*) recogiendo datos sobre uso de recursos y mediciones de rendimiento que son notificados a un monitor central que emplea Zookepeer[7], un servidor fiable y distribuido. Este monitor central proporciona estadísticas agregadas al gestor de elasticidad (*Elastic Manager*). El gestor de elasticidad examina los desequilibrios en las instancias de cada capa y reconfigura la capa equilibrando dinámicamente la carga. Si la carga está equilibrada, pero se ha alcanzado el umbral superior de uso de recursos, una nueva instancia se provisiona y es ubicada en esa capa. Si la carga es suficientemente baja para ser satisfecha por un número más pequeño de instancias en una capa, algunas instancias de esa capa transfieren su carga a otras y las instancias que no tienen carga son puestas fuera de servicio.

4. Procesamiento Transaccional Ultra-Escalable

Snapshot isolation [2] es el criterio de corrección empleado para la ejecución concurrente de transacciones en CumuloNimbo. Con *snapshot isolation* (SI) dos transacciones concurrentes tienen conflicto solo si las dos transacciones modifican el mismo dato, es decir, no hay conflictos de lectura-escritura, como ocurre con el criterio de *serialidad*. Esta característica hace que los sistemas de bases de datos que implementan SI escalen más que aquéllos que implementan *serialidad*, ofreciendo garantías muy similares [2].

La escalabilidad de SI frente a serialidad se ha mostrado en el contexto de replicación de bases de datos [8]. En este contexto, los límites de escalabilidad no fueron conflictos de escritura, sino el modelo de replicación, escrituras en todas las réplicas. Por tanto, la carga de escrituras no puede exceder la carga que una réplica puede manejar. CumuloNimbo sólo incluye replicación en la capa de persistencia (HDFS) para tolerar fallos. En el resto de las capas los datos son particionados en distintos nodos de tal forma que cada nodo pueda soportar la carga enviada a los datos que mantiene. Esta división de los datos no es observada por las aplicaciones.

Aunque SI escala mejor que la serialidad, en sistemas que ejecutan varios miles de transacciones por segundo, las tareas de control de concurrencia necesarias para implementar SI pueden convertirse en un cuello de botella. Cuando una transacción ejecuta bajo SI las lecturas se realizan sobre la última versión comprometida en el momento en el que empezó la transacción que lee (*snapshot*). La primera modificación de un dato por parte de la transacción crea una versión privada del dato para esa transacción. Posteriores lecturas por parte de esa transacción del dato modificado se harán sobre esta versión. Si la transacción compromete, sus versiones privadas de los datos modificados se harán visibles. Otras transacciones que empiecen después del compromiso de ésta podrán leerlas. Para implementar la semántica SI, las versiones comprometidas de

[5] hbase.apache.org
[6] hadoop.apache.org/hdfs
[7] http://zookeeper.apache.org/

los datos reciben una marca de tiempo creciente (marca de tiempo de compromiso). Cuando una transacción empieza recibe una marca de tiempo (marca de tiempo de inicio) igual a la marca de tiempo de la última transacción comprometida. Esta transacción leerá la versión de los datos con la mayor marca de tiempo menor o igual que su marca de tiempo de inicio. Los conflictos entre transacciones concurrentes que modifican el mismo dato se pueden manejar de manera optimista (al final de la transacción) o pesimista (empleando cerrojos). Este último método es el que implementan las bases de datos. Tanto la generación de marcas de tiempo de inicio y compromiso como la detección de conflictos se puede convertir en un cuello de botella.

Existen tres componentes principales en el gestor de transacciones de CumuloNimbo para implementar SI: *el gestor de conflictos, el secuenciador de compromisos* y *el servidor de marcas de tiempo.*

El gestor de transacciones de CumuloNimbo proporciona un *gestor distribuido de conflictos.* Cada gestor se ocupa de un conjunto de tuplas. El gestor de conflictos es accedido cada vez que se va a modificar una tupla. Éste comprueba si la tupla ha sido modificada por una transacción concurrente y si es así, aborta la transacción. Si no, anota que la tupla ha sido modificada por la transacción. Existe un gestor de conflictos por cada nodo de caché y éste detecta los conflictos de las tuplas que almacena ese nodo de caché. Ambos están en la misma máquina, de esta forma la detección de conflictos es local.

El *secuenciador de compromisos* proporciona marcas de tiempo de compromiso a las transacciones de actualización cuando comprometen. El *servidor de marcas de tiempo* suministra marcas de tiempo de inicio de transacción.

Hay un gestor de transacciones en cada servidor de aplicaciones. Éste recibe las peticiones de inicio y compromiso de las transacciones. Además, intercepta las operaciones de lectura y escritura. Cuando una transacción empieza, el gestor de transacciones proporciona a la transacción la marca de tiempo de inicio. Esta marca de tiempo es la última marca de tiempo que ha recibido del servidor marcas de tiempo. Las lecturas de esa transacción se harán empleando esa marca de tiempo (marca de tiempo de inicio). Cuando una transacción compromete, se comprueba si ha realizado alguna escritura. Si no es así, se trata de una transacción de lectura que comprometerá localmente. Si la transacción ha modificado datos, el gestor de transacciones local asigna a la transacción y a las tuplas modificadas (writeset) una marca de tiempo de compromiso de las enviada por el secuenciador de compromisos, propaga el writeset tanto al *logger*, encargado de hacer duraderos los cambios, como a la capa de base de datos, la cual lo reenvía a HBase. Cuando el logger notifica que el writeset es duradero, el gestor de transacciones local confirma el compromiso al cliente. Cuando la capa de base de datos confirma que el writeset se ha escrito en HBase, el gestor de transacciones local comunica al servidor de marcas de tiempo que los datos asociados a marca de tiempo de compromiso son visibles y duraderos.

El servidor de marcas de tiempo realiza un seguimiento de qué marcas de tiempo de compromiso han sido usadas y cuáles han sido descartadas. Una marca de tiempo de compromiso se considera usada, cuando el writeset es duradero y visible. El servidor de marcas de tiempo también determina cuál es la marca de tiempo de compromiso más reciente, es decir, aquélla tal que todas las marcas temporales previas hayan sido usadas y/o descartadas, e informa periódicamente al gestor de transacciones local sobre esta marca de tiempo que será empleada por los gestores de transacciones locales como marca de tiempo de inicio. Como las tuplas que tienen esa marca de tiempo (o anteriores) se encuentran en la caché y/o en HBase, las nuevas transacciones leerán los datos más recientes comprometidos (snapshot).

El secuenciador de compromisos es responsable de la asignación de marcas de tiempo de compromiso. Éste envía lotes de marcas de tiempo de compromiso a cada uno de los gestores de transacciones locales en intervalos regulares de tiempo (por ejemplo, cada 10 milisegundos). Para determinar el tamaño adecuado del lote, cada gestor de transacciones informa al servidor de compromiso del número de transacciones de modificación que ha comprometido en el periodo anterior. El tamaño del nuevo lote será función de este valor.

Cuando un gestor de transacciones local recibe un nuevo lote de marcas de tiempo de compromiso, descarta todas las marcas de tiempo no usadas y notifica al servidor marcas de tiempo sobre las mismas, pasando a usar las marcas de tiempo de compromiso del nuevo lote. De esta forma, el

tiempo para realizar el compromiso mínimo. La marca de tiempo de compromiso está en el gestor local de transacciones. La detección de conflictos se realizó cuando la operación de escritura tuvo lugar. El único tiempo de espera ocurre por la escritura del writeset por parte del *logger*. Para evitar un que el *logger* se convierta en un cuello de botella, existen varias instancias del *logger*. Cada una de ellas se encargará de un conjunto de writesets. Las instancias del *logger* se pueden crear dinámicamente, bajo demanda.

5. Resultados preliminares

Se ha construido un prototipo inicial e integrado con JBoss, Hibernate, Derby, HBase y HDFS. El prototipo ha sido evaluado en un cluster de 120 núcleos. El despliegue consta con 5 nodos dual-core para almacenamiento en HBase (*region servers*) y *data nodes* HDFS (ambas instancias en un mismo nodo físico); 1 nodo dual-core para el *Name Node* de HDFS y el aprovisionador de la caché; 1 nodo dual-core para *Zookeeper* y el *Secondary Name Node* de HDFS; 1 nodo dual-core al aprovisionador de servidores de aplicación y al *Master Server* de HBase, 1 nodo dual-core al secuenciador de compromiso; 1 nodo dual-core al servidor marcas de tiempo, 5 nodos dual-core para las instancias de la caché y 5 nodos dual-core para las instancias del *logger*. En un quad-core hay una instancia de JBoss, una instancia de Hibernate y una instancia de Derby. Variamos el número de nodos quad-core dedicados a ellos entre 1, 5 y 20. Con esta configuración se ejecuta el *benchmark* SPECjEnterprise2010[8].

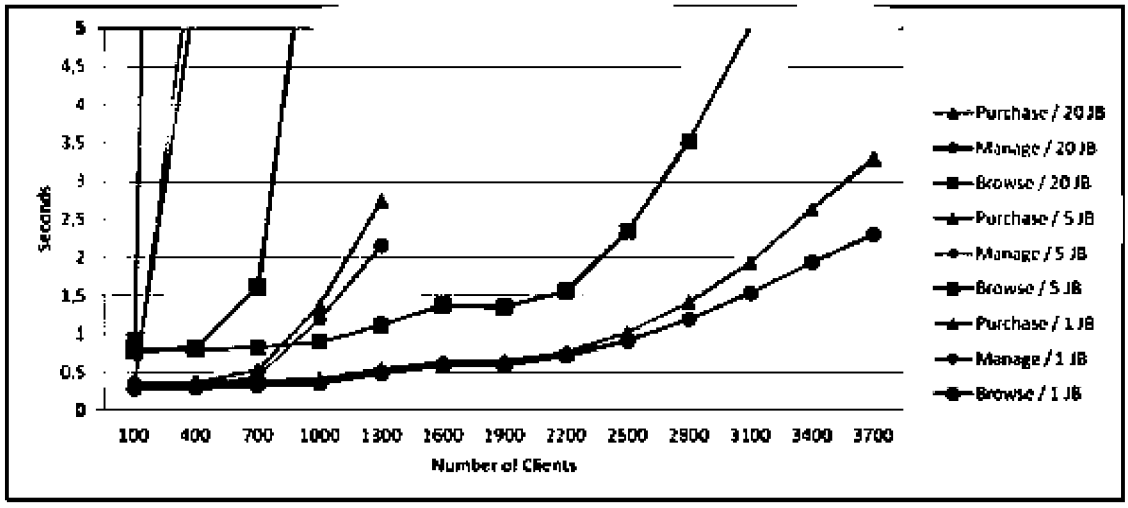

Figura 2. Evolución del tiempo de respuesta con 1 a 20 instancias

La figura 2 muestra los resultados de la evaluación. El umbral del *benchmark* para el tiempo de respuesta es de 2 segundos. Como se puede ver, para las cargas de trabajo con transacciones de escritura (*Manage* y *Purchase*) y con un único nodo JBoss+Hibernate+Derby (Purchase/1 JB y Manage/1 JB) el sistema es capaz de manejar sólo 100 clientes, mientras que con 5 y 20 nodos (5 JB y 20 JB), puede manejar 1000 y entre 3100-3400 clientes. Para la carga de trabajo tipo búsqueda (*Browse*), un nodo JBoss+Hibernate+Derby puede manejar 100 clientes (Browse/1 JB), y con 5 y 20 nodos puede manejar 700 y 2200 clientes (Browse/5 JB y (Browse/20 JB)).

[8] http://www.spec.org/jEnterprise2010/

6. Conclusiones

Este artículo presenta una nueva plataforma transaccional para la *nube*, CumuloNimbo. Las principales características de CumuloNimbo son: trasacciones ACID sin necesidad de particionar los datos (*sharding*), coherencia en todas las capas, transparencia completa, sintáctica y semánticamente. Los resultados preliminares muestran que el sistema escala. El principal problema de rendimiento que hemos encontrado está relacionado con la falta de concurrencia de HBase y el equilibrado de carga que realiza.

Referencias

1. J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.
2. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD*, 2005.
3. P. Bernstein, C. W. Reid, and X. Y. M. Wu. Optimistic concurrency control by melding trees. In *Int. Conf. on Very Large Data Bases.*, 2011.
4. M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *SIGMOD Conference*, pages 251–264, 2008.
5. S. Das, D. Agrawal, and A. E. Abbadi. Elastras: An elastic transactional data store in the cloud. *CoRR*, abs/1008.3751, 2010.
6. R. Jiménez-Peris and M. Patiño-Martínez. *System and Method for Highly Scalable Decentralized and Low Contention Transactional Processing.*
7. R. Jiménez-Peris, M. Patiño-Martínez, I. Brondino, B. Kemme, R. Oliveira, J. Pereira, R. Vilaa, F. Cruz, and Y. Ahmad. CumuloNimbo: Parallel-distributed transactional processing. In *Cloud Futures*, 2012.
8. B. Kemme, R. Jimenez-Peris, and M. Patiño-Martinez. *Database Replication.* Morgan & Claypool Publishers, 2010.
9. J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133, 2011.
10. Microsoft.com. Microsoft SQL Azure Database. Submitted for publication, 2010. http://www.microsoft.com/azure/data.mspx.
11. D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
12. Z. Wei, G. Pierre, and C.-H. Chi. CloudTPS: Scalable transactions for Web applications in the cloud. *IEEE Transactions on Services Computing*, 2011.