In-Memory Data Grids

IMDGs store data in memory as key-value pairs across multiple nodes where the keys and values can be any business object or application data in serialized form. This supports schema-less data storage through storage of semi/unstructured data. Data access is typically provided via APIs. The symbol used to depict an IMDG is shown in Figure 7.17.

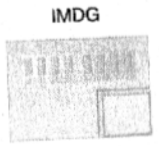

Figure 7.17
The symbol used to represent an IMDG.

In Figure 7.18:

1. An image (a), XML data (b) and a customer object (c) are first serialized using a serialization engine.
2. They are then stored as key-value pairs in an IMDG.
3. A client requests the customer object via its key.
4. The value is then returned by the IMDG in serialized form.
5. The client then utilizes a serialization engine to deserialize the value to obtain the customer object...
6. ... in order to manipulate the customer object.

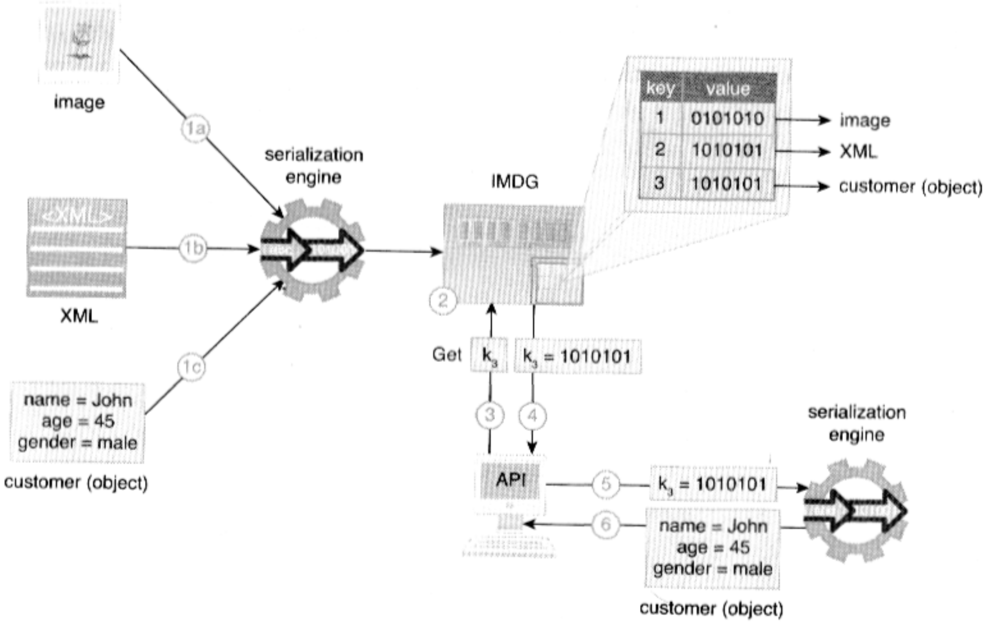

Figure 7.18
An IMDG storage device.

Nodes in IMDGs keep themselves synchronized and collectively provide high availability, fault tolerance and consistency. In comparison to NoSQL's eventual consistency approach, IMDGs support immediate consistency.

As compared to relational IMDBs (discussed under IMDB), IMDGs provide faster data access as IMDGs store non-relational data as objects. Hence, unlike relational IMDBs, object-to-relational mapping is not required and clients can work directly with the domain specific objects.

IMDGs scale horizontally by implementing data partitioning and data replication and further support reliability by replicating data to at least one extra node. In case of a machine failure, IMDGs automatically re-create lost copies of data from replicas as part of the recovery process.

IMDGs are heavily used for realtime analytics because they support Complex Event Processing (CEP) via the publish-subscribe messaging model. This is achieved through a feature called *continuous querying*, also known as active querying, where a filter for event(s) of interest is registered with the IMDG. The IMDG then continuously evaluates the filter and whenever the filter is satisfied as a result of insert/update/delete operations, subscribing clients are informed (Figure 7.19). Notifications are sent asynchronously as change events, such as *added*, *removed* and *updated* events, with information about key-value pairs, such as *old* and *new* values.

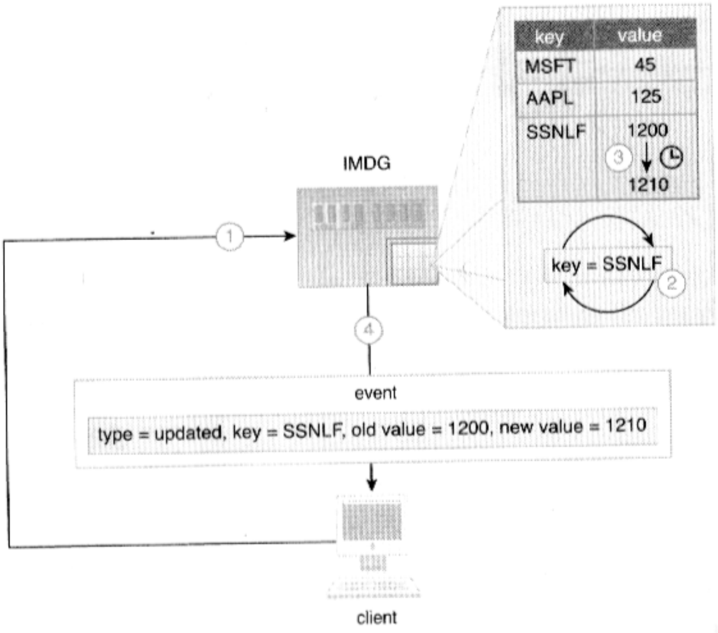

Figure 7.19
An IMDG stores stock prices where the key is the stock symbol, and the value is the stock price (shown as text for readability). A client issues a continuous query (key=SSNLF) (1) which is registered in the IMDG (2). When the stock price for SSNLF stock changes (3), an updated event is sent to the subscribing client that contains various details about the event (4).

From a functionality point of view, an IMDG is akin to a distributed cache as both provide memory-based access to frequently accessed data. However, unlike a distributed cache, an IMDG provides built in support for replication and high availability.

Realtime processing engines can make use of IMDG where high velocity data is stored in the IMDG as it arrives and is processed there before being saved to an on-disk storage device, or data from the on-disk storage device is copied to the IMDG. This makes data processing orders of magnitude faster and further enables data-reuse in case multiple jobs or iterative algorithms are run against the same data. IMDGs may also support in-memory MapReduce that helps to reduce the latency of disk based MapReduce processing, especially when the same job needs to be executed multiple times.

An IMDG can also be deployed within a cloud based environment where it provides a flexible storage medium that can scale out or scale in automatically as the storage demand increases or decreases, as shown in Figure 7.20.

IMDGs can be added to existing Big Data solutions by introducing them between the existing on-disk storage device and the data processing application. However, this introduction generally requires changing the application code to implement the IMDGs API.

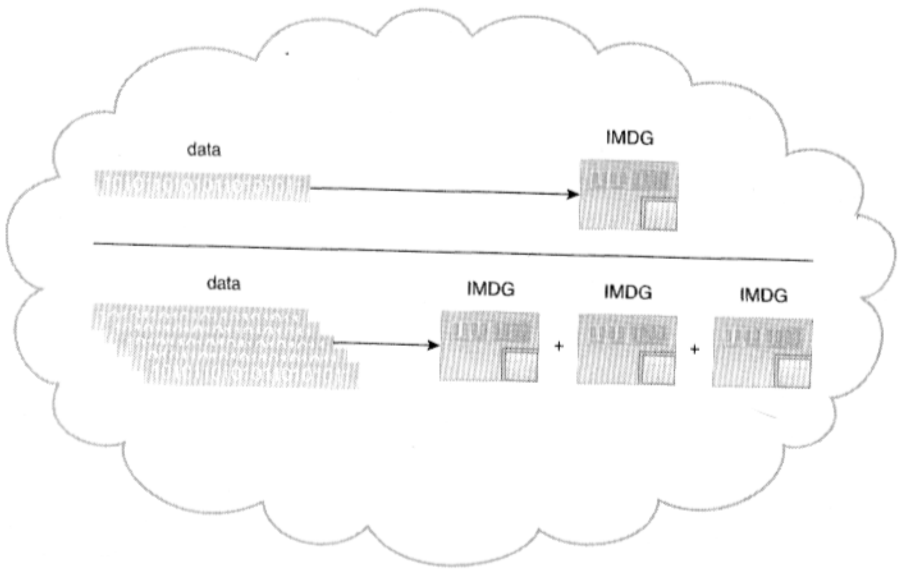

Figure 7.20
An IMDG deployed in a cloud scales out automatically as the demand for data storage increases.

Note that some IMDG implementations may also provide limited or full SQL support.

Examples include In-Memory Data Fabric, Hazelcast and Oracle Coherence.

In a Big Data solution environment, IMDGs are often deployed together with on-disk storage devices that act as the backend storage. This is achieved via the following approaches that can be combined as necessary to support read/write performance, consistency and simplicity requirements:

- read-through
- write-through
- write-behind
- refresh-ahead

Read-through

If a requested value for a key is not found in the IMDG, then it is synchronously read from the backend on-disk storage device, such as a database. Upon a successful read from the backend on-disk storage device, the key-value pair is inserted into the IMDG, and the requested value is returned to the client. Any subsequent requests for the same key are then served by the IMDG directly, instead of the backend storage. Although it is a simple approach, its synchronous nature may introduce read latency. Figure 7.21 is an example of the read-through approach, where Client A tries to read key K3 (1) which does not currently exist in the IMDG. Consequently, it is read from the backend storage (2) and inserted into the IMDG (3) before being sent to Client A (4). A subsequent request for the same key by Client B (5) is then served directly by the IMDG (6).

Write-through

Any write (insert/update/delete) to the IMDG is written synchronously in a transactional manner to the backend on-disk storage device, such as a database. If the write to the backend on-disk storage device fails, the IMDG's update is rolled back. Due to this transactional nature, data consistency is achieved immediately between the two data stores. However, this transactional support is provided at the expense of write latency as any write operation is considered complete only when feedback (write success/failure) from the backend storage is received (Figure 7.22).

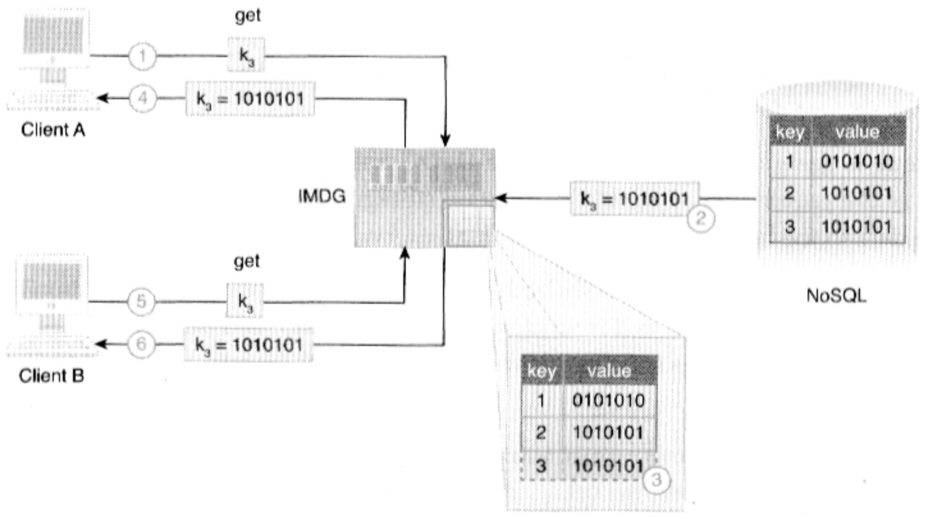

Figure 7.21
An example of using an IMDG with the read-through approach.

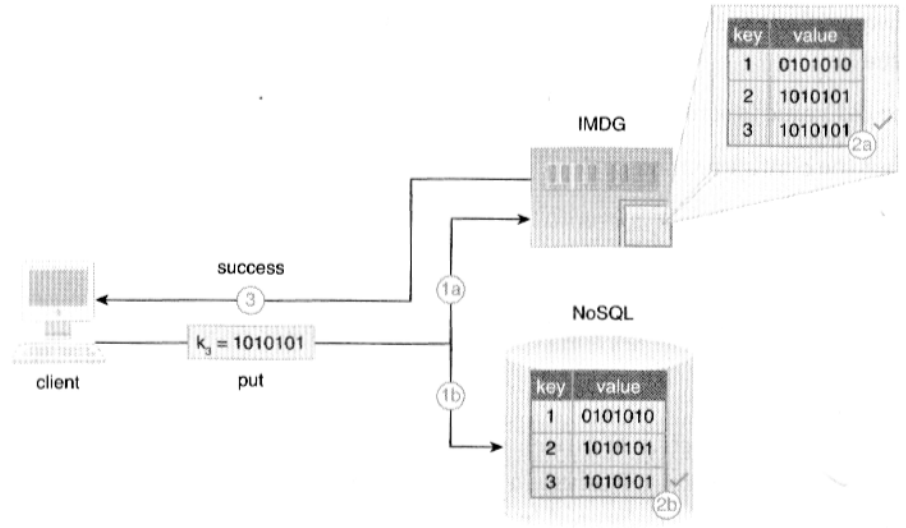

Figure 7.22
A client inserts a new key-value pair (K3,V3) which is inserted into both the IMDG (1a) and the backend storage (1b) in a transactional manner. Upon successful insertion of data into the IMDG (2a) and the backend storage (2b), the client is informed that data has been successfully inserted (3).

Write-behind

Any write to the IMDG is written asynchronously in a batch manner to the backend on-disk storage device, such as a database.

A queue is generally placed between the IMDG and the backend storage for keeping track of the required changes to the backend storage. This queue can be configured to write data to the backend storage at different intervals.

The asynchronous nature increases both write performance (the write operation is considered completed as soon as it is written to the IMDG) and read performance (data can be read from the IMDG as soon as it is written to the IMDG) and scalability/availability in general.

However, the asynchronous nature introduces inconsistency until the backend storage is updated at the specified interval.

In Figure 7.23:

1. Client A updates value of K3, which is updated in the IMDG (a) and is also sent to a queue (b).
2. However, before the backend storage is updated, Client B makes a request for the same key.
3. The old value is sent.
4. After the configured interval…
5. … the backend storage is eventually updated.
6. Client C makes a request for the same key.
7. This time, the updated value is sent.

Refresh-ahead

Refresh-ahead is a proactive approach where any frequently accessed values are automatically, asynchronously refreshed in the IMDG, provided that the value is accessed before its expiry time as configured in the IMDG. If a value is accessed after its expiry time, the value, like in the read-through approach, is synchronously read from the backend storage and updated in the IMDG before being returned to the client.

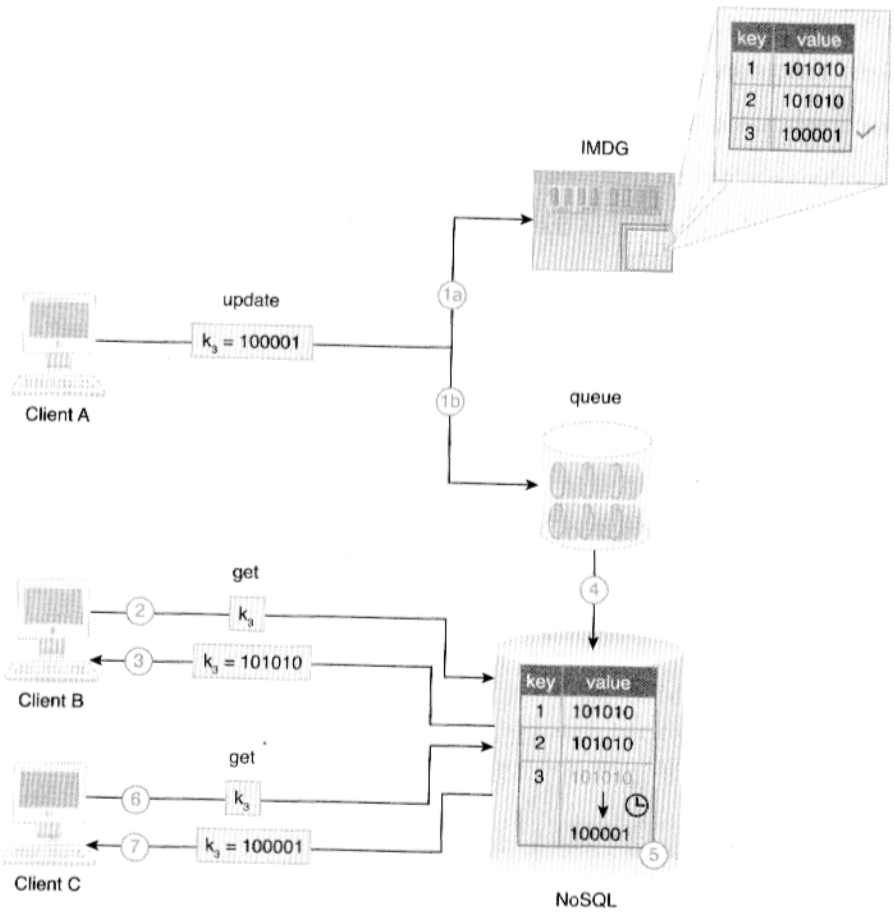

Figure 7.23
An example of the write-behind approach.

Due to its asynchronous and forward-looking nature, this approach helps achieve better read-performance and is especially useful when the same values are accessed frequently or accessed by a number of clients.

Compared to the read-through approach, where a value is served from the IMDG until its expiry, data inconsistency between the IMDG and the backend storage is minimized as values are refreshed before they expire.

In Figure 7.24:

1. Client A requests K3 before its expiry time.
2. The current value is returned from the IMDG.
3. The value is refreshed from the backend storage.
4. The value is then updated in the IMDG asynchronously.
5. After the configured expiry time, the key-value pair is evicted from the IMDG.
6. Now Client B makes a request for K3.
7. As the key does not exist in the IMDG, it is synchronously requested from the backend storage…
8. …and updated.
9. The value is then returned to Client B.

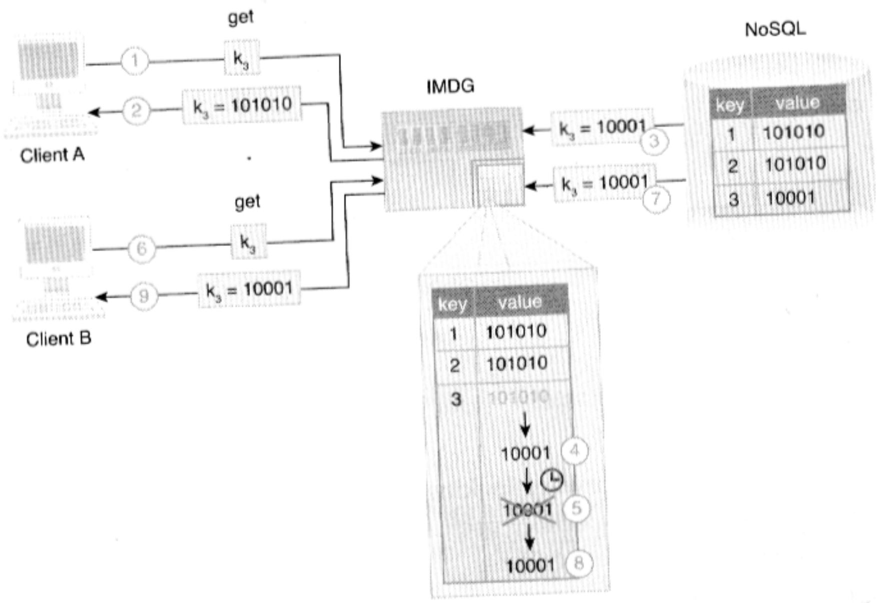

Figure 7.24
An example of an IMDG leveraging the refresh-ahead approach.

An IMDG storage device is appropriate when:

- data needs to be readily accessible in object form with minimal latency
- data being stored is non-relational in nature such as semi-structured and unstructured data
- adding realtime support to an existing Big Data solution currently using on-disk storage
- the existing storage device cannot be replaced but the data access layer can be modified
- scalability is more important than relational storage; although IMDGs are more scalable than IMDBs (IMDBs are functionally complete databases), they do not support relational storage .

Examples of IMDG storage devices include: Hazelcast, Infinispan, Pivotal GemFire and Gigaspaces XAP.