NoSQL, Object Caching & IMDSs

Alternatives for Highly Scalable Data Management

McObject LLC
33309 1st Way South
Suite A-208
Federal Way, WA 98003

Phone: 425-888-8505
E-mail: info@mcobject.com
www.mcobject.com

The Data Availability Bottleneck

Business and social transactions are rapidly moving to the Internet. But as Web-based services and data center applications scale, their quality of experience can vary widely between visits. Developers of financial, commercial, e-commerce, social network and other high-volume applications need a data management solution that can keep up with their growth while providing high availability, predictability, reliability and safeguards on data integrity.

This white paper examines various solutions for real-time, high-volume data management. It starts with the "traditional" back-end database management system (DBMS). While the DBMS (typically paired with an SQL application programming interface) has proven itself in a vast range of applications over the past three decades, it has arguably reached its limits in today's highly scalable, distributed applications. The paper looks at these limits, then investigates several types of solution that are often put forth as adjuncts or successors to the RDBMS in this new world, including object caching software, NoSQL solutions, and in-memory database systems (IMDSs).

Solution: The "Traditional" Relational Database Management System (RDBMS)

The relational DBMS – typified by commercial products like Oracle, and open source packages including MySQL and PostgreSQL – already serves well as the back-end repository for millions of enterprise applications and Web sites. The technology is mature and offers characteristics that many view as "must haves" for production applications, including:

- **Guaranteed data integrity** through transactions that comply with the ACID (Atomic, Consistent, Isolated and Durable) rules, which ensure that related changes to the database complete or fail as a single unit

- **Recoverability** provided by features such as transaction logging, a journaling process that enables changes to the database to be recovered following a hardware or software failure

- **Storage efficiency** via an architecture that supports normalized database designs

- **Development and run-time efficiency** from tools including multiple database indexes (for fast lookup from database tables), application programming interfaces (both high-level SQL and faster native), and a formal schema (data definition) language

- **High Availability and load-balancing** through technology that replicates data across multiple nodes and enables multiple servers to share the data processing load (although this can often require integration of separate clustering software, which adds expense and complexity)

DBMS Limitations

Just as RDBMSs' strengths are well known, decades of use have made clear their limitations, as well.

- **Disk and File Access**

A perennial complaint against the RDBMS is the long wait for an operation to complete. Much of this latency is hard-wired into traditional databases due to their reliance on file- and disk-access for record storage. DBMSs typically provide caching, to keep the most frequently-used records in memory, for faster access. However, caching only eliminates the performance overhead of information retrieval , or "database reads," and only when the requested information happens to be in the cache. Any database write – that is, an update to a record or creation of a new record – must ultimately be flushed out of the cache, to disk. To enforce the ACID properties, this must be a synchronous (blocking) operation. So, while the I/O of database updates can be delayed through caching, it can't be eliminated.

Caching is also inflexible. While the user has some control over cache size, the data to be stored there is chosen automatically, usually by some variant of most-frequently-used or least-frequently-used algorithms. The user cannot designate certain records as important enough to always be cached. It is typically impossible to cache the entire database. In addition, managing the cache imposes substantial overhead on the DBMS.

- **Data Transfer**

In addition, data that is managed using a traditional DBMS architecture must be transferred to numerous locations inside and outside of the database as it is used. Figure 1 shows the handoffs

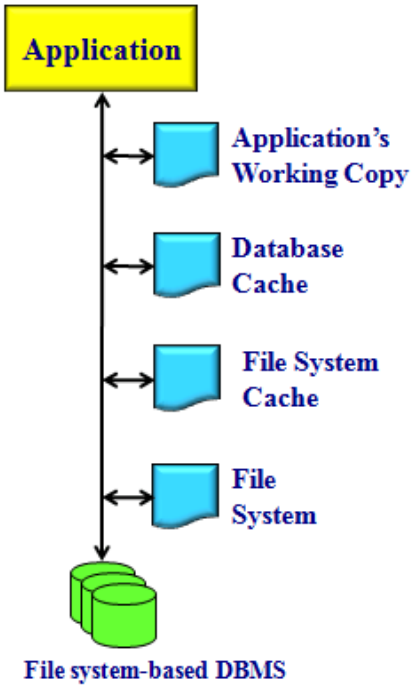

Figure 1. Data transfer in a database management system (DBMS)

required for an application to read a piece of data, modify it and write that record back to the database

These steps require CPU cycles (which equates to time) and cannot be avoided in a traditional database (even when physical I/O is avoided, such as when the DBMS is deployed on a RAM-disk). Still more copies and transfers are required if transaction logging is active.

- **Scalability: Large Data Volumes**

Traditional databases' client/server architecture is fine for online transaction processing (OLTP). It can support hundreds or even thousands of concurrent users/connections performing business transactions or updating customer records. But this approach breaks down when dealing with the "big data" of, for example, analytical processing or social graphs hosted in the server farm environment typical of high-volume Web applications (See Figure 2). In this situation, the DBMS server can quickly become a performance bottleneck that slows processing elsewhere in the system.

Disk-based relational DBMSs' approach to storage almost guarantees a slowdown. As the size of the database grows, the b-tree indexes get deeper (the b-tree can be visualized as an upside-down tree, with a root-node at the top that has left and right sub-nodes. Each of those, in turn, have their own sub-nodes, and so on). Each level in that tree equates to a logical disk I/O, so the deeper the tree is, the greater the number of I/Os required to find a value there. Translation: the larger the database, the more I/O and CPU cycles that are required to read from and/or update it.

- **Scalability: Many Concurrent Users**

Another problem is that with a high user load the sheer number of requests can swamp the database. Pokerstars.com, for example, can have 200,000+ players online during peak hours. Concurrent requests from users of popular social networking sites would crush any database system. Requests can overwhelm the database cache, forcing out the data from a previous user's request even if that user is still working with the data.

Clustering solutions built on top of traditional databases can address the problem of high user loads by dividing those users across N servers in the cluster, so that any single database instance only has to handle concurrent-users/N connections. Some DBMSs can also be partitioned, to keep index trees shallow within the different partitioned areas.

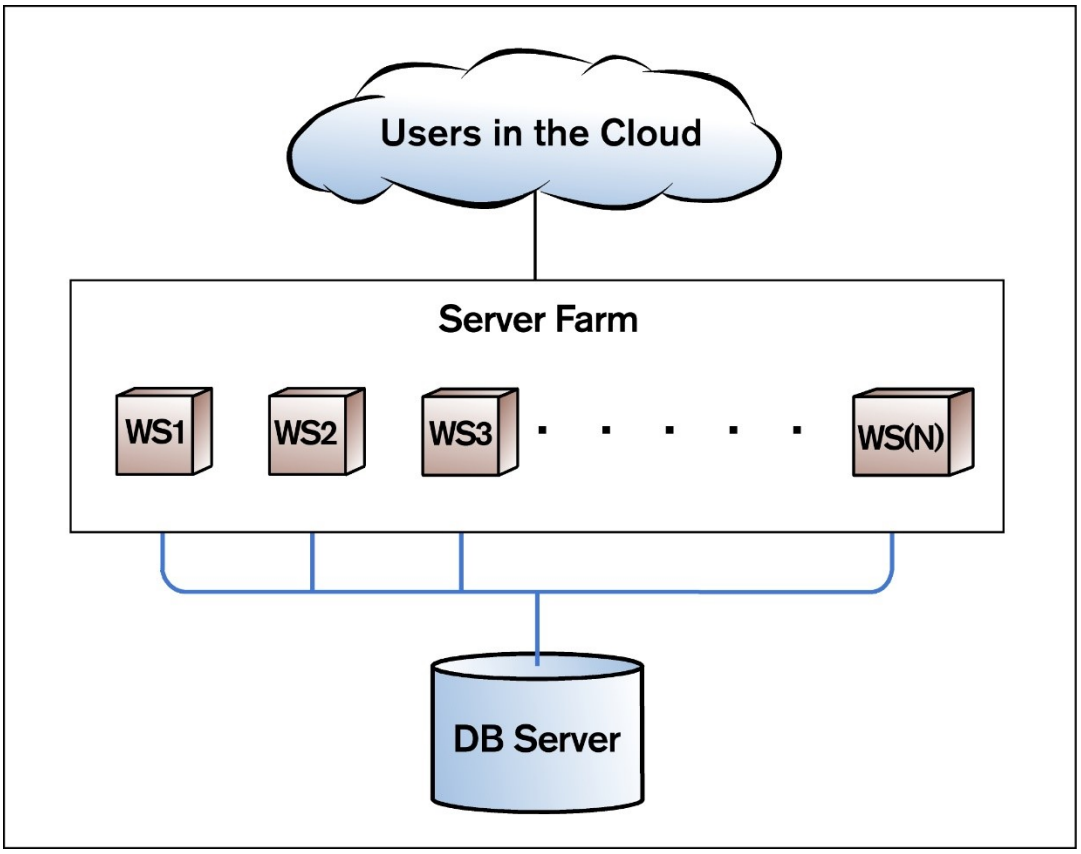

Figure 2. Typical sever farm environment of high volume Web-based applications, where multiple Web servers access a DBMS backend. Traditional databases can overcome the limited scalability implied in this diagram via clustering or a storage area network (SAN). However, even these solutions are still burdened with DBMSs' disk and file I/O, data transfer and inflexible caching inherent in the technology.

- **Incompatible Data Models**

Traditional relational databases are built on the logical concept of tables consisting of rows and columns. Data that does not naturally fit into such structures must be "taken apart and re-assembled" through the process of object-relational mapping (ORM) in order to fit into a tabular data layout. This eats up CPU cycles and memory. The overhead of dealing with non-relational data may be unnoticeable in a system that relies on a smaller database. But when an RDBMS is used as the back-end for large-scale applications, this latency can significantly harm performance. For this reason, developers of such applications are increasingly seeking out data management solutions that do not force them to organize data in rows and columns.

Due to the limitations discussed above, it's unsurprising that developers are looking beyond the traditional DBMS to meet the data access demands of real-time, high volume applications. The following sections look at some of the emerging alternatives.

Solution: Distributed Object-Caching Software

Object-caching software accelerates dynamic (database or API-driven) Web sites by holding frequently-used items (objects, strings, etc.) in memory outside the DBMS, and is often spread across multiple nodes or servers. For example, Memcached, one of the most popular object-caching technologies, consists of an in-memory key-value store for small chunks of arbitrary data; it is typically compared to a hash table distributed across multiple machines. Each node functions independently but treats memory available on all nodes, in aggregate, as a single memory pool, and can instantly "see" changes that other nodes have made to the key-value store.

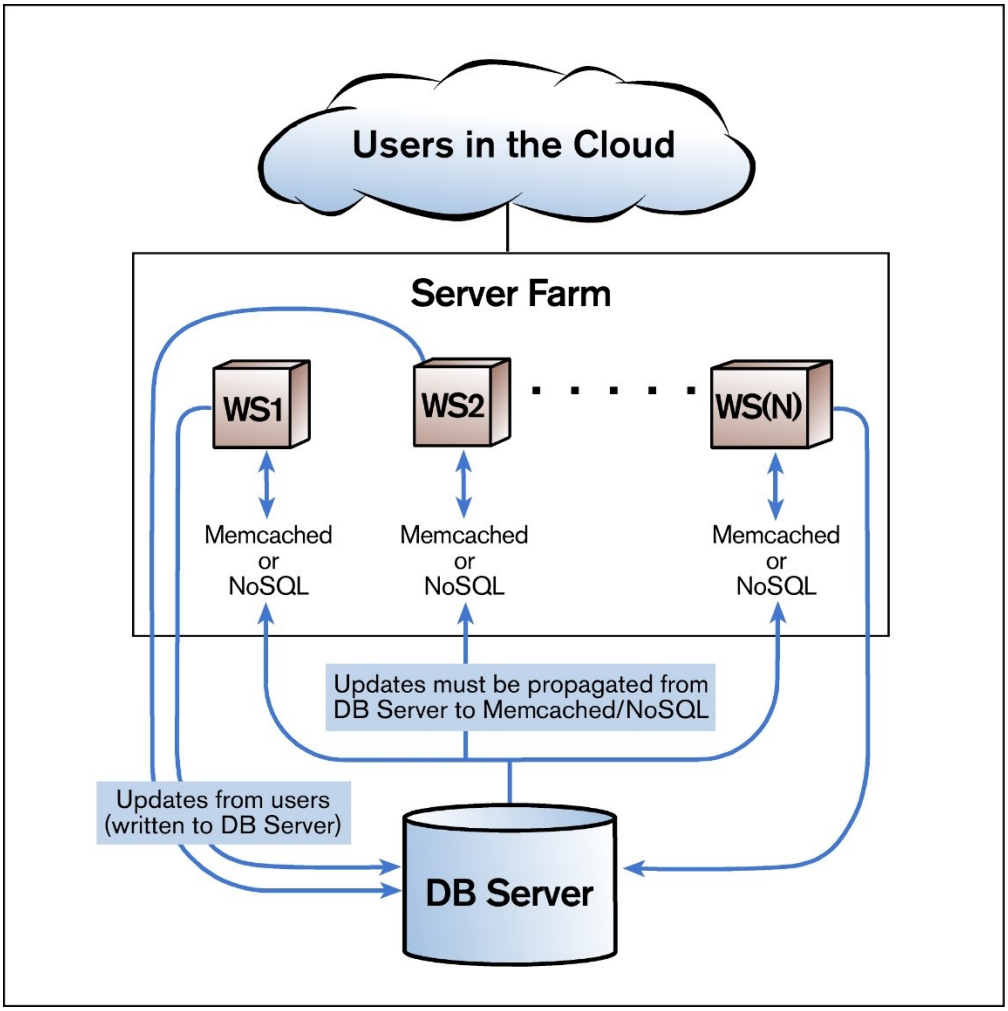

Figure 3. In high-volume systems based on Memcached or other NoSQL object-caching solutions, updates to shared data must still be written to the database server and then propagated to the distributed data management nodes. For write-intensive applications, DBMS limitations (disk and file I/O, inflexible caching, etc.) plus the propagation traffic will become a bottleneck.

Like a database cache, an object cache along the lines of Memcached is finite in capacity, and typically uses a least-recently used algorithm to purge data, making room for content that is deemed to be of higher value for application users. However, advantages of object caching solutions include the ability to manage much greater volumes of data than can be accomplished using database caching; easy

accessibility of data in main memory; ability to scale out over multiple physical nodes; and fault tolerance through redundancy of multiple nodes. Figure 3 shows a typical high volume Web system based on object-caching, with a traditional DBMS as a back end to store persistent data.

Object-Caching Limitations

While solutions such as Memcached are used successfully in many Web-based applications, they have drawbacks that limit their applicability for systems with certain requirements and access patterns. Following are several of the technology's frequently-cited limitations.

Database Writes

Object-caching helps sites with a high database load that contains mostly read threads. It is not designed to address the bottleneck that occurs when users frequently change database content – for example, when on-line purchases cause records to be rapidly and continuously updated. Web sites and applications that are write-intensive will need to take a different approach in addressing the challenge caused by accelerating user demands.

Persistence

What happens when someone "pulls the plug" and a heavily used Web application, including its object-caching layer, goes down? Solutions such as Memcached permit updates to their relatively simple collection of keys and values. And when it goes down, any updates to the information (that is, any data that differs from the records stored in the back-end DBMS) are gone for good. Neither an on-disk copy, nor a log of changes, exists to restore the cache to its pre-failure state. This lack of persistence gives pause to some developers and has driven major social networking Web sites to seek a more persistent caching solution for higher value data.

Storage Efficiency

One downside of object-caching pointed out by its users is a need for large amounts of storage (memory). Because the technology is not designed to perform complex queries or sorts, the system designer must anticipate different "views" of a data set that an end-user might want. These results are then pre-computed and held in memory, for ready access. When the application needs to manage large data sets and provide many possible views, storage requirements can become prohibitive.

Other issues

MemCached does not support transactions. Multiple records cannot be updated as an atomic action. In addition, MemCached does not support a rich querying language. It provides key-value search only.

Solution: NoSQL

"NoSQL" is an umbrella term applied to a variety of recently emerging, non-relational data management technologies – from distributed key-value stores (Memcached is often included in the NoSQL camp) to

document-oriented databases to graph databases – used to manage big data. While the products are diverse, they typically share characteristics, including:

- Emphasis on redundancy and on managing large data stores by "scaling out" to multiple nodes
- Use of commodity hardware (i.e. cheap PCs)
- Relaxation of transactional (ACID) rules supported by conventional relational databases
- Lack of fixed table schema (data definition)
- Avoidance of join operations
- Origins in data management for highly scalable Web. 2.0 sites (FaceBook, LinkedIn)

NoSQL is still far from the mainstream – the various products are emerging, and their merits still being debated. Analysts already credit NoSQL as a potentially cost-effective way to consolidate disparate types of data that may exist within an organization, and avoid the time and expense of mapping the information to an SQL format.

Unlike one-size-fits-all relational DBMSs, NoSQL refers to multiple specialized solutions that address different data management challenges: document-oriented databases for applications managing uniform, document-type information; columnar databases for analytical processing; distributed key-value stores for very high-volume applications with relatively simple data relationships; graph-oriented databases based on nodes and edges to reflect the interactions and connections common on social networks, etc. For this reason, many experts say, NoSQL is best used to supplement, rather than replace, the traditional DBMS.

However, the very specialization of NoSQL products presents a management challenge: will an organization be able to afford the expertise needed to manage two or three NoSQL technologies in addition to its conventional DBMS investment? And if NoSQL *is* relied on heavily, certain NoSQL drawbacks must be weighed:

Lack of "real" database tools. NoSQL is best at providing fast access to large amounts of relatively non-complex data, or data that fits a certain pattern (key-value pairs, documented-oriented, etc.). However, when an application must "flex" to support a new type of data or query, NoSQL can fall short. Lack of specialized database indexes, fixed schema and/or denormalized designs may hinder efforts to support efficient queries. By definition, NoSQL technology largely rejects the SQL database language that is standard for business and other types of processing.

"Eventually Consistent" instead of ACID. One DBMS hallmark is its strict enforcement of data consistency (the "C" in ACID) through transactions. Grouping changes into units that succeed or fail together ensures, for example, that when funds are transferred from one bank account to another, one of the accounts is credited and the other is debited (or the transfer fails).

Many NoSQL solutions replace ACID with the idea of "eventual consistency." For example, a vast data store may become temporarily partitioned into areas that are inconsistent, although the NoSQL solution will eventually eliminate these inconsistencies. (Under the CAP Theorem— formulated by U.C. Berkeley Professor Eric Brewer, and a hot topic in the world of distributed applications—a data management solution can support only two out of the three characteristics of Consistency, Availability and Partition Tolerance. The NoSQL solutions that support eventual consistency have traded away C in favor of A and

P.) While jeopardizing data consistency in a social networking application might be a risk worth taking, it is less likely to be an option in, say, finance or medical records.

Immature technology. NoSQL is still very much a work in progress. To plunge headlong into using one of the NoSQL solutions is to be an early adopter, and face the well-known risks of betting on a technology that ultimately dead-ends, or being saddled with a solution that is "not ready for prime time" (e.g. buggy, incomplete, etc.). It is worth noting that NoSQL solutions are mostly open source. While mature and widely used open source technologies such as Linux have well-established communities willing to help fellow users (through forums, user groups, publications, etc.), community-based support resources for the separate NoSQL products are considerably thinner.

In-Memory Database Systems

In-memory database systems (IMDSs) manage records entirely in main memory – they need never go to disk. In contrast, traditional or on-disk database systems hold some frequently used data in cache for quick access, but write updates through the cache, to persistent media. For this reason, IMDSs are much faster: disk- and file- I/O are eliminated, along with the CPU demands, data transfer and other overhead imposed by caching. IMDSs' streamlined architectures also often minimize memory and CPU demands (a major reason for their initial popularity in resource-constrained embedded systems).

In their other characteristics, IMDSs can look a lot like traditional DBMSs, with transactions that support the ACID properties, formal data definition languages (DDLs), SQL support, and more.

Some IMDSs' roots are in embedded systems, providing real-time data lookup and storage for devices such as telecom switches, set-top boxes and industrial controllers. However, with the addition of features including 64-bit support, multi-version concurrency control (MVCC), and high availability (through database replication), IMDSs have emerged as a contender to meet the data management needs of high volume real-time enterprise applications. In such deployments, the IMDS typically serves as a front-end real-time cache for an enterprise relational database management system, holding frequently-accessed records and thus pre-empting costly (in performance terms) hits to the back-end.

For example, the *eXtreme*DB® IMDS from McObject® has been chosen as an in-memory cache for numerous highly scalable applications, in fields ranging from finance to social networking Web sites to software-as-a-service (SaaS) business systems. To meet these diverse applications' needs, the *eXtreme*DB product family offers specialized editions ranging from 64-bit to Transaction Logging, High Availability and state-of-the-art "shared nothing" clustering (see a more detailed description below). These specialized capabilities are typically used in combination: a fault-tolerant system that manages a large data set would use *both* the 64-bit and High Availability *eXtreme*DB technologies.

Clearly, in some areas – such as fast, in-memory data access and ability to distribute a data store – the capabilities of IMDSs, object-caching solutions and NoSQL overlap. Why would real-time enterprise system designers choose an in-memory database system over the other options? Benefits of the IMDS solution include the following.

- **Performance**. In an IMDS, records are accessed directly from main memory, eliminating disk and file I/O, cache management, data transfer and other performance overhead of disk-based

DBMSs. As explained above, traditional DBMS caching can postpone, but never eliminate, disk writes, and the latency gap between writing to disk vs. writing to main memory storage equates to milliseconds vs. microseconds. Other performance advantages are inherent in IMDSs' database features, and are absent in object-caching and NoSQL solutions (these features are discussed in their own bullet point, below).

- **Persistence**. Databases – even in-memory database systems – provide a higher level of persistence than object-caching solutions. For example, *eXtreme*DB optionally supports transaction logging, a process in which changes to the database are recorded in a log as they occur. In the event of hardware or software failure, the log can be used to return the database to its most up-to-date state.

- **Database features**. Because IMDSs are "real" databases, many offer performance-enhancing features that are missing in object cache and NoSQL solutions. For example, *eXtreme*DB provides a variety of *database index types* that can greatly improve application efficiency, including some indexes for specialized purposes such as supporting mapping and IP routing functions. *eXtreme*DB also includes a variety of *application programming interfaces* (APIs); industry-standard SQL is supported, but for performance, many developers prefer to use the faster native API, or the Java Native Interface (JNI). Other database features in IMDSs are designed to protect data integrity, particularly transactions that support the ACID properties.

- **Storage efficiency**. Users cite significant gains in efficient use of storage (where RAM is "storage space") from using an IMDS rather than an object-caching solution such as memCached. This is because with IMDSs' features for data organization and querying – such as indexes, efficient programming interfaces, and the ability to create optimized data designs using a data definition language – only core data need be stored in the database. In contrast, when using object-caching, the designer must anticipate and compute results that users may want to see, and store these "views" for ready access.

The *eXtreme*DB IMDS: Proven Capabilities for Scalable Real-Time Systems

McObject's *eXtreme*DB in-memory database system emerged to meet reliability and zero-latency responsiveness needs in fields such as telecommunications, aerospace and defense. Based on this core engine, McObject has built a product family that provides unmatched scalability, durability, flexibility and performance for today's high-volume enterprise and Web applications. Key building blocks of the *eXtreme*DB solution include:

***eXtreme*DB-64**. This 64-bit IMDS increases the amount of memory *eXtreme*DB can address from a few gigabytes to terabytes. In a benchmark, an *eXtreme*DB-64 database grew to 1.17 terabytes and 15.54 billion rows, with near-linear scalability. For a simple SELECT against the fully populated database, *eXtreme*DB-64 processed 87.78 million query transactions per second using its native API and 28.14 million query transactions per second using the SQL ODBC API.

***eXtreme*DB High Availability (HA)** enables deployment of synchronized database instances (a "master" and one or more replicas), with automatic failover. If the master database is brought down by hardware or software failure, a replica takes over its job, to maintain system operation. Replication can be configured either as "1-safe", in which a transaction is committed on the primary node without waiting

for updates to occur on replicas; or “2-safe”, where the transaction is committed only after it is applied to the master and all replicas. Developers use these options to tune availability and performance to meet system requirements.

***eXtreme*DB Fusion** provides the option of on-disk storage for selected record types, while the rest are managed in RAM. File I/O and other overhead are incurred only when absolutely necessary. With *eXtreme*DB Fusion, developers can take advantage of the persistence of traditional on-disk DBMSs, and the high performance of IMDSs.

Multi-Version Concurrency Control (MVCC). An optional MVCC Transaction Manager eliminates locking when regulating access to the database. No task or thread is ever blocked by another because each is given its own copy (version) of objects in the database to work with during a transaction. MVCC significantly improves performance and scalability, particularly when many tasks are running on multiple CPU cores.

McObject’s latest updates to the *eXtreme*DB product family, *eXtreme*DB Data Relay and *eXtreme*DB Cluster, further extend the technology’s scalability, reliability and interoperability:

*eXtreme*DB Data Relay

Certain IT scenarios present a need to continuously and selectively update an external database with information from a real-time system. For example, a securities trading application may update its real-time database with both transient information, such as stock price ticks, and with data that needs to be treated specially (aggregated or analyzed, for example), such as trade executions. Writing an application to select the desired sub-set of real-time information would be time-consuming and an inefficient use of CPU cycles.

McObject addresses this need with its ***eXtreme*DB Data Relay** technology. Data Relay maintains a buffer of *eXtreme*DB database transactions as they occur. Developers can use a familiar database “cursor” to iterate over objects in the transaction buffer, and propagate those changed objects (or not, by inspecting the object values to determine whether those objects should be propagated).

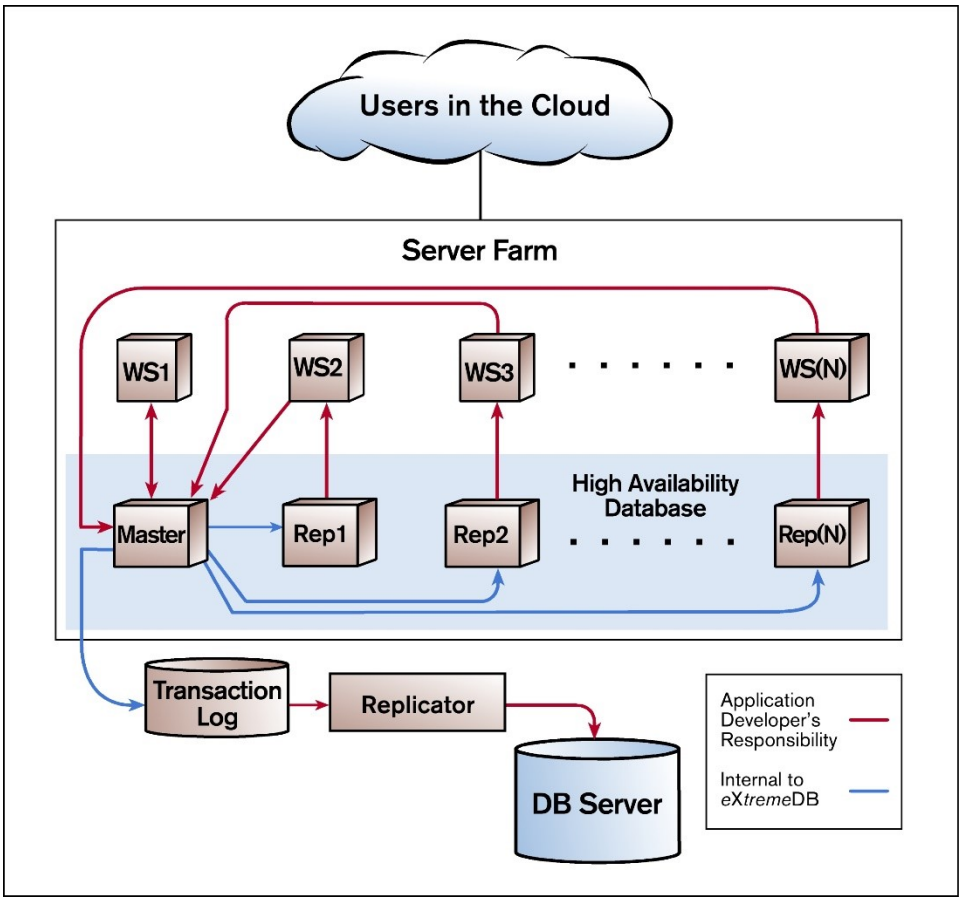

Figure 4. In this diagram, *eXtreme*DB Data Relay runs in tandem with the *eXtreme*DB High Availability IMDS to relay information from a real-time system to an external DBMS. *eXtreme*DB-HA resides at the Master node and propagates changes to *eXtreme*DB database instances at replica nodes, which in turn supply data to Web servers. Updates made by users (via the Web servers) go directly to the Master database, which then updates replicas. Meanwhile, *eXtreme*DB Data Relay (represented by the Replicator box) sends changes in the Master *eXtreme*DB database to an enterprise database (represented by DB Server).

*eXtreme*DB Cluster

*eXtreme*DB Cluster is the latest specialized edition of McObject's proven IMDS. It provides the highest level of scalability and availability by distributing the real-time database across multiple hardware nodes. Every database instance serves as a "master." Any process on any node can update its local database, and the Cluster software will replicate the changes to other nodes in the cluster. This architecture greatly increases the net processing power available to users, resulting in much faster database activity (adds/reads/deletes).

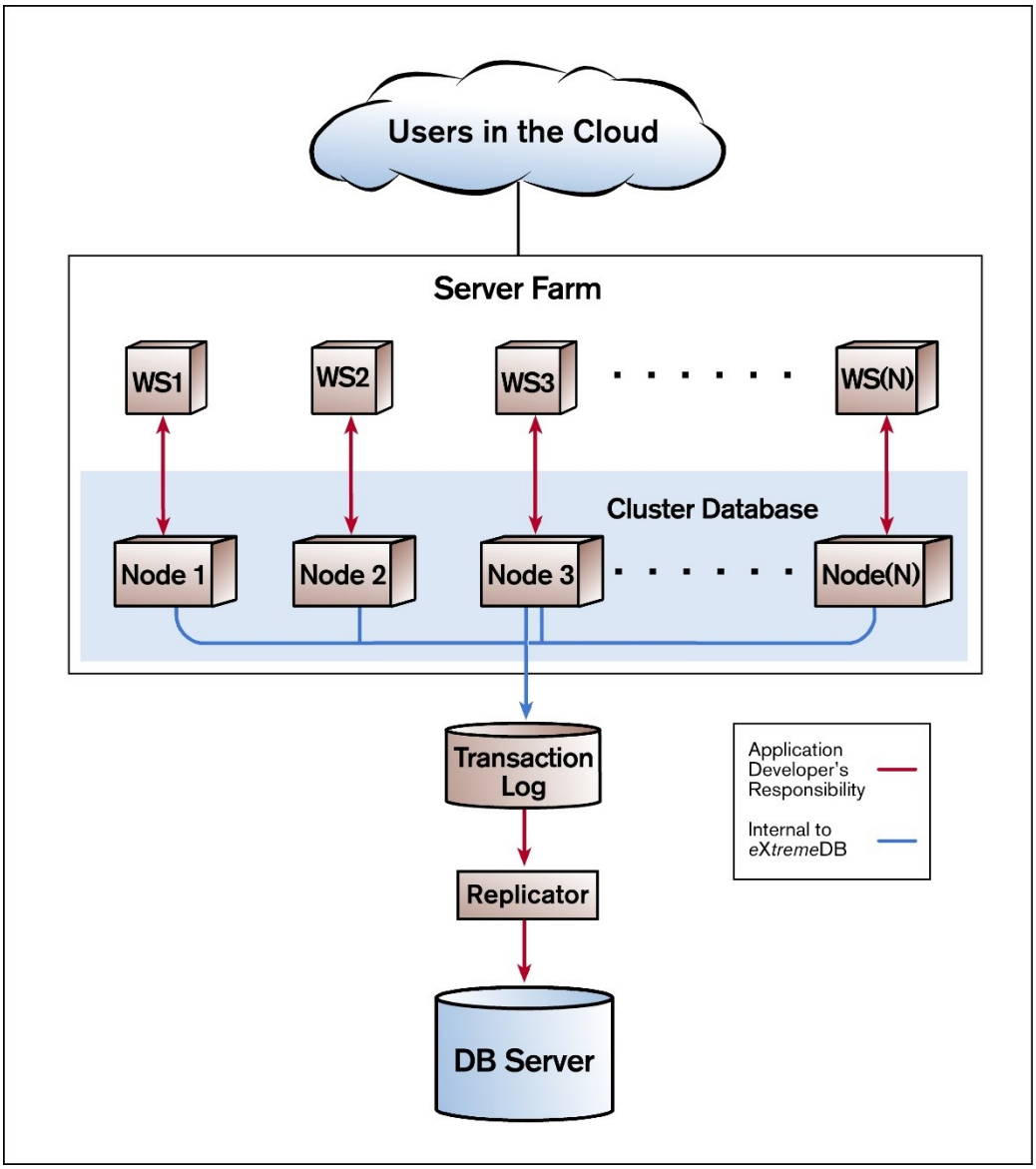

Figure 5. In this depiction, instances of *eXtreme*DB Cluster reside at nodes 1…N. Each node is read/write (eliminating the master). The figure shows a 1-1 paring of web servers and Cluster nodes, though this is not required; both Web servers and Cluster nodes likely run on commodity servers. *eXtreme*DB Data Relay is set up on Node 3 to propagate changes in the shared database to the enterprise DB Server (the choice of Node 3 is arbitrary. Data Relay could be used at any node, or on more than one node). To gain IMDS scalability, such a configuration can also take advantage of *eXtreme*DB's 64-bit support.

The hardware for each node can be a low-cost (i.e. "commodity") server, so that the system can expand cost-effectively. Distributing the system across multiple nodes/hosts ensures continuous availability in the event of a data node, hardware or network failure. *eXtreme*DB Cluster's "shared nothing" architecture eliminates reliance on a shared SAN or other storage resource. Figure 5 shows a typical

deployment using *eXtreme*DB Cluster, with *eXtreme*DB Data Relay operating on one node, to update an external DBMS with real-time information.

*eXtreme*DB Cluster supports the same ACID transactions offered by the non-clustering *eXtreme*DB editions, making it an attractive choice for applications that demand integrity of distributed data. The clustering mechanism is based on the concept of a quorum. The application defines a minimum number of nodes that must be communicating in order for clustering to be active. If this number falls below a quorum, *eXtreme*DB Cluster returns an error message. Application logic determines how this is handled (options include lowering the quorum size, or enabling the nodes to continue operating on a non-clustered basis).

*eXtreme*DB Cluster builds on *eXtreme*DB 's proven IMDS, HA, 64-bit, Fusion, and MVCC strengths to eliminate barriers to scalability. Its core in-memory architecture overcomes the I/O and caching bottlenecks inherent in disk-based DBMSs. 64-bit support allows in-memory databases to grow to terabyte-plus sizes; with *eXtreme*DB Fusion, on-disk database sizes are limited only by available file system space (in either 32-bit or 64-bit implementations); MVCC allows threads to simultaneously insert/update/delete the database without having to be queued by the transaction manager; and, finally, *eXtreme*DB Cluster enables two or more servers to share the workload, eliminating any ceiling imposed by being CPU-bound on a single server.

Conclusion

As high-performance applications move to the Web and require more scalable underpinnings, system designers are looking beyond the RDBMSs that have served them well for decades. Emerging data management technologies deliver accelerated performance, even as they serve growing numbers of concurrent users. For software that presents only moderate performance needs, the existing RDBMS may apply. Systems that require scalability and higher performance, but do not handle mission critical data, may be appropriate for object-caching or NoSQL solutions, although developers may sorely miss the development efficiency of a full-featured DBMS API, run-time optimization provided by database indexes, data integrity protection of ACID transactions, and other features.

An in-memory database system can provide performance equal to or better than a caching engine, but with all the functionality of a full-fledged database. With features such as 64-bit support, database replication and hybrid data storage, IMDSs become a plausible solution for high volume, real-time systems. With clustering, IMDSs offer almost unlimited scalability, “real” database characteristics (APIs, ACID transactions, indexes, etc.), and the ability to use low-cost commodity hardware. All of the data management solutions discussed in this paper present tradeoffs along the axes of scalability, cost, reliability, and development and run-time efficiency. System designers must match against specific application needs when choosing among them.