# VMware vFabric GemFire®

## High Performance, Distributed Main-Memory and Events Platform

TECHNICAL WHITE PAPER

vmware®

## Table of Contents

# Premise

Modern applications are putting a strain on traditional database technology. These new style applications often serve hundreds of thousands—or even millions—of users across wide geographies with real-time requirements. In addition, applications are no longer accessed solely from personal computers, but rather from connected devices of all stripes and at all hours. A one-size-fits-all database management system designed forty years ago for an entirely different set of requirements simply cannot keep up. This technical paper explores how memory-oriented data systems meet the needs for many modern applications.

## Traditional Database Design Focus: ACID Transactions and I/O

Traditional database design is based on strict adherence to ACID (atomicity, consistency, isolation, durability) transactional properties. In practice, when coordinated, enterprise-wide processes are built by integrating stovepipe applications, the individual database becomes merely one participant in a complex transaction that involves many sources of data—with no single coordinator ensuring data consistency across disparate systems. Often the application (or an interfacing human) acts as the coordinator and must apply compensatory action when failures occur.

Traditional databases have taken a one-size-fits-all approach to data management[1] with an overly centralized, disk-oriented design (depicted in Figure 1) that makes extensive use of locking for concurrency control. As the features they offer have increased over the years, the complexity of database engines has increased. By its nature, the design impedes highly concurrent data access. It is highly optimized to address the single biggest bottleneck for disk-oriented databases: disk I/O performance. That choice makes perfect sense, given that disk access can be 100 times slower than RAM. Initial attempts at memory-based data repositories were based on centralized designs and focused only on preserving ACID properties for all data access[4].

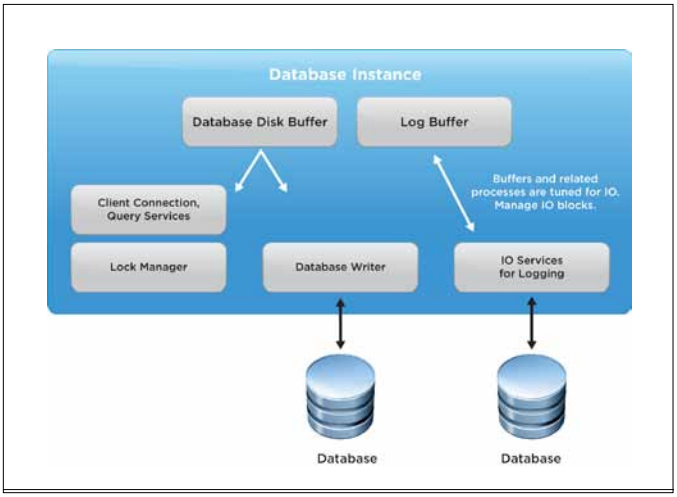



**Figure 1:** Typical Relational Database Architecture

In contrast, VMware® vFabric™ GemFire® has the goal of completely eliminating disk access as a bottleneck. Its design is primarily optimized for memory and—with data distributed across a cluster of nodes—for scalability and performance. The GemFire design presumes no requirement for ACID transactions. The developer can choose the degree of consistency required for concurrent data modifications—and either apply that choice across the entire system or tailor it for specific data sets or logical data partitions. As a result, applications concerned about the scalability and availability characteristics of data can trade off, or relax, the needs for strict data consistency while preserving strict consistency in other parts of the system as needed. In other words, GemFire enables system designers to optimize high-performance solutions by flexibly tuning consistency, availability and partition-tolerance at different places in their applications.

Traditional online transaction processing (OLTP) system design promotes the idea of a database tier for managing all data. This approach undervalues the significant need for temporal data management in the application tier—session state, workflow state and the caching of frequently accessed data. Many caching solutions exist today, and often caching is incorporated into middle-tier platforms. However, these implementations are too relaxed with respect to data consistency, lack support for managing referential data integrity or are limited by the available process memory on a single machine.

## Managing Data in Cluster Memory and Local Disk

The design premise of GemFire is to distribute data across a cluster of nodes, providing a choice between full replication to selected nodes and fault-tolerant partitions of data across selected nodes. Although GemFire supports logging of every single update to disk, that approach can often be avoided because the update is synchronously applied to one or more replicas in memory. GemFire allows data to be queried, to participate in transactions, to share the application's process space and to be synchronized with external data repositories (synchronously or asynchronously). The working data set for most OLTP applications is small enough to fit in the composite memory of a reasonable number of commodity servers. GemFire shifts the focus from disk I/O optimization to optimization for distribution. The internal data structures are optimized to conserve memory as a resource and are designed for highly concurrent access.

GemFire uses a radically different storage architecture from traditional databases to protect against data loss if multiple nodes in the cluster fail or are shut down. Writes are "journaled" into rolling append-only logs on each cluster node. (Each replica streams its writes to disk in parallel.) With no disk seeks and parallelizing writes across all the disks in the cluster, very high aggregate write throughput is achieved (with the theoretical maximum being the aggregate disk-transfer rates of all the disks combined).

To effectively address continuously changing load patterns across applications and to optimize resource utilization, GemFire is designed to adapt to dynamic expansion or contraction in the cluster's capacity that might accompany demand spikes or ebbs. The design can also handle a myriad set of failure conditions. Data availability is ensured through efficient replication techniques among distributed processes or replication across clusters that are geographically isolated. Because GemFire replaces disk I/O with memory access and can capitalize on the increased, reliable network bandwidth now available, it can provide near-instantaneous propagation of updates across networks. The benefits observed in customer deployments indicate that this approach performs far better and is more cost-effective than database replication or the use of highly available disk-storage clusters. GemFire presents a clear opportunity for quick ROI.

## Scaling with Reliable Messaging

Modern real-time architectures are realized by using reliable messaging platforms to asynchronously push events between services and applications. It is common for different services connected through messaging systems also to share fast-moving data in databases. For instance, a trade-order management system publishes new arriving orders into an orders database and pushes new-order messages to a trade-execution engine. The trade engine responds by accessing the order details in the orders database. But this simple design pattern often exposes the application to a number of challenges: Events can be delivered before the dependent data arrives in the database; and processing latency caused by disk access becomes a bottleneck in the database and very likely in the messaging system as well. Solving these problems requires a complex two-phase commit protocol between messaging systems and databases, further reducing the scalability of the overall design.

GemFire, as an alternative, enables applications to subscribe directly to data changes through interest registration and query expressions. Any changes to the data or its relationships are propagated reliably as change notifications to the subscriber. GemFire combines the power of stream data-processing capabilities with traditional database management. It extends the semantics offered in database management systems with support for *continuous querying* that eliminates the need for application polling and supports the rich semantics of event-driven architectures. With GemFire, applications can execute ad-hoc queries or register queries for continuous execution. Data updates are evaluated against registered queries, and change notifications are reliably propagated through the data fabric to interested, distributed applications.

## "Data-Aware" Behavior and Parallel Execution

Database vendors introduced stored procedures to enable applications to offload data-intensive application logic to the database node, where the behavior can be collocated with the data. GemFire extends this paradigm by supporting invocation of application-defined functions on highly partitioned data nodes such that the behavior execution itself can be parallelized. Behavior is routed to and executed on the node or nodes that host the data required by the behavior. Application functions can be executed on just one node, executed in parallel on a subset of nodes or executed in parallel on all the nodes. This programming model is similar to the MapReduce model popularized by Google.

As with stored procedures, applications can pass an arbitrary list of arguments and expect one or more results. In addition, the function-invocation API allows the application to hint about the keys (corresponding to entries managed in the data regions) that the function might depend on. These keys provide the routing information required for precisely identifying the *members* (distributed processes) where the function should be parallelized. If no routing information is provided, the function is executed only on a single data host or in parallel on all the data hosts. If the function is parallelized, results from each member are streamed back, aggregated and returned to the caller. The application can also provide custom aggregation algorithms.

Data-aware function routing is most appropriate for applications that require iteration over multiple data items (such as query or custom-aggregation functions). Overall processing throughput dramatically increases for applications that use GemFire features to collocate the relevant data and parallelize the function. More important, the calculation latency is inversely proportional to the number of nodes on which the function can be parallelized. GemFire customers have used these features to reduce the time it takes to complete complex calculations from hours to seconds.

The rest of this paper reviews additional important concepts and features offered in GemFire and exposes elements of the runtime architecture that speak to the design's low-latency, data-consistency and high-availability aspects. It provides a glimpse into the programming model and how reliable publish-subscribe semantics is offered as a core part of the distributed container. Finally, it describes a simple performance benchmark.

# Object Storage Model

For simplicity and high performance, GemFire manages data as object entries (key-value pairs) in concurrent hash maps. The average GemFire application accesses the data through an enhanced map interface (for instance, as `java.util.Map` in Java) and configures a variety of attributes that control options such as

- Replication (Will synchronized copies be maintained on each node?)
- Partitioning (Will the entries be striped across nodes?)
- Disk storage (Will the data only be in memory, overflow to disk or be persisted to disk?)
- Eviction (Will least recently used entries be evicted when memory usage exceeds a certain threshold?)
- Caching plug-ins to
  - Respond to events (on the server or in clients)
  - Perform read-through (load from external source on a read miss)
  - Perform write-through (synchronously write to external source)
  - Perform write-behind (asynchronously write back to an external data source)

Arbitrarily complex objects can be stored in GemFire. Objects are formatted using either built-in object serialization (i.e., Java or .NET serialization) or an optimized format that supports object versioning and interoperability across multiple languages. This collection of key-value pairs, along with the configurable attributes, constitutes a *data region*. In some sense, a data region is similar to a relational table—except that it is typically managed across multiple servers. For instance, you could configure a replicated data region for managing product-catalog objects and have a large partitioned data region for managing millions of orders spread across many nodes.

Object fields in keys or values in the region can be indexed, and regions can be queried using Object Query Language (OQL)—an Object Data Management Group (ODMG) standard similar to SQL. The querying features do not include all of the fringe modifiers offered in modern relational databases but are instead designed to offer very high performance, especially for queries based on Boolean or relational operators (AND, OR, <, > , etc.). Main-memory indexes can be created on any of the object fields (in the root object or a nested field in the object graph). The query engine uses a cost-based optimizer to effectively utilize indexes.

Data regions can themselves be nested and contain child data regions. All the data regions fit into a single logical namespace that is accessible from any member that participates in a GemFire distributed system (further described below). A typical deployment would consist of a cluster (tens to hundreds) of cache-server nodes that manage the data regions, and a larger number (hundreds to thousands) of client nodes that access the cache servers. The cache server could be launched using scripts or through APIs. Each cache server defines the data regions it will manage declaratively (using XML) or explicitly using an API. The client nodes accessing the cache servers could themselves also host a local edge or Level 1 (L1) cache for high, in-process cache reads but always delegate to the servers when the data needs to be updated.

These in-process, edge caches can be hosted in C++, Java or .NET client processes. All cache servers form a peer-to-peer (P2P) network and always run in a Java Virtual Machine (JVM). GemFire uses a language-neutral binary format for object representation on the wire that enables data interchange across different languages at very high speeds without incurring the traditional costs associated with using a self-describing interchange format like XML.

Figure 2 depicts the high-level architecture of GemFire.

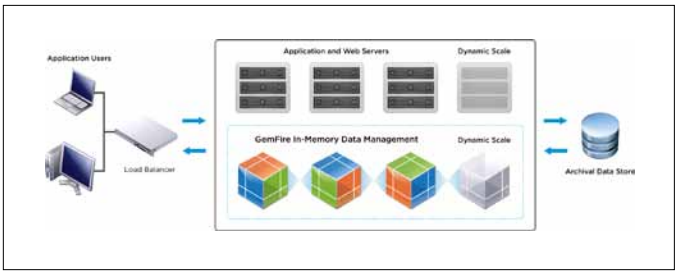
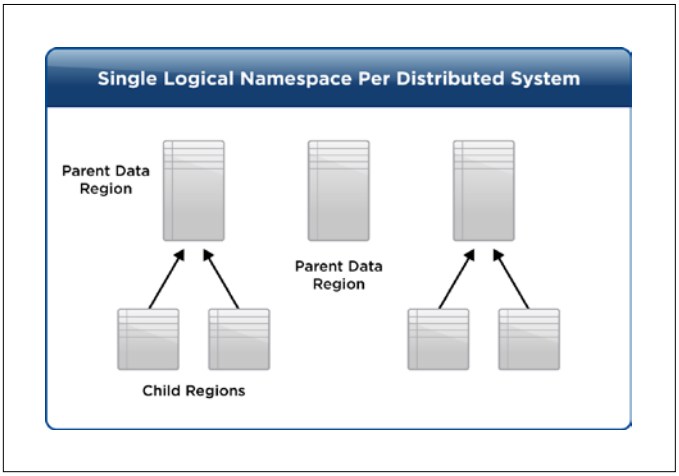



**Figure 2:** VMware vFabric GemFire Architecture



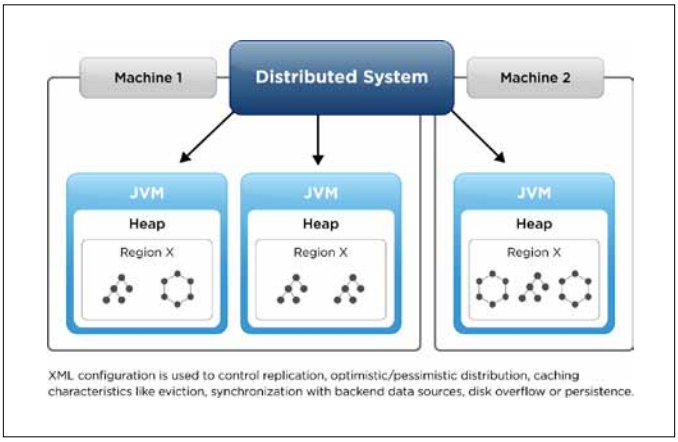



**Figure 3**. GemFire Schema Based on Data Regions

### Dynamic Membership-Based Distributed System

In GemFire, members that host data connect to one another in a P2P network to form a *distributed system*. GemFire supports dynamic group membership, whereby members can join or leave the distributed system at any time with minimal impact on other members. This ability to dynamically alter capacity is the most important characteristic enabling stateful applications to be built without overprovisioning for peak demands—while still targeting specific service-level agreement (SLA) goals for data, such as availability or performance. Membership changes do not introduce locking or contention points with the other members. Members can discover one another either by subscribing to a common multicast channel or by using a TCP-based discovery service (called a *locator*) if the network is not enabled for multicast. The system automatically elects a group membership coordinator—a member that is responsible for allowing new members to join the distributed system and for communicating any membership changes in a consistent fashion to all members.

When GemFire is used as an embedded data fabric within a clustered application, access to any data within the cluster will incur at most a single network hop, with each application node being directly connected to every other member of the distributed system. In this model—if the data is managed in a partitioned manner and if all concurrent access to the data is uniformly distributed across the entire data set—increasing the capacity (the application cluster size) would linearly increase the aggregate throughput for data access and processing, assuming network bandwidth doesn't become a bottleneck. Additionally, the at-most-a-single-hop access to any data offers predictable data-access latency.

### Failure Detection

GemFire is commonly used to build systems that offer predictable latency and continuous availability. When any member of the distributed system fails, it is important for other services to detect the failure very quickly and transition application clients to other members. GemFire uses multiple techniques to detect failure conditions. When a member departs, normally other members are notified immediately through a dedicated membership event channel. If a member departs abnormally, GemFire detects this condition using a combination of TCP/IP stream-socket connections and User Datagram Protocol (UDP) datagram heartbeats. (Heartbeats are sent by each member to one of its neighbors, forming a ring for failure detection.) Failure to respond to the heartbeat is communicated to other members. If all the members agree that the offending member is a suspect, it is removed from the membership in the distributed system.

When communicating with peer replicas, GemFire defaults to synchronous communication using an ACK-based protocol. Lack of ACKs within a configured time interval automatically triggers suspect processing with the coordinator, which in turn makes the final determination. GemFire supports numerous properties for configuring the timings and tolerances of the failure-detection system, enabling it to be tailored to the conditions experienced in different network environments. It also includes specialized configuration for diagnosing and properly handling "temporarily slow" members differently from failed members—a control that can avoid dangerous "membership thrashing" that might otherwise occur. The important point is that the distributed-system membership view is kept consistent across all members.

## Deployment Architecture

GemFire supports three primary architectures:

- **P2P** – All the members have direct connectivity to one another.
- **Client-server or "super-peer" model** – Client application processes connect and load-balance across a subset of P2P GemFire cache servers. Each client process can employ an edge, or "near," cache.
- **Gateway-connected distributed systems** – Any P2P distributed system can be configured to replicate all or part of its managed data using asynchronous store-and-forward gateways to one or more remote distributed systems over LAN- or WAN-based connections.

### P2P

In an *embedded P2P* GemFire architecture, application logic runs locally, in process, with cached data residing on a set of server nodes. The embedded P2P architecture is most suitable when the applications are long-running or continuously running, and the number of processing nodes is relatively stable. Short-lived processes can overwhelm the group-membership system and could also be daunting (from a manageability perspective) if they run over a very large number of nodes.

Although all members of a P2P architecture are equal from the point of view of membership, different members often play different roles in the architecture. Some members might host no data but still process events.

P2P architectures are commonly used to distribute a large data set, and the processing associated with it, across a linearly scalable set of servers. This model is also used when GemFire is embedded within clustered Java Platform, Enterprise Edition (JEE) application servers, as depicted in Figure 4.

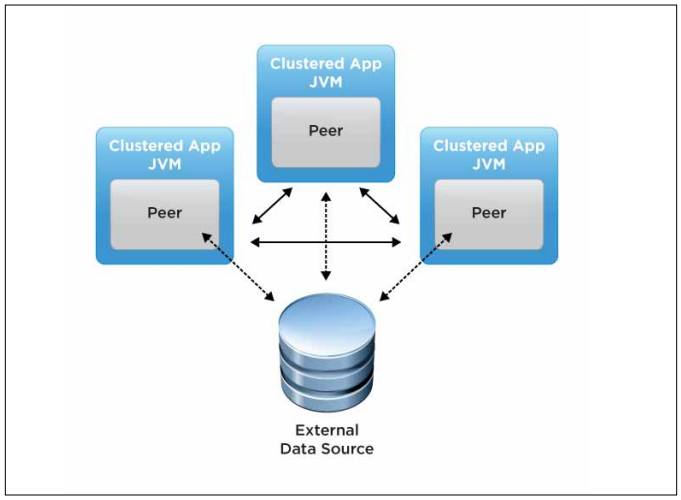



**Figure 4**. GemFire Embedded Within Clustered JEE Application Servers

## Super-Peers (Client-Server)

A *super-peer* is a member of the distributed-system network that operates both as a server to a set of clients, and as an equal in a network of peers. GemFire super-peer architectures can support thousands of client connections (such as in a computational grid application), load-balanced across a P2P server cluster. The clients delegate their requests to one of the super-peers, resulting in a highly scalable architecture. The trade-off (compared with the P2P architecture) is that it can potentially require an additional network hop to access partitioned data. These clients themselves can host an edge cache with the most frequently accessed data stored locally. Edge-cache consistency is maintained through expiry or by configuring the servers to automatically push invalidations or data updates to the clients. Clients always connect to servers using TCP/IP.

To scale the number of concurrent connections handled per server, nonblocking I/O is used to multiplex a large number of incoming connections to a configurable number of worker threads. The incoming channels use a message-streaming protocol that prevents the socket buffers from being overwhelmed and from consuming too much memory. For instance, with a simple communication infrastructure, 1,000 connections reading or writing a 1-MB object will require 1GB of buffer space if no streaming or chunking of the message is being used, expending memory that could otherwise be used to manage cached data.

Load information from each peer server is continuously aggregated in the locator service and used to dynamically condition the client load across all super-peer servers. Essentially, each client can be configured to transparently acquire load information, reconnect to a less loaded server and drop the existing connection. GemFire borrows several concepts highlighted in Staged Event-Driven Architecture (SEDA)[3]. In SEDA, each server can be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity.

## Gateway-Connected Distributed Systems

The P2P distributed system design requires that the peers be tightly coupled and share a high-speed network backplane. Traffic among peers is generally synchronous and can be very chatty. To distribute data across members that span WAN boundaries (or across multiple clusters within a single network), GemFire offers a novel approach that extends the super-peer model: a gateway feature that provides asynchronous replication to remote distributed systems. The gateway design is depicted in Figure 5.

A gateway process listens for update, delete and insert events in one or more data regions and then enqueues the events in a fault-tolerant manner (in memory or on disk) for delivery to a remote system. Event ordering is preserved inside of the queues. When multiple updates are made against the same key, the events can optionally be *conflated* (i.e., earlier updates are dropped from the queue and the latest update stays in the order it occurred). Events that are sent to a remote system are batched based on the optimal message size for the network in use between the distributed systems. Gateways can be configured to be unidirectional or bidirectional and to receive events from multiple remote distributed systems. Gateways can be used to create a number of layouts and architectures, such as rings, hub-and-spoke, and combinations thereof.

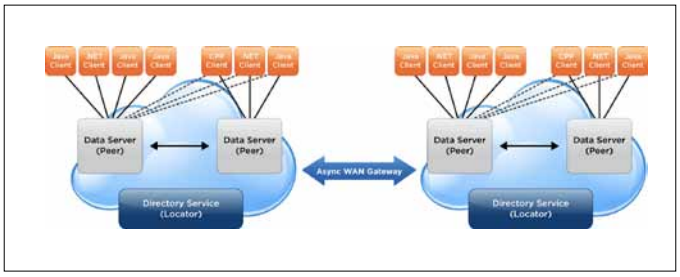



**Figure 5**. Gateway-Connected Distributed System

The gateway design is biased toward high availability of data and propagation of events with the lowest latency. To achieve sender-side continuous availability, events are enqueued on at least two members that play primary and secondary roles. Failure of a primary member results in election of a secondary member as the new primary to continue propagating enqueued events. Similarly, to guard against receiver-side failures, each sender node is visible to more than one receiver so that updates continue to be propagated if one of the receivers fails or becomes unresponsive.

# Replication and Consistency

The GemFire design is based on the following key principles:

- GemFire is designed for high performance based on the assumption that concurrent updates don't typically conflict (the window of conflict being measured in few milliseconds).
- When conflicts do occur, the application is in the best position to resolve the conflict. The application can be notified of the changes and of the object's prior state.
- Consistency, availability and partition-tolerance are tunable at functional points throughout system. For example, availability and scale-out with predictable performance can be prioritized, compared to maintaining ACID transactional properties for all data. Thus updates will not be rejected because of concurrent writes or failure conditions. ["Propagate operations quickly to avoid conflicts. While connected, propagate often and keep replicas in close synchronization. This will minimize divergence when disconnection does occur."[2]

The design assumes that distributed locking will be used at a coarse level and when access is infrequent. For instance, in an application that manages customer accounts, it's appropriate to use distributed locking to prevent multiple users from logging into a single account. It's preferable to avoid using pessimistic concurrency control for fine-grained operations.

## Replicated Regions

For replicated regions, all data in the region is replicated to every peer server node that hosts that region. Replicated Regions use a *'multiple masters'* replication scheme.

Multiple Masters: When a data region is purely replicated (not partitioned), there is no designated master for each data entry. Updates initiated from any member are concurrently propagated to each member that hosts the region. Upon successful processing of the received event, an ACK is sent as a response to the initiating member. The initiating member waits for an ACK from each replica before returning from the data update call.

For updates to replicated regions, consistency is configurable, with the following options:

- **Distribution without ACKS (d-no-ACK)** – This is the most optimistic model. Replication is eager, assuming the traffic on the network has no congestion. GemFire, when using TCP as the transport, turns off the Nagle algorithm and avoids sender-side buffering of packets. When it uses UDP or UDP multicast transports, sender-side buffering is typically done (unless GemFire flow control kicks in to slow traffic on the channel because of negative acknowledgements). The sender does not wait for a response from the replica and returns control to the application the moment the message is routed to the transport layer.

  Applications use d-no-ACK when lost updates can be tolerated. For instance, in financial trading applications, continuous price updates on a very active instrument can tolerate data-update misses because a new update will replace the value within a short time window. In practice, distribution failures that are due to problems in the transport layer are raised as alerts notifying network administrators. Also, the use of multiple failure-detection protocols in the system will forcibly disconnect a member from the distributed system if it has become unresponsive, reducing the probability of replicas diverging from one another for too long.
- **Distribution with ACKs (d-ACK)** – Replication is eager. The message is dispatched to each member in parallel, and ACKs are processed as they arrive from each receiver. The invocation completes only after all the ACKs (one for each replica) have been received. Any update to an existing object is done by swizzling to a new object to provide atomicity when multiple fields in an object are modified. Along with sending the entire object over the wire, GemFire also supports sending just the updated fields (the *delta*) to replicas.

  To guarantee the atomicity, by default, GemFire clones the existing object, applies the delta and then replaces the object. The application can choose to serialize concurrent threads using Java synchronization and avoid the costs associated with cloning. Applications in which primitive fields are constantly updated using this mechanism can experience a dramatic boost in performance with significant reduction in the garbage generated in the "older generation" of the JVM.
- **Distribution with ACKs and locking (d-ACK with locking)** – This is similar to the d-ACK protocol, except that before propagation to the replicas occurs, distributed locks are acquired on the entry key. If the locks are granted to some other thread or process, the replication can block until the locks become available or time out.
- **Replication and transactions** – All transactions are initiated and terminated in a single thread. Any replica node that initiates a transaction acts as the transaction coordinator and engages the replicas only at commit time. No locks are acquired until commit time, and the design generally assumes that the transactional unit of work is small and that there are no conflicts. Conflicts are detected at commit time, and the transaction is automatically rolled back if it fails. Repeatable read-isolation level is provided when the data regions are accessed using keys.

  The design avoids the overhead and complexities associated with undo and redo logs in traditional database systems by associating the transactional working set with the thread of execution. This enables the working set to be efficiently transmitted as a single batch at commit time and also simplifies the query-engine design. The transactional working set manages the read set (all key-value pairs fetched in the scope of the transaction) as well as the dirty set (updated entries) and is used for queries only when the access is based on keys. Any OQL query executed within the scope of the transaction is executed only on the committed state. So, any transactional updates that need to be subsequently retrieved

within the scope of the transaction must fetch using primary keys. Essentially, repeatable read semantics is offered to applications that perform key-based access.

- **D-no-ACK replication in the face of "slow receivers"** – With synchronous communication to replicas, the sender can be throttled if any one of the receivers is unable to keep up with the rate of replication. Often, one or more receivers are momentarily slow. When this occurs, you might want the system to continue operating as quickly as the healthy members can. GemFire facilitates this by detecting the slowness of a receiver and automatically switching to asynchronous communication using a queue for that member. The queue can optionally be conflated so that continuous updates on the same set of keys are propagated only once. If the receiver is later able to catch up, the queue is removed and all communication becomes synchronous again.

### Partitioned Regions

For replicated regions, the data in the region is distributed across every peer server node that hosts that region. Some peer server nodes can also host replicas of partitions for backup purposes. Partitioned regions use a *single master* replication scheme: When data is partitioned across many members of the distributed system, the system ensures that only a single member at any moment owns an object entry (identified by a primary key). The key range itself is uniformly distributed across all the members hosting the partitioned region so that no single member becomes a scalability bottleneck. The member owning the object is responsible for propagating the changes to the replicas. All concurrent operations on the same entry are serialized by making sure that all replicas see the changes in the exact same order.

Essentially, when partitioned data regions are in use, GemFire ensures that all concurrent modifications to an object entry are atomic and isolated from one another, and that "total ordering" is preserved across all replicas. When any member fails, the ownership of the objects is transferred to an alternate node in a consistent manner (i.e., making sure that all peer servers have a consistent view of the new owner).

Because of the single master scheme used for partitioned regions, alternative consistency mechanisms are neither required nor available. Key locking is available through a lock service, but it is not enforced by the region. (I.e., all users must respect the lock protocol for it to be effective.)

Traditional optimistic replication uses lazy replication techniques designed to conserve bandwidth, increasing throughput through batching and lazily forwarding messages. Conflicts are discovered after they happen, and agreement on the final contents is reached incrementally. System availability is compromised for the sake of higher throughput.

GemFire, in contrast, uses an eager-replication model between peers by propagating to each replica in parallel and synchronously. The design is biased in favor of data availability and lowest possible latency for propagation of data changes. Because a peer eagerly propagates to each of its replicas, clients reading data can be load-balanced to any of the replicas.

## Horizontal Partitioning with Dynamic Rebalancing

GemFire supports horizontal partitioning of entries in a partitioned data region. The default partitioning strategy is based on a simple hashing algorithm applied against the entry's key. Alternatively, applications can configure custom partitioning strategies. When custom partitioning is used, data in multiple data regions can be collocated so that related data will always be located on the same server (still in different regions), regardless of any rebalancing activity.

Unlike static partitioning systems in which changes in the cluster size might require either a rehashing of all data or even a restart of the cluster, GemFire uses a variant of *consistent hashing*. GemFire partitioning starts with either the entry's key (default) or with a *routing object* returned from a custom partitioning algorithm. These objects are then hashed into logical buckets. The number of buckets is configurable but should always be set to a large prime number, because it cannot be changed without dropping the region from (or restarting) all servers hosting that region.

Buckets are then assigned to physical servers (processes) that are configured to host the region. Partition replicates are also based on the same buckets. The principal advantage with consistent hashing is that when capacity is changed (added or removed), it is not necessary to rehash all of the keys, and a potential migration of every entry is avoided. When new peer servers are added and the administrator initiates rebalancing, a small number of buckets are moved from multiple peers (based on the current load on existing members) to the new member. The goal of rebalancing is to achieve a fair load distribution across the entire system.

Each region can be configured to use a maximum amount of heap memory. That maximum can be expressed in terms of maximum number of entries, total cumulative entry size or a maximum percentage of the available heap. It can be configured separately on each server that hosts the region.

This fine control enables clusters to be built using heterogeneous server platforms and helps avoid the occurrence of hot spots (too much data stored on a single host that has limited resources).

The process of publishing a new entry into a partitioned data region is depicted in Figure 6. Each peer member maintains a consistent view of which buckets are assigned to each peer server. Because all peers maintain open communication channels to every other peer, a request for any data from any peer can be resolved with at most a single network hop.

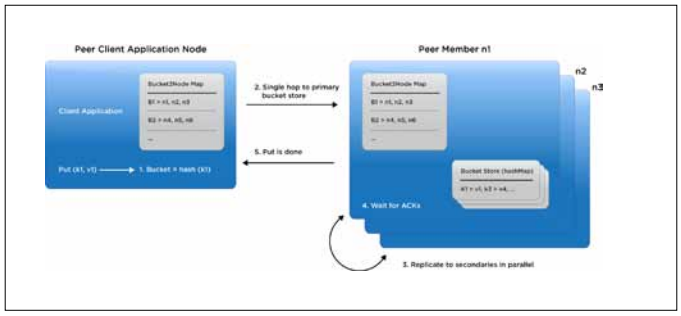



**Figure 6**. Publishing a New Entry into a Partitioned Data Region

One of the key design goals for GemFire was that an increase in the number of peer servers would translate to a near linearly proportional increase in throughput (or the ability to handle users) while maintaining predicable, low latency. Most of the current data-partitioning schemes in caching platforms or database designs are based on *sharding* and promise linear scaling while assuming that concurrent access will be uniform across all partitions at all times. In practice, data-access patterns change over time, causing uneven load across the partitions. A system that can adapt to changing data-access patterns and balance the load has a higher probability of scaling linearly.

In GemFire, each peer server continuously monitors its resource utilization (CPU, memory and network usage), throughput, garbage-collection pauses and latencies within different layers of the system. When uneven load patterns are detected, rebalancing can be triggered by applications (using a Java API) or through administrative action (via the Java Management Extensions [JMX] agent or GemFire tools). The smallest unit of migration is a single, entire bucket. Bucket migration is a nonblocking operation, meaning that data reads and updates can occur while the bucket is being migrated.

The creation of a new replica of the bucket is also a nonblocking operation, whereby all interleaved updates are also routed to the new replica. If a larger total number of buckets is defined for the region, each bucket will be smaller, resulting in smaller amounts of data being transferred when buckets migrate. Setting the number of buckets too high increases the administrative load on the cluster. As a rule of thumb, the maximum number of buckets should be a prime number that is 10 to 100 times larger than the expected maximum number of peer servers that will host the partitioned region.

# Partitioning with Redundancy

When redundancy is configured for partitioned data regions, the replication is synchronous and the update returns only after an explicit ACK from each replica has been received. Buckets are assigned to be either primary or secondary buckets. All reads are load-balanced across primary and secondary buckets, but writes are always coordinated by the primary and then routed to the secondary buckets. All inserts or updates to a specific key are serialized by the primary bucket, which applies the update locally and then sends the update to the secondary buckets. Replication to all secondary buckets is done in parallel. Serialization through the primary bucket ensures that all replicas see changes to any entry in the same order, ensuring consistency. If primary buckets are uniformly spread across all the eligible members, no single member becomes a choke point.

When a member departs (normally or abnormally) primary buckets can be lost. A primary-election process picks the oldest secondary bucket to be the new primary and communicates this change to all peer servers, enabling updates to continue without interruption. Then, if configured to maintain a certain redundancy level, the remaining members recover the lost buckets by replicating the newly elected primary buckets. If not enough members exist—or if capacity isn't sufficient to recover all the lost buckets—the system executes a best-effort algorithm and logs warnings on buckets with compromised redundancy levels.

Often, when machines are brought down for system maintenance, it is acceptable to allow the system to operate with reduced redundancy for short durations. Delays can be configured, causing the system to wait for certain duration before attempting to recover secondary buckets. This enables the node(s) to come back up, the original buckets to be re-created, primary ownership to be restored, and the system load and performance characteristics to return to the way they were before the machine(s) went down.

# Persistence: "Shared-Nothing Operations Logging"

Unlike a traditional database system, GemFire does not manage data and transaction logs in separate files. Its design principles are fundamentally different from those of typical clustered databases. First, disk storage is "shared-nothing": Each cluster member owns its disk store, eliminating process-level contention. Second, the design is biased in favor of memory. Instead of maintaining complex B-tree data structures on disk, GemFire assumes that complex query navigations will always be done through in-memory indexes. Third, the design uses rolling append-only log files to completely avoid disk seeks. Finally, the design preserves the rebalancing model in GemFire when capacity is increased or decreased—the disk data also relocates itself.

### Factors Contributing to Very High Disk Throughput

Three factors contribute to very high disk throughput:

- **Pooling** – Each cache instance manages its own disk store, and there is no disk contention between processes. Each partition can manage its data on local disk(s). Assuming that application write load can be uniformly balanced across the cluster, the aggregate disk throughput will be (disk transfer rate * NumOfPartitions), assuming a single disk per partition. Using GemFire, you can achieve disk-transfer rates up to 100MB/sec on commodity machines, compared with 2MB/sec in the 1980s, as shown in Table 1.

| | MID 1980S | 2009 | IMPROVEMENT |
|---|---|---|---|
| Disk capacity | 30 MB | 500 GB | 16667x |
| Maximum transfer rate | 2MB/s | 100 MB/s | 50x |
| Latency (seek + rotate) | 20 ms | 10 ms | 2x |

**Table 1.** Improvement in Disk-Transfer Rates

**Source:** John Ousterhout et al., "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," SIGOPS Operating Systems Review, Vol. 43, No. 4, December 2009, pp. 92-105

- **Avoiding seeks** – Because most (or all) of the data is managed in cluster memory, all reads are served without navigating through B-tree–based indexes and data files on disk, which would generate continuous seeking on disk. Average disk seek times today are still 2ms or higher.

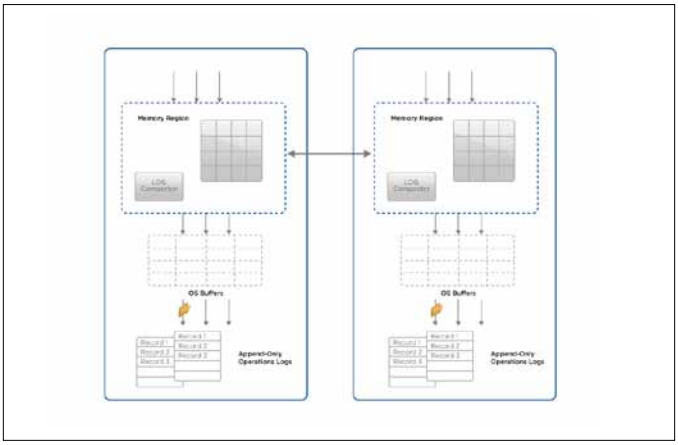


**Figure 7.** Buffered Logging

- **Buffered logging** – When writes do occur, they are logged to disk in append-only log or data files (see Figure 7). Appending implies the ability to continuously write to consecutive sectors on disk without requiring disk-head movement. Probably the most controversial GemFire design decision was to allow all writes to be flushed only to the OS buffer rather than fsync all the way to disk. The writes are buffered by the I/O subsystem in the kernel, enabling the I/O scheduler to merge and sort disk writes to achieve the highest possible disk throughput.

The write requests need not initiate any disk I/O until sometime in the future. Thus, from the user-application perspective, write requests stream at much higher speeds, unencumbered by disk performance. Any risk of data loss that is due to sudden hardware failure is mitigated by having multiple nodes write in parallel to disk. In fact, it is assumed that hardware will fail, especially in large clusters and datacenters, and software must take account of this. The system is designed to recover in parallel from disk and to guarantee data consistency when data copies on disk disagree with one another. Each member of the distributed system logs membership changes to its persistent files. These are used during recovery to determine the replica with the latest changes and automatically synchronize everything at startup.

### Advantages of Native GemFire Persistence over a Relational Database

Most data-grid deployments use a relational database as the backing store. A synchronous design—whereby every change is reflected in the database first—has obvious challenges for write-heavy applications: Throughput is only as fast as the database, and the database can become the single point of failure (SPOF). Moreover, the need for two-phase commits between data grid and database increases the complexity of transaction execution.

One remedy is to execute all writes on the data grid and asynchronously propagate the changes to the database. But this pattern poses the same SPOF challenges and is not well-suited for cases with sustained high write rates.

Designs that employ database *shards*—each cache instance writing to its own independent database instance for scale—are interesting but difficult to implement with good high-availability characteristics.

The GemFire parallel persistence model overcomes all of these limitations.

## Caching Plug-Ins

A common usage pattern with GemFire involves bootstrapping the distributed system from one or more data sources at startup time. Often the entire data set—or a well-defined subset of the data—is loaded into GemFire to act as a distributed cache. This enables the application to issue ad-hoc queries using OQL with any number of memory-based indexes on the managed data. Any updates to the data can be propagated synchronously (*write-through*) or asynchronously (*write-behind*) to the data sources. When ad-hoc queries are used and the entire data set is not held in the cache, eviction or expiry of data causes inconsistent data to be returned. The "Overflow to disk" option does not impact the accuracy of queries, because indexes and primary keys are always kept in memory.

When the application wants to use GemFire purely as a cache, all access to the data must be based on primary keys. This usage now permits the cache to evict objects that are not frequently used, and lazily load on a cache miss. When used as a cache, GemFire can also be plugged in as an L2 cache within Hibernate or as a caching interceptor within Spring containers.

GemFire offers the following artifacts to enable caching environments:

- **Least recently used (LRU) eviction** – A *clock-based algorithm* is used to efficiently evict entries when a certain eviction threshold is crossed. Multiple policies are configurable to control when the eviction is triggered. For instance, the policy could be based on the count of the entries managed in a data region, on the memory consumed by a data region or on when the heap usage exceeds a certain threshold. With data being managed in the JVM heap, the design takes into account how modern generational garbage collectors work. Generational garbage collectors avoid being too aggressive about eviction when most of the heap is consumed by garbage, and they avoid being too lazy and causing out-of-memory conditions. The action taken when the eviction thresholds are crossed is also configurable. For instance, one action could be to remove the entry from the cache, invalidate it so it can be refreshed from the data source or simply overflow to disk. When heap-based eviction is configured, the overflow to disk becomes a safety valve that prevents an out-of-memory condition by moving objects that aren't frequently accessed to disk.
- **Expiry** – Objects in the cache can be configured to have a lifetime (specified through a "Time to live" attribute). If the object remains inactive (no reads or writes) for the time period specified, the object can be removed, invalidated or moved to disk.
- **Read-through** – When an application attempts a read on a key that is not in the cache, a *data loader* callback can be invoked to load the entry from an external data source. Synchronization is used to prevent multiple concurrent threads from overwhelming the data source trying to fetch a popular key. These callbacks can be configured on any node in the distributed system. Irrespective of which member originates the request, the loader is invoked in the remote member, published in the cache and made available to the caller. Often, when data is partitioned, the loader is collocated and executed on the node that is hosting the data.
- **Write-through** – Invoking a *cache writer* callback synchronously propagates all updates. If and only if the writer is successful, the update becomes visible in the cache. If the writer is triggered on a thread that is currently in a transaction, the update to the data source can also participate in the transaction. If the cache is embedded in a container such as JEE, then GemFire—along with the writer callback—can participate in an externally coordinated Java Transaction API (JTA) transaction. When participating in an external coordinated transaction, GemFire does not register a global-transaction (XA) resource manager, but instead relies on the *before completion* and *after completion* callbacks to commit (or roll back) the transaction in the cache after the transaction outcome has been determined and communicated to any other participating resource managers (such as Java Database Connectivity [JDBC]).
- **Write-behind** – All updates are enqueued in the same order as seen by the distributed system and delivered asynchronously to a listener callback. The underlying machinery for how queues are managed in memory with one or more secondaries—and the batching, conflation and durability semantics—are the same as used for asynchronous replication to remote distributed systems (see the Deployment Architectures section).

# Programming Model: "Hello World" Example

To illustrate the programming model, we assume that the deployment uses a cluster of GemFire cache-server nodes managing the data, and clients accessing data using keys and OQL queries. We also assume that the client is a Java application that embeds a local cache. (The C++ and C# APIs are very similar to the Java API.)

The typical development model consists of the following steps:

**1. Configure Cache Servers (Create <cache.xml>)**

In this step, you describe declaratively which data is being managed on any member node.

Each cache server is provided with an XML description of the cache it will host during startup.

Here is an example:.

```
<cache>

     <!-- Each cache instance declares one or more data regions it wants to host -->
     <region name="Customers" refid = "PARTITION_REDUNDANT">
        <!-- Customers records will be hash partitioned on the key across
             all data hosts. Each entry will be redundantly copied (synchronous) to
             different physical node in the cluster. Replication uses an ACK based
             protocol.
             -->

             <region-attributes >
                 <!-- Application wants to explicitly load the data
                from a mysql database upon a cache miss -->
              <cache-loader>
                     <class-name>com.company.data.DatabaseLoader</class-name>
                           <parameter name="URL">
                               <string>jdbc:mysql://myObeseHost/UberDatabase</string>

                             </parameter>
               </cache-loader>
             </region-attributes>
      </region>
</cache>
```

Each cache server announces itself to a discovery service so it is easy for clients to load balance across a farm of cache servers. This discovery service is called the 'locator'.

## 2. Start the Cache Server Locator Using the GemFire Command-Line Tool

```
shell > gemfire start-locator -server=true -port=41111
```

## 3. Launch the Cache Servers

The cache servers are the servers that manage the application data. Each server is connected to the others in a P2P network to distribute data among them quickly and detect any failure conditions in the system. Servers can discover one another using the locator or optionally use a multicast channel.

Use the following command to start the cache servers on each node in the cluster where the application wants to manage data:

```
 > cacheserver start -J-Xmx8000m locator-port=41111 cache-xml-file=<location of cache.xml>
-classpath=<classpath>
```

Here -J-Xmx8000m specifies the maximum heap available for data, and the classpath specifies the location of the application classes that are being invoked to load data.

GemFire cache servers can also be launched using APIs and can be embedded within any Java process.

## 4. Code the Java Client

The following code snippet Illustrates connecting to the server cluster and creating a local edge cache for fast access. The client application discovers all the servers by connecting to the locator and dynamically establishes connections to the appropriate server based on client requests.

Creating a "region factory" using the **CACHING_PROXY** template instructs GemFire to create a local LRU cache.

```
ClientCache cache = new ClientCacheFactory()
      .addPoolLocator("localhost", 41111)
      .create();

Map Customers = cache.<String,
Customer>createClientRegionFactory(CACHING_PROXY).create("customers");

// Put some customer objects
 customers.put(<customerKey>, <Serializable Customer object>);

 // get a customer
 Customer someCust = (Customer)customers.get(<custId>);

 // A simple OQL query illustrating navigating the objects using methods or public fields
SelectResults results = customers.query("select distinct * from customers c where c.getAddresses().size
>1");

 // results is a collection of objects of the type managed in the region

 // executing batch puts by using the 'Region' interface
 Region custRegion = (Region) customers;
 custRegion.putAll(Map<K,V> someMapOfCustomers);
```

# Reliable Publish-Subscribe and Continuous Querying

With the core system built for efficient and reliable data distribution, GemFire enables client applications to express interest in rapidly changing data and reliably delivers data-change notifications to the clients. Unlike some database solutions, in which update event processing is built as a layer on top of the core engine, the GemFire design incorporates messaging capabilities as a fundamental building block.

Applications that update data in a shared database and dispatch change events to subscribers through a messaging solution, such as JMS, are common. The subscribers are often components or services that closely cooperate and share the same database. Essentially, the services are loosely coupled from an availability standpoint but are tightly coupled at a data-structure level because of the shared database. The application design is often complicated by the need to do two-phase commits between the database and the messaging server, which are inherently slow. This architecture also exposes the systems to increased failure conditions that must be handled at the application level—and it exposes the need to deal with race conditions caused by multiple, asynchronous channels (message bus and replicated databases). For instance, an incoming message causes the application to look for related data in a replicated database that has not arrived yet, resulting in an exception.

GemFire avoids all these issues by combining the data-management aspects with the messaging aspects for applications running in distributed environments. The characteristics that distinguish the active nature of GemFire from a traditional messaging system are continuous querying, delta propagation, the availability of contextual information at memory speeds, and high availability and durability through memory-based replication.

## Continuous Querying

GemFire clients can subscribe to data regions by expressing interest in the entire data region, specific keys, or a subset of keys identified using an OQL query expression or a regular expression. When a query is initially registered, a result set from the evaluation of the query is returned to the client. Subsequently, any updates to the data region result in automatic, continuous query evaluation, and updates satisfying the registered queries are automatically pushed to the subscribing clients.

## Delta Propagation

When clients update any existing objects, they have the choice to capture only the *delta* (i.e., the updated fields) and propagate just the delta across the distributed system all the way to subscribing clients that host a local edge cache. This ability to capture and propagate just the updated fields to a subscriber saves bandwidth and can potentially deliver higher message throughput.

## Availability of Contextual Information at Memory Speeds

Unlike in a database, in which individual data objects can have relationships, no inherent relationships exist between multiple messages in a messaging system. The sender can choose either to include the contextual information as part of the payload or to force the receiver to use a shared database to retrieve the context required to process the messages. The first choice slows down the messaging middleware (because it must push more data through its guaranteed delivery system). The second slows down the database (because of additional, redundant queries). So, even if the speed of message delivery were high, the subscriber could still be throttled by the speed of access to the databases.

With GemFire, event notifications can be generated only on objects that are being managed by GemFire. When events are delivered, the related data objects can be fetched at memory speeds. This combination of data management and messaging provides a unique architecture enabling applications with predictable SLA characteristics in massively distributed environments.

## High Availability and Durability Through Memory-Based Replication

Most messaging systems use some form of disk persistence to provide reliability. If a primary messaging server goes down, the secondary reads from disk and delivers the rest of the messages. GemFire instead uses a memory-based, highly available first-in, first out (FIFO) queue implementation whereby the queue is replicated to the memory in at least one other node in the distributed system. This is far more efficient for overall throughput and achieves the lowest possible latency for asynchronous message delivery.

Each subscribing client's events are managed in a server-side queue that is replicated to at least one other server node. Often, this is the node where the cache is also replicated, so the events delivered into the queue are merely references to cached objects. A pool of dispatcher threads is continuously trying to send the events to the client as quickly as possible. Keeping the queues separate enables different clients to operate at different speeds, and no single slow consumer can throttle the event-propagation rate of the other consumers.

GemFire provides the option for durable subscriptions that keep the events alive and continues to accumulate events, even if the client disconnects temporarily (normally or abnormally). Options also allow for the conflation for the queues so that only the latest update for each key is sent to a slowly consuming client. A client session and its event queue will be dropped if the consumer is so slow that the queue reaches a configurable maximum size threshold.

# Performance Benchmark

One of the goals of the GemFire design is to achieve near linear increases in throughput as the numbers of concurrent clients and servers managing data increase. Below, we present a throughput benchmark demonstrating the raw throughput and level of scalability that can be achieved using commodity hardware.

All tests used the GemFire Enterprise 6.0 release and two blade centers—one for storing the data with redundancy in memory and the second for simulating a large number of concurrent clients. All blades were multicore commodity machines. Each blade within each of the blade centers has gigabit connectivity to the other, but the network switch connecting the two blade centers is limited to x Gb/sec.

Specific details are available from VMware.

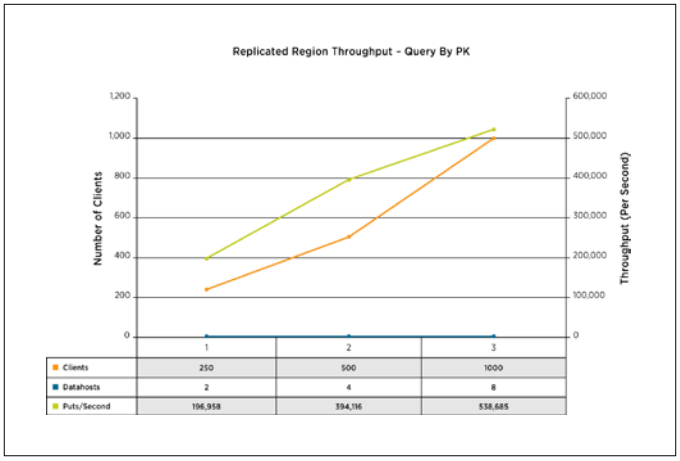



**Figure 8.** Replicated-Region Throughput

# Replicated-Region Query Test

In the replication-region tests, the number of servers used for managing data is doubled from two to four to eight. The number of physical nodes used to simulate the clients are limited to five. Keys are `longs`, and the value is an object about 1KB in size.

Each client application is a simple Java process that starts 18 concurrent threads that query using the primary key or write objects as quickly as they possibly can. The test starts with a single server hosting data, with a single client JVM accessing the data and then progresses by increasing both the number of clients and the number of servers in proportion.

Each client does some amount of warmup and then fetches the value stored in a data region using a randomly generated key. There are 18 concurrent threads active in any client JVM. The X-axis in the Figure 8 shows the increase in the number of replicated servers (data hosts), the Y-axis on the left shows the number of concurrent client threads (18 per JVM) and the Y-axis on the right shows the aggregate throughput observed across all the data hosts.

The results (see Figure 8) show that the aggregate throughput nearly doubles when the client threads count is doubled along with the available servers to access the data from. With 1,000 concurrent client threads, the throughput doesn't double, because load in the client hosts reaches a saturation point with 200 threads competing for CPU (150,000 context switches per second).

# Partitioned Region Query Test

The partitioned-region tests used 12 physical data hosts and 9 physical hosts for clients. Each client virtual machine had two threads and nine virtual machines per host. The test was similar to the replicated-region test, except the number of clients used was lower, and the object size was about 70 bytes.

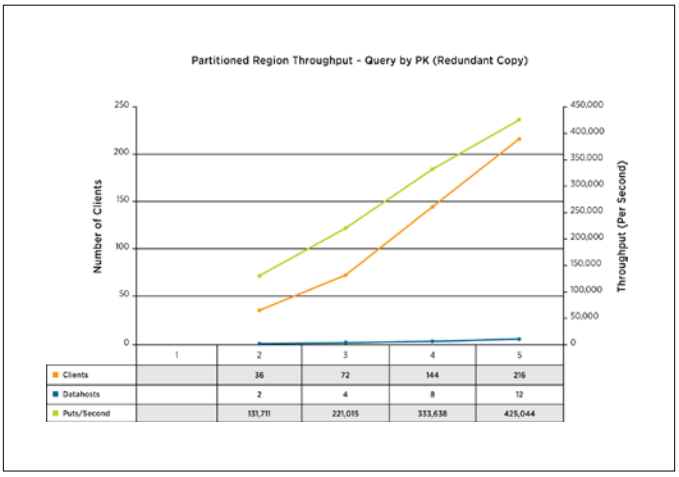



**Figure 9.** Partitioned-Region Throughput (No Redundancy)

Similarly to the replicated-region test, observed throughput nearly doubles when capacity is doubled but is bounded by the available CPU and network constraints at higher loads (see Figure 9).

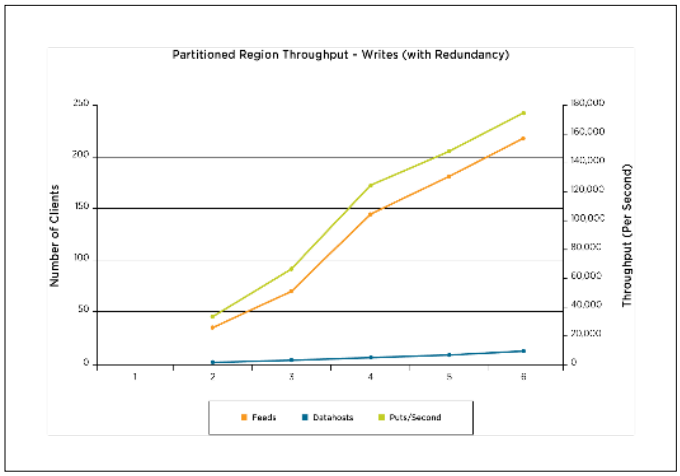



**Figure 10.** Partitioned-Region Throughput (Redundant Copy)

With a redundant copy for any key, client access patterns have a better chance of being load-balanced uniformly, resulting in a more linear increase in throughput with increasing load and capacity (see Figure 10).

# Partitioned-Region Write Test

Writes, in general, can be more expensive than reads with the increased garbage-collection activity. Note that in this test most of the writes update existing entries, creating garbage that needs to be collected (see Figure 11).

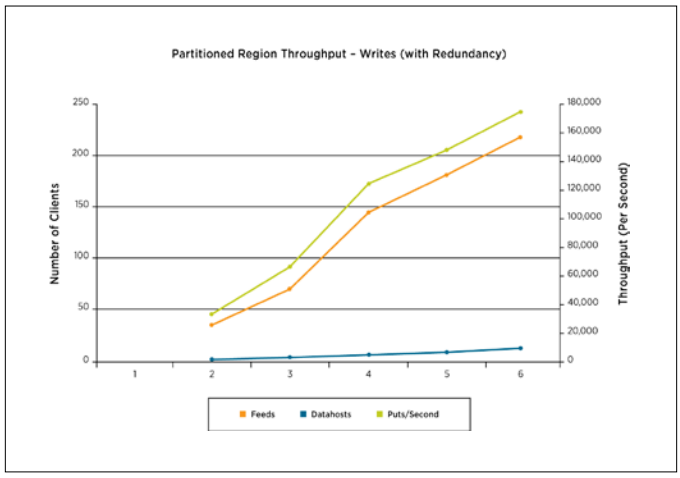



**Figure 11.** Partitioned-Region Throughput—Writes (No Redundancy)

Redundancy provides higher resiliency but comes at the cost of reduced throughput, because each update must be synchronously applied to two member nodes before the write completes (see Figure 12).

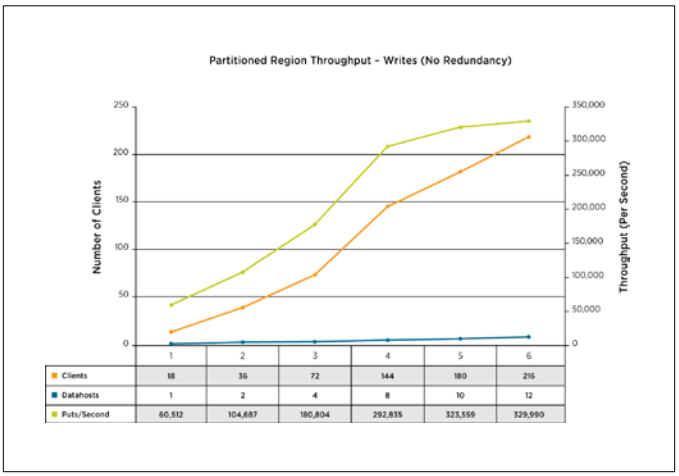



**Figure 12.** Partitioned-Region Throughput—Writes (with Redundancy)

# Conclusion

The features of general-purpose OLTP database systems were developed to support transaction processing in the 1970s and 1980s, when an OLTP database was many times larger than main memory—and when the computers that ran such databases cost hundreds of thousands to millions of dollars[5]. Today, modern processors are very fast, memory is abundant, cheap commodity servers and networks are much more reliable, and clusters with hundreds of gigabytes in total memory are not uncommon. Demand spikes are much more unpredictable—yet customer expectations with respect to availability and predictable latency (expectations raised by engines such as Google) are very high. GemFire presents an alternative, memory-oriented strategy that brings the key semantics of relational databases to commodity cluster environments, relaxing some of the strict consistency requirements in favor of higher performance and scalability. It offers a single platform that serves like a database (or a cache) and also like a reliable publish-and-subscribe system, enabling clients to express complex interest in data through queries.

# References

[1] M. Stonebraker and U. Cetintemel. One Size Fits All: An Idea Whose Time Has Come and Gone. In ICDE '05, pp. 2–11, 2005.

[2] http://highscalability.com/paper-optimistic-replication

[3] Matt Welsh, David Culler and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. Computer Science Division, University of California, Berkeley.

[4] A.-P. Liedes and A. Wolski. A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases. In ICDE '06, p. 99, 2006.

[5] Stavros Harizopoulos, Michael Stonebraker, Samuel Madden and Daniel J. Abadi. OLTP Through the Looking Glass, and What We Found There. Massachusetts Institute of Technology Cambridge, MA.

**VMware, Inc.** 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

 Item No: VMW-WP-vFBRC-GemFR-USLET-105 08/12