The End of an Architectural Era

It’s time for a Complete Rewrite

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, Pat Helland

Javier D. Fernández

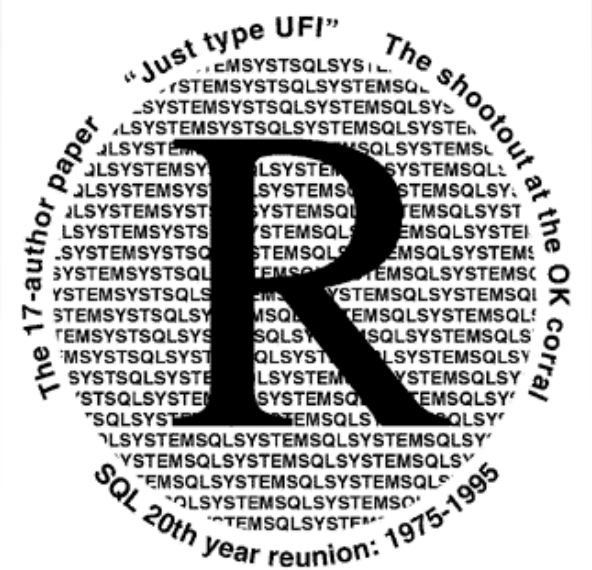
Idea principal

CAMBIO

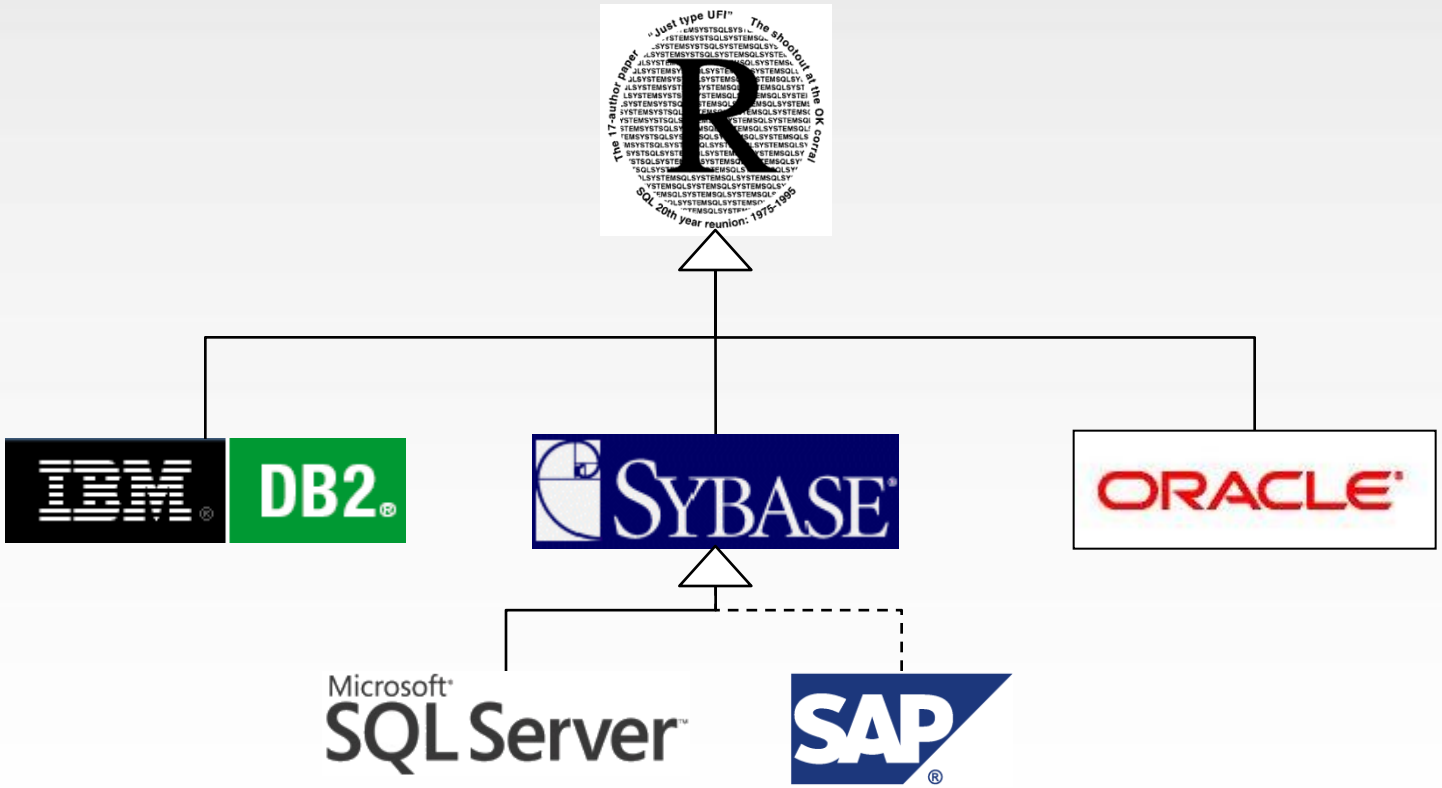
"Los vendedores de DBMS (y la comunidad investigadora) debe empezar desde cero el diseño de sistemas para los requisitos del mañana. NO CONTINUAR ARRASTRANDO LÍNEAS DE CÓDIGO Y ARQUITECTURAS DEL PASADO"

Origenes

- **1970: System R**
  - System R is a database system built as a research project at IBM San Jose Research (now IBM Almaden Research Center) in the 1970's. System R introduced the SQL language and also demonstrated that a relational system could provide good transaction processing performance.



Origenes



Los problemas

- 25 años de diseño →cambio de Hardware
  - Procesadores más potentes
  - Memoria casi infinita
  - Menor crecimiento ancho de banda entre disco-memoria principal
- Centrado en “business data processing”
  - Ahora, nuevos requisitos: data warehouses, manejo de texto, procesamiento streaming.
- Diseñado para “dumb terminal”
  - Ahora, potentes computadores, WWW

Los problemas

- Misma arquitectura que System R
  - Orientación a almacenamiento en disco y estructuras indexadas
  - Multithreading para ocultar latencia
  - Mecanismos de control de la concurrencia “locking-based”
  - Recuperación basada en Log.

- Rediseños:
  - Compresión, arquitecturas de discos compartidos, índices basados en bitmaps, soporte para operadores y tipos de datos definidos por el usuario, etc.

Es hora de una rescritura completa

Demostración

- Trabajo Previo [SBC+07]:
  - RDBMSs superados (benckmarking):
    - Texto (Google, Yahoo)
    - Data Warehouses (column stores:Vertica, Monet)
    - Stream processing (StreamBase, Coral8)
    - Científico, IA (MATLAB, ASAP)
  - ➡ diseño original apto para “business data processing”
  - ➡ no apto para otros mercados
- Este trabajo:
  - H-STORE para OLTP
    - 2 ordenes de magnitud en TPC-C

Es hora de una rescritura completa

5 Consideraciones de Diseño

- 1. Memoria Principal
  - 1 Mb (1970) → 100 Gb (Hoy) → 1 Tb (futuro cercano)
  - Mayoría de BBDD OLTP < 1 Tbyte (<<)
  - TPC-C requiere 100Mbytes por warehouse
    - 1000 warehouse = 100Gb
  - ➡ OLTP debe considerar el mercado de la memoria principal.
    - Diseño orientado a disco originalmente, sobrepasado por la ley de Moore.
    - TimesTen, SolidDB, orientadas a memoria pero heredan System R
      - Recuperación log basado en disco
      - Dynamic locking

5 Consideraciones de Diseño

- 2. *Multi-threading* y Control de recursos
  - Transacciones ligeras
    - TPC-C, 400 registros, 1 ms
  - Mayoría sin intervención usuario *(e.g. "buy it"),* sin operaciones en disco→ tiempo reducido
  - ⇨ No tiene sentido multi-threading, todos los comandos SQL en una transacción.
    - No sobrecarga de aislar ejecución concurrente
    - No estructuras adicionales, ni código elaborado
  - ¿Comandos de larga ejecución?
    - Generalmente divididos en varias transacciones
    - Redirigidos a *data warehouse* que lo optimizan

5 Consideraciones de Diseño

- 3. *Grid Computing* y actualización masiva (*fork-lift upgrade)*
  - *Shared-memory (1970) → shared-nothing (grid) (hoy, futuro cercano)*
    - *Particionamiento horizontal en los nodos del grid*
  - *fork-lift upgrade → expansion incremental*
    - *N nodes + K nodes = N+K nodes*
      - *Sin parada del sistema (nuevo requisito)*

5 Consideraciones de Diseño

- 4. Alta disponibilidad (*High Availability)*
  - *Single machine (1970)*
    - *Log tapes + compra de nuevo equipo ante fallos*
  - Hot standby (Ahora)
    - Mitad de recursos disponibles
  - ⇨ Alta disponibilidad *(HA),* requisito esencial en OLTP
    - Múltiples replicas consistentes.
    - *Shared-nothing*
    - *Peer-to-peer*

  - *HA simplifica log, siendo aún más eficiente.*
    - *En memoria*
    - *Nunca redo, sino volcado desde otro equipo*

5 Consideraciones de Diseño

- *5. no knobs* (sin botones)
  - *Máquinas caras, personal barato (1970) → contrario (ahora)*
    - *Cuidado, mantenimiento,* tuning, optimización manual
    - *Evolución hacia el automatismo*
      - *Manuales? Legado?*
  - ⇨ *"self-everything"*

Asumciones

- *Grid con almacenamiento en memoria principal, HA, sin paradas de usuario, transacciones<1ms:*
  - *Log → HA*
  - *Sobrecarga interfaz JDBC/ODBC → rediseñar con lógica de aplicación en ejecución.*
  - *Un único hilo*
  - *No undo log (*)*
  - *Eliminar el coste de control de concurrencia dinámico (*)*
  - *Evitar commit en dos fases para ahorrar latencia en la red. (*)*

  - *(*) ¿Cómo? → Características del esquema y las transacciones*

Características del esquema y las transacciones

- *H-Store requiere que la carga (*workload) *se especifique con antelación*
  - Clases transacionales *(*transaction classes)
    - Separación de transacciones individuales que afectan a los mismas constantes en ejecución.
    - No contienen parada del usuario
    - El esquema lógico afectado es conocido previamente.
- Joins 1-n *(tree schemas)*
  - E.g. clientes, ordenes, items, historial…
  - Partición horizontal: todo lo de una *primary key* en un nodo (*site*)
- *CTA (constrained tree application*)
  - Todos los comandos de todas las transacciones de una clase transaccional van a un nodo
    - No comunicación entre sitios del grid (salvo sincronización)
  - *Ejecución eficiente*

Características del esquema y las transacciones

- *OLTP* suelen diseñarse siendo CTA's o se pueden descomponer en ellas[Hel07]
  - *Read-only tables (replicadas)*
  - *One-shot* :varios CTAs paralelos sin necesidad de comunicación intermedia.
    - Se puede conseguir con Particionamiento vertical en columnas entre sitios (replicando columnas no actualizables)
- Transacciones *two-phase: read-only (*abortable) + (Q,U) necesiten integridad.
  - Muchas son *two-phase* (TPC-C)
  - Permite eliminar el log undo
  - *Strongly two-phase*: operaciones de la primera fase producen mismo resultado de abortar o continuar en todos los nodos.
- Transacciones *sterile:* Clase transaccional *conmuta* con todas las transacciones
  - dos transacciones conmutan cuando pueden permutar las operaciones en un nodo sin cambiar su resultado final.

H-Store. Arquitectura

- Grid de computadores
- *K-safety* configurable
- En cada nodo, filas continuas en memoria principal
  - B-tree con tamaño de bloque = ancho cache L2
- *Single threaded*
- *Site* físico se divide en *sites* lógicos
  - 1 Core + porción de mem. ppal.
- Procedimientos en C++
  - E.g. *Execute transaction(parameter_list)*
  - Llamadas a procedimientos (y devolución en variables), no JDBC.
- No log *redo*, sí *undo* cuando es necesario y en mem. ppal.

H-Store. Ejecución de consultas

- Optimizador de consultas basado en coste, más sencillo
  - Joins con menos tablas en OLTP
    - de producirse, son secuencias de 1-n joins
  - GROUP BY y agregación ocurren infrecuentemente

- ⇨ Planes de ejecución puede ser:
  - *Single-sited*
  - *One-shot:* varios single-sited
  - *General:* comunicación entre *sites*
    - *Medir la profundidad (*depth) de la transacción, en número de mensajes entre sitios.

H-Store. Diseño de la BBDD

- Automático (*no-knobs)*
  - Tablas del usuario → particionamiento horizontal, replicas, índices
    - Disponibilidad (HA)
    - Favorecer transacciones single-sited
- Estrategia de búsqueda de patrones (similar C-store)
  - Estrella (*star)*
  - Copos de nieve (*snowflakes*)
  - Casi copos de nieve (*near snowflakes)*
- + Búsqueda de CTA, read-only, particionamiento vertical

H-Store. Gestión, Réplica y Recuperación de Transacciones

- Consistencia:
  - Consultas se dirigen a cualquier réplica (de 1 tabla)
  - Actualizaciones a todas las réplicas
  - Orden total (*site_id,unique_local_timestamp)*
- Control de concurrencia y *commit*:
  - **Single-sited/one shot**
    - Cada transacción individual se dirige a replica adecuada
    - Pequeño tiempo de espera para ejecución ordenada por timestamp
    - En distintas replicas, todas terminan o todas abortan
    - No inconsistencia, no hay necesidad de log redo, no control de concurrencia, no commit distribuido
  - **Two phase**
    - No log *undo*. Nada más necesario que lo anterior.

H-Store. Gestión, Réplica y Recuperación de Transacciones

- ***Sterile***
  - Pueden ejecutarse sin control de concurrencia.
  - Sitios necesitan responder con "*abort"* o "*continue"*, al final de la primera fase
    - Supervisor recolecta comunicaciones e informa a trabajadores (*workers*).
    - Se puede evitar si es *strongly two phase* (todos el mismo resultado de continuar o abortar)
- **Otros casos** (*non-sterile, non-single-sited,non-one-shot)*.
  - Evitar dynamic locking (1980's)
    - Transacciones tienen muy corta vida (optimista vs. Pesimista)
    - Ejecuciones en un único hilo
    - Se conocen las clases de transacciones de antemano
    - Pocas colisiones y *deadlocks* con sistema bien diseñado

Política frente a Otros casos (*non-sterile, non-single-sited,non-one-shot)*. 3 posibilidades (cambia dinámicamente si hay muchos abort)

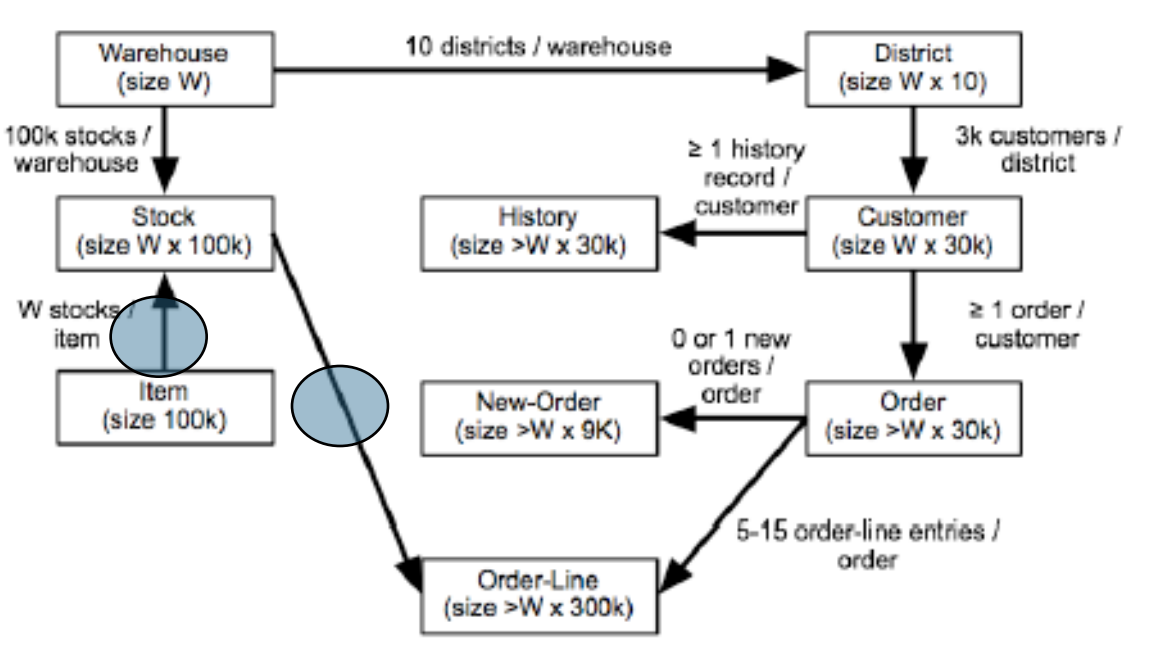
- Básica:
  - Clase transaccional tiene una colecciones de otras clases con las que puede tener conflicto
  - Coordinador envía subplanes a sitios
    - Esperan si hay transacciones conflictivas con menor timestamps
      - Ejecuta el subplan si no hay clases clases conflictivas sin hacer commit con menor timestamp
      - Aborta en otro caso
    - Si recibe OK de todos los sitios, contínua (o commit). Sino aborta.
- Intermedia
  - Ajusta el tiempo de espera (MaxDepth*Average_message_delay)

H-Store. Gestión, Réplica y Recuperación de Transacciones

- Política avanzada:
  - Almacena conjuntos de escritura y lectura de cada transacción en cada sitio.
    - Reglas de control de la concurrencia optimista
  - Sobrecarga

Comparación de Rendimiento (*performance)*

- TPC-C



Comparación de Rendimiento (*performance)*

- Replica y particionamiento
  - Item es *read-only* → replica en cada site
  - Order-line se particiona por Order en cada site.

Comparación de Rendimiento (*performance)*

- *Two-phase*
- Strongly-two phase
- Sterile

- ⇨ No control

intervención humana : conocimiento para escribir adecuadamente las clases transaccionales

`new_order`	Place an order for a customer. 90% of all orders can be supplied in full by stocks from the customer's "home" warehouse; 10% need to access stock belonging to a remote warehouse. Read/write transaction. No minimum percentage of mix required, but about 50% of transactions are new_order transactions.
`payment`	Updates the customer's balance and warehouse/district sales fields. 85% of updates go to customer's home warehouse; 15% to a remote warehouse. Read/write transaction. Must be at least 43% of transaction mix.
`order_ status`	Queries the status of a customer's last order. Read only. Must be at least 4% of transaction mix.
`delivery`	Select a warehouse, and for each of 10 districts "deliver" an order, which means removing a record from the new-order table and updating the customer's account balance. Each delivery can be a separate transaction; Must be at least 4% of transaction mix.
`stock_ level`	Finds items with a stock level below a threshold; read only, must read committed data but does not need serializability. Must be at least 4% of transaction mix.

Implementación

- Comparación con un RDBMS comercial (“muy popular”)??
- TCP/IP
- C++ vs lenguaje de procedimiento propietario
- Dual-core 2.8 Ghz, 4Gb memoria, discos 250Gb SATA.
- Ambos con particionamiento horizontal

Resultados

- H-Store: 70.416 TPC-C transacciones por segundo vs. 850 transacciones por segundo del sistema comercial (tuning profesional)
  - 82 veces más rápido
  - Sobrecarga de logging en el sistema comercial (2/3 del tiempo)
    - 2500 transacciones por segundo sin log.
  - En un futuro también contar la sobrecarga de control concurrencia.
- H-Store 35000 transacciones por núcleo vs. 425 sistema comercial

- ⇨ Mejora de casi 2 órdenes de magnitud

Cometarios: El Modelo Relacional no es necesariamente la Respuesta

- “vaca sagrada” (debate 1974, Codasyl)
- Apropiado considerar otros modelos de datos para nuevos mercados, incluyendo *data-warehouses,* búsqueda orientada a la Web, análisis en tiempo real, datos semiestructurados. Observaciones
  - Data warehouse: casi 100 % son esquemas estrella o bola de nieve.
    - Tabla central con 1-n joins, a su vez estos 1-n…
    - E-R pueden tomar ventaja en representación y consulta.
  - Stream processing:
    - Requiere
      - Velocidad
      - Correlación de streams con datos almacenados
    - StreamQL (mezcla tablas y streams en cláusula FROM)

Cometarios: El Modelo Relacional no es necesariamente la Respuesta

- Text processing (nunca ha usado el modelo relacional)
- DBMS científico no considera tablas sino arrays
- Necesidad de buenos modelos para datos semiestructurados
  - Sobrecomplejidad de XMLSchema
  - RDF?
    - BBDD orientadas a columnas (OLAP)

- ➡ Modelo relacional se desarrolló para “one size fits all”. Sistemas específicos pueden repensar qué modelo de datos funciona mejor para sus necesidades particulares

Cometarios: SQL no es la Respuesta

- SQL no es mínimo para OLTP
  - No consultas ad-hoc
  - Procedimientos están omnipresentes
- Data warehouse:
  - Consultas ad-hoc
  - No procedimientos almacenados
- Se puede hacer un mejor trabajo
  - Interfaces más eficientes que JDBC/ODBC
  - Lenguajes más reducidos (python, perl…)
    - Open source
    - Más “modificables”
    - →ruby on rails como ejemplo

Conclusiones y Trabajo Futuro

- Últimos 25 años, cambio dramático en:
  - Mercados DBMS
  - Nuevos requisitos (shared nothing, HA)
  - Tecnología (memoria, replicación, etc.)
- Resultado:
  - Invalidez de “one size fits all”
  - Invalidez de control de concurrencia relacional para todos los casos
  - Repensar modelos de datos y lenguajes de consulta para motores específicos.
- H-Store demuestra posibilidad de mejora.

Conclusiones y Trabajo Futuro

- Futuro:
  - Identificación automática de single-sited, two phase. Etc.
  - Optimización multicore
  - Estudio más fino del rendimiento
  - Estudio más fino de sobrecargas
  - Mejora de estructuras de datos en memoria
  - Integración con herramientas warehouse
    - Actualización diferida

Debate

- ¿Comparación demasiado obvia?
- ¿Automatismo es posible?
- ¿Realmente es necesario reescribir?
  - Título engañoso: don’t rewrite
    - Complete rewrite de nuevo “one size fits all”

The end…