# 8

# FOUNDATION FOR COMPARING DATA GRIDS

In this chapter we will itemize the key points of comparison of various data grid implementations so as to provide the reader with a methodology for selecting the best "tool for the job." Recalling Figure 1.1, the two major areas of comparison are in the "engine" of the data grid itself and the support for the necessary data management features required by the business application. The latter includes support for traditional data management techniques, data management features specific to the data grid, as well as accessibility to the data grid. It is expected that the reader understands the traditional data management features such as those supported by a relational database. The chapters subsequent to this will focus on data management features that are unique to a data grid.

## CORE ENGINE DETERMINES PERFORMANCE AND FLEXIBILITY

Data grid architecture and the associated characteristics can vary widely. With many different options in architecting the data grid, there are two that are most prevalent: (1) "data replication" versus "data distribution" and (2) "centralized" versus "peer-to-peer"-based synchronization management. Each of these architectures provides support for a global or common feature sets as well as unique feature sets. We will also see that there are "policy"-based data management features for the data grid's data distribution/replication and for data synchronization. The policy-based data management features can be supported only if the underlying engines support the mechanics of those features. For example, if the underlying engine only supports data replication, then data management policies involving a distributed data scheme cannot be implemented with that engine. Similarly, the engine must also support the mechanics for all the data grid management policies, for example, "event notification" and not just the management policies for data synchronization and distribution.

The sections that follow highlight the most common data grid architecture and the associated feature sets.

## Replicated versus Distributed

*Replication-based architectures* rely on a duplicating cache across engines that guarantees that any data modified in one cache are shared across all members. This allows for total cache synchronization regardless of where the data modifications occurred. The common features of replication-based architecture are a high degree of reliability and data integrity since the data resides on many nodes but at the cost of scaling and performance due to data concurrency across the nodes of the data grid. The replication schema is typically achieved through levels of reliability built on top of the multicast or broadcast protocols.

The *distribution-based architecture* of the data grid, on the other hand, tends to share data on a peer-to-peer (P2P)-oriented nature. The advantage of such architecture over the replicated architecture is greater scalability since all the data are not replicated across all nodes of the data grid. One way to visualize this is to compare data distribution to how a RAID (redundant array of inexpensive disks) device "stripes" data across the disk array. Data distribution involves striping a piece of data across a number of the physical nodes of the data grid. These nodes are a subset of the total nodes in the data grid and will be considered as peers to each other for data update, distribution, and access. As does a RAID device, this method of data distribution yields an upper bound of available data grid storage capacity solely as a limit of the number of physical nodes to the data grid. (Conversely, in a replicated engine, the upper bound of data grid storage capacity is limited to the physical node in the data grid with the least physical capacity.)

The disadvantage of such architecture is data reliability and data integrity. Even though the distribution-based architectures does not replicate the data completely, some degree of replication is achieved, thus yielding a level of resilience to failure. If a piece of data is distributed or "stripped" across 10 of the 100 nodes of the data grid, for example, then that data are resilient if at least one of those nodes remains operational. Should all 10 of those nodes fail while the other 90 remain operational, then that piece of data is lost. The ratio of nodes to stripe a piece of data across is managed by the data management policies described below. Adjusting this ratio of nodes to stripe or "replicate" a piece of data across to the total number of nodes in the data grid will determine the level of resilience of the data grid for that piece of data.

### Centralized versus Peer-to-Peer Synchronization

The implementation of an engine of a data grid will follow one of two general architectures (distributed or replicated) as it relates to how it physically manages data integrity across the nodes. There is a centralized manager for data integrity among the nodes of a data grid as well as for synchronizing data in and out of the data grid with external data sources. The second method employs a decentralized manager for data integrity. In this approach, only those nodes of a data grid that have a piece of data stored locally in it will be involved in managing the integrity of that data among themselves. Drawing from the example in distribution engines, if a distributed data engine also supports a P2P synchronization implementation, then only those 10 nodes on which a piece of data resides will be involved in any transactional operation for that piece of data. The other 90 nodes are free to go about supporting other usage requests made on the data grid.

## ACCESS TO THE DATA GRID

Access to the data grid must support methods similar to those found in traditional data management tools. There needs to be a programmatic API set as well as some method to query the data grid through a string-based query, and finally there needs to be a management interface for the data grid as a system. These topics are addressed in more detail in Chapter 20. However, when comparing which data grid is best suited for your purposes, you must consider the following:

- The support of a programmatic API in the languages (Java, C++, C#, etc.) in which the business applications are implemented. The quality and flexibility of those APIs are important points of comparison.
- For a string-based access method, one must consider how the data grid will be integrated into the environment and which class of applications are required to leverage it. For example, if the business application is Web-oriented, then support for an XML-based query is more useful than a standard SQL-type access method.

### User-Level APIs

The data grids offer a variety of application-level APIs depending on the vendor and the type of architecture. All vendors and architectures support a concept of a data-grid-aware structure; for example, if the data grid engine supports an "object-oriented" API set, then it will have the concept of a collectable object and a collection object such as a bag or list. In addition, the APIs support rudimentary querying, updating, and retrieval of data in and out of the collections of the data grid. For operational functions, the APIs support a set of both traditional (startup, shutdown, user and user entitlements, etc.) and data-grid-specific data management features such as synchronization, distribution, replication, and event notification.

### String-Based Interfaces

As straightforward as a programmatic API interfaces are, string-based queries of a data grid are less clearly defined. In Chapter 20, an argument is posed regarding which is the best method to use for string-based queries as there currently is no standard similar to SQL in the relational data models. Is a SQL or SQL-like interface best since the majority of developers are familiar with it as a tool, as well as its properties and syntax? Or is a more Web Services–like interfaces best, such as an XML-based interface? The answers to these questions have yet to find a consensus among technology specialists and the marketplace in general.

## SUPPORT FOR TRADITIONAL DATA MANAGEMENT FEATURES

In order for the APIs to support the traditional data management features, they need to provide capabilities that include querying, indexing, administration, and replication. These features typically need to be "loosely" available within data grid products.

The *querying* feature is typically supported through a proprietary XML interface that has some of the features available in ANSI-SQL. Since a data query is typically unstructured or hierarchical data in nature, traditional ANSI-SQL specifications are not commonly used.

*Indexing* and reorganization is typically loosely supported in data grids through both garbage collection and graph transformations. The garbage collection and graphic transformation allow unstructured data to be formatted, reorganized, and structured to the requirements of the receiving application.

*Data grid administration* is currently supported through command-line interfaces. Most administration is achieved via modifications to configuration files.

*Replication* is supported through processes that allow data replication across engines, database servers, and nodes.

## SUPPORT FOR DATA MANAGEMENT FEATURES SPECIFIC TO GRID COMPUTING

With new technology come new features that are required and the associated data management specific to that technology. This, too, is also true for the many unique aspects of data management in a grid environment. Services specific to data management in grid environments include data regionalization, data synchronization policy, transactional data policy, coordination of task scheduling to data locality, event notification policy, and data load policy, which are discussed in the sections that follow.

- *Data Regionalization.* Regionalization of data within a grid is a key performance and management feature that is seldom available within other infrastructures. A data region within a grid is an organization of data that spans machines and potentially geographies and caters to the needs of the users of that region. “Region” is the highest level of constructing the data in the data grid. Region drives the aggregation of the data and the policy instructions regarding the data. For example, a data region is analogous to the “database” in the relational model. It contains other structures of data or data schema specific to a line of business. The management policies of the region will describe the behavior of the region in order for it to best meet the requirements of the line of business or “business services” that it supports. The data-grid-specific management policies are listed below.

- *Data Synchronization*. Dynamic data synchronization is performed per the defined management policy. The “data region” definition encompasses the data synchronization features and enables the control of different types of consistency policies across the data grid. Data synchronization typically falls into two spectra: strong synchronization and weak synchronization. Strong synchronization policies enforce a tight replication of “like” patterns of data among the nodes of the data region as well as strictly enforcing the replication policies among the nodes of the data region. The strong synchronization mechanism is used when low latency and high consistency are required from the data grid at the cost of scalability and flexibility. Weak synchronization policies, on the other hand, enable data to be synchronized on an “as needed” basis and sometimes not at all. The weak synchronization mechanism allows for less data consistency but for higher scalability and flexibility.
- *Transactional Data Synchronization*. Transactional data synchronization is very important even in the data grid since the ability to recover, commit, roll back, and the like are important to data integrity. Transactional data policies fall into basic categories within a data grid: *optimistic* and *pessimistic*. One of the main drivers of the transactionality with external data sources is the transactional features and semantics supported by that external source. The data grid must support a level of transactionality equivalent to that external source in order to maintain a quality of service as required by the line of business that the data region (data grid) supports. *Optimistic* transactions are typically implemented with little or no locking and coordination; they are transactions only in the sense that in a possible conflict situation, an exception is thrown and the user is notified. *Pessimistic* transactions are typically multiphase with the proper locking, unlocking, commit, and rollback that are typical for transactional integrity. The mechanism used is that of XA, which is associated with the destination, and all transactional management (e.g., all locking) is done through that destination.
- *Data Locality* (*Data Distribution*). The ability to locate and group data depending on data usage is essential to the data grid from a performance perspective. Data locality thus can be defined as the clustering of data depending on usage. This feature, which is specific to the data grid, enables the architectures to scale significantly better than did previous technologies architectures. The data

grid implements data locality through two sets of synergistic architectures: (1) data within a data grid are associated with a locality that pinpoints the exact resource that owns a primary copy of the data and its neighboring topology (this information is provided through the APIs to the architecture using the data grid in order to propagate and distribute work to the particular resource) and (2) metrics of data usage improve the availability of the data in the data grid. Data are monitored through a set of metrics, and individual data blocks can migrate from resource to resource depending on the usage patterns and history of use.

- *Event Notification.* The data grid supports a variety of push-based and pull-based event notification policies, depending on the product and its associated architecture. In general, event notification is supported around a particular data type, which can be by region, collection, or individual data object, as well as a particular transaction type. APIs for event notification involve either callback handlers or queues, and typically require the handler to be in the same data region as the event source.
- *Data Load/Save.* Data loading and data saving is an essential part of the data grid. Without capability of loading and saving data, any import and export of data would be haphazard and require custom code development. The data grid supports the "load and save" feature through a variety of mechanisms, some coupled with pessimistic transactional support and some without. Transactional load/save mechanisms work by allowing users to specify a block of code (or in the vocabulary of enterprise information integration, a data adapter) that would map inbound data to the transaction source and query (load) or commit (save) within the context of a data transaction. Transactional mechanisms are activated at demand time and will complete before any access or update to data within the grid is allowed. Nontransactional load/save allows users to specify individual procedures (adapters) to map data into and out of the data grid. These procedures are executed on data demand but are not required to complete successfully before any access or updates are permitted.