9

DATA REGIONALIZATION

A scene as serene and innocuous as clouds rolling across a clear, crisp blue sky offers a useful analogy for regarding the concept of data regionalization. Like the clouds crossing the sky, data regions float across the data grid plane (DGP). Like the droplets of water held within the clouds, the data within a data region can be gathered from a variety of sources and are now united to form this region of ever-changing size and shape as it traverses across the data grid plane. And, as with the effect of atmospheric conditions on our vision of a clear blue sky, the data region continually adjusts to changing external factors, such as business need, usage demand, overall data size, and performance requirements, so as not to allow its data to fall to Earth.

Data region management policies for distribution, synchronization, and other functions affect the region's size, shape, and traversal in the data grid plane (see Figure 9.1).

Other external forces play a hand. They include hardware, mean time between failures, scheduling and routing of tasks (in the compute grid plane), time of day, and cycling of available resources in the data region's size, shape, and traversal across the data grid plane. This complex interaction of forces is counterbalanced by the data management policies of the data region, continually adjusting its characteristics to keep itself in an optimal state to meet the supply–demand curves imposed on it.

Leveraging this concept of the data region as condensation and the data grid plane as the sky allows us to bridge to the physical implementation with the right tools of analysis and mathematical modeling. All these affect data distribution

Distributed Data Management for Grid Computing, by Michael Di Stefano

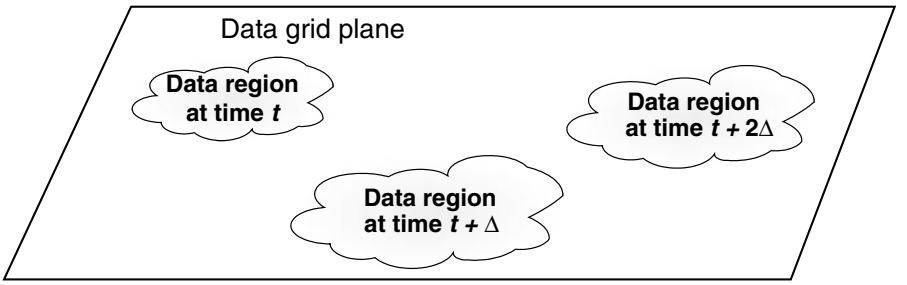

Figure 9.1. Data regions in the data grid plane.

and data synchronization policies, as well as other concrete data management policies involved in the proper access and management of a data region in the highly distributed environment of grid computing.

WHAT ARE DATA REGIONS?

Traditional client/server data architectures defined a concept of multiply siloed databases, or a data warehouse that contained the entire set of information required to run a particular business. Applications built around this concept promulgated this notion and themselves became associated with a set of information and a particular business. As businesses grew, many different, and sometimes competing silos were created to deal with different aspects of the business. In finance, the back office and the front office tended to have similar information, but could rarely share. Data grid architectures are designed to specifically decouple the location of particular data from the resources that use them. In order to accomplish this, the concept of a data region needs to be defined. Once defined, the data region can be managed through data management policies.

Data regions are defined as a logical organization of virtual resources that provide the storage necessary to house data. That storage and the virtual resource that provides it are typically unspecified in terms of service level and locale. In addition to virtual resources, regions have a set of management policies associated with them. The data contained within a data region represent a logical grouping independent of source.

DATA REGIONS IN TRADITIONAL TERMS

Data regions are similar to databases in traditional terms. Figure 9.2 illustrates this relationship.

It is always best to establish a baseline that is grounded in concepts that most of us are familiar with, a common knowledge that we can use to visualize and build on when learning new concepts. In introducing data regions within a data grid, we

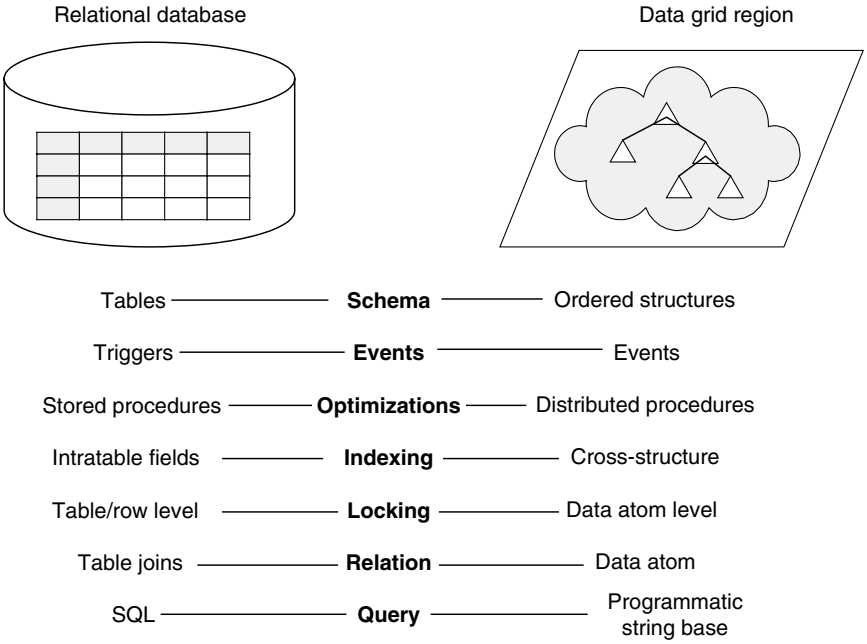

Figure 9.2. Data regions in traditional terms.

will use relational data management as the baseline. This analogy is as functional as it is visual; the traditional elements of data management need to be maintained and expanded on when entering into a data grid environment:

- *Logical Data Groupings*. Data are logically grouped, typically aligned with a business vertical. In the relational data management realm, this grouping is by a database. In the case of the data grid, this grouping is a data region.
- *Schema*. The schema relates to the way data are organized within the database: a definite structure organized in a logical grouping and aligned with the business and data applications where data storage, retrieval, and updates are applied to the database. In a relational database, the fundamental organization for the schema is a table, a two-dimensional matrix of rows and columns. Typically, databases contain many tables. [*Note*: Within a data region the choice of data structure goes beyond the two-dimensional rows and columns that constitute a table.] The fundamental structural elements for data schema within a data region are dependent on the implementation of the data grid. It is important to make clear that the data grid implementation is independent of the underlying engine (i.e., file system or distributed cache; see Figure 1.1). The engine defines the data grid's ability to fulfill other QoS levels, but imposes no limit on the kinds of data schema structures that can be defined. Again referring back to Figure 1.1, this is purely a function of

the data management layer and its implementation. Structures are dependent only on the implementation of the data management layer and its support for *N*-dimensional structures, including arrays, tables, matrices, and trees.

- *Events*. An event is an occurrence or happening at a single point in time and space. In computer science, events trigger a change of state for an object or system. Typically, a system's state is defined by its data attributes. Programmatic paradigms are based on the concept of events and form the basis for event-based processing that drive straight-through processing (STP). In relation to data management, an event triggers a change in state of an array, table, database, or data region. Programmatic "event handlers" or "triggers" are registered for a specific event and are invoked when that event occurs. A common use of triggers in a relational database is to maintain referential data integrity among the tables of the database. Triggers tie together or are set to a specific event, such as the insertion or deletion of data to a table. When these events occur, the triggers are programmed to make the required changes, such as insertions, deletes, or updates to the other structures in the database or data region to ensure data integrity. Events caused by triggers can in turn set other triggers into motion; thus a chaining effect can take place. Data regions can extend event handling by allowing systems external to the region to "register" interest in data region events. The external systems can either be notified directly by the data region or invoke an event handler registered by the external system when an event, with its associated triggers, occurs.
- *Optimizations*. Data management systems support various optimization tools for data access and update. One of the more common methods is the precompiling of queries. In the case of the relational databases, SQL statements are precompiled into procedures that are given well-known handles so that user programs can invoke them directly. For example, when an application queries the database, a certain amount of processing must be done by the database to transform the SQL into executable code. If the query is frequently used by one or more applications, it can be optimized within the database to eliminate many of the preprocessing steps. This optimization improves the performance of the database queries. In some implementations of relational databases, these are called "stored procedures." Similarly, procedures can be maintained within the data grids and their associated data regions. However, given the nature of the implementation of a data grid, the exact meaning of a stored procedure can vary. For example, a data grid that is based on a distributed cache can distribute the precompiled procedures across all the nodes of the data region. When invoked, the data grid can execute the "distributed stored procedure" in parallel, each node processing it against its own set of data within that region.
- *Indexing*. Indexing is a way to gain faster access to atoms of data within the database or data region. In the relational model, a column, or groups of columns, within a table can be indexed in ascending or descending order. The cost of creating and maintaining indexes is extra overhead for the data management system. The amount of overhead is dependent on the design

and implementation of the data management system. In the case of a relational database, an index is an extra data structure that needs to be maintained each time an indexed table is updated or changed. Indexing within a data region is conceptually the same as that within the relational model; however, data grids can support additional data structures, including arrays, tables, matrices, and trees. Therefore, the exact implementation and benefit of indexing within the data grids can vary greatly depending on specific data grid designs and implementations.

- *Locking*. For a multiuser system, maintaining data integrity is essential, and therefore locking is required. Locking data atoms assists in maintaining data integrity when multiple users have permission to update and change data within a database or data region. When updates, inserts, or deletions are performed on a data set, it may be necessary to block others from accessing that data until the data modifications are complete and committed. This process of blocking access to data from other users when they are being changed is called "locking." Typically, the updating application "acquires" the lock from the database before starting and then "releases" it when the operation is completed. In a relational database, the level of data atom locking can be at table level within the database or at a much finer level of granularity: the row within a table. When row-level locking is employed, other users can acquire locks on different rows of the same table without being blocked. Thus users are not interfering with each other. On the other hand, locking within a data region will take place at the data atom level. Therefore, the finest atom of locking of a data structure is dependent on the type of data structures supported by the data grid. When structures in a relational database are mirrored in a data grid, the data grid must support the same locking features and same level of granularity as those in a relational database. However, for more complex structures, the granularity of data atom locking is dependent on the data grid implementation.
- *Relation*. How are the various data structures and data atoms related to each other? In the world of relational databases, where the fundamental schema is a table—a two-dimensional structure of rows and columns—additional dimensions can be created by joining tables based on common fields, allowing relationships to be established between two or more tables. Within the data region, relations between data structures can be as fine as the most basic data atom joining a heterogeneous set of data structures offering more granular flexibility. For example, trees can be joined to arrays that can be joined to matrices. Depending on the implementations of the data grid, a single data atom may be a member of multiple structures, thus providing the relation between the structures.
- *Query*. Relational databases have standardized on structured query language (SQL), a text-based query language. Within the data grids and data regions there are fundamentally two ways to query data: (1) a string-based query language similar to SQL and (2) a programmatic querying or filtering.

Programmatic querying is a higher-level language such as C++ or Java, where the query is done via a set of programmatic APIs. Early implementations of data grids will support the programmatic query interface first. String-based query languages for data grids are an area that today requires industry standardization on exactly what the language syntax needs to be. This is primarily because data grids can support a wide range of data structures and the most optimal query language may not be SQL. Rather, it may be more of an XML-like language, or a hybrid of SQL, Object SQL, and based on other theories. Such standards are necessary, especially if the data grid is to succeed in the commercial arena.

Data grid queries must support two types of functions: queries into user-defined data structures and queries into the operational structures resulting from data management. For data grids, the operational data include most of the same administrative data sets for users, entitlements, and logging. However, additional information is needed for data-grid-specific data management features. The most obvious is the support for data affinity, which is discussed later in this book. Statistical information on each data atom is required to support data affinity, including

- The physical data location of each data atom and all its replicas
- Access patterns
- Movement patterns

Therefore, just as with relational database engines, administrative query support is an integral part of the data grid system.

DATA MANAGEMENT IN A DATA GRID

As with many other solutions, data management is very important in the data grid; the ability to define data management policies specific to each data grid region is very powerful at implementation:

- *Data Grid Resources*. Many components or resources are required for the data grid. A data grid resource includes the processors and storage associated with a data grid (the nodes constituting the grid), and the data grid "daemon" that monitors and manages the physical nodes of the data grid. The data grid daemon tracks and records "metered information" describing the state of each node of the data grid. Metered information includes details about the memory layout, processor, and size of the CPUs of the grid's nodes, local node transaction/storage, and load sources. Together this information provides the data grid normalized information that it uses to determine the proper amount of resources required to efficiently and effectively service usage demand.

- *Management Policies*. Flexible data management policies are required. The management policies are applied at region level, enabling a region to behave similarly to a relational database instance. Management policies for the regions include data distribution/data replication, synchronization, and load/store. Each of these policies addresses a particular behavior of the region. Data management policies include (1) data distribution, (2) data replication, (3) synchronization, (4) data load and store, and (5) event notification. The interrelationships between these policies are discussed in the sections that follow.

For each of the data management policies listed above, we will express the key parameters that define them. These expressions are not intended as a complete expression of the respective policies, but simply as a basis on which to build insight into their roles and interactions with each other and affect on the system as a whole. As in Chapter 5, section entitled "Application Characteristics for Grid," the expression for an application is in equation form. For similar reasons—namely, for the purpose of quickly developing multidimensional relationships among the parameters and policies themselves—we will follow that notation here.

Data Distribution Policy

Regions contain collections of resources that manage data. Data associated with a particular region can be distributed or replicated through a number of methods. Distribution of data takes individual data "atoms," associating them with a particular resource and resource ownership of that data atom. Distributions of data "atoms" include simple techniques such as round-robin, mathematical models (e.g., Gaussian and Poisson distribution), and dynamic schemes based on real-time system behaviors. Each of these distributions results in a specific data topology:

- *Round-Robin Distribution*. Round-robin distribution is a simplistic distribution scheme that distributes data "atoms" in a sequential mechanism. It does not guarantee a particular distribution except if all the resources have exactly the same capacity.
- *Gaussian Distribution*. Gaussian or normal distribution takes as its parameters a central machine or machines, a standard deviation, and a set of distances, and attempts to distribute most of the data in close proximity to the central machine.
- *Random* (*White Noise*) *Distribution*. Data atoms are randomly distributed across the data region.
- *Poisson Distribution*. Poisson or jump distribution uses parameters similar to those of Gaussian distribution and in addition takes into account the probability of jumps. It attempts to distribute the data within the proximity of a central machine, but also adds the possibility of data "jumping" away from the central machine on the basis of some probability algorithm.

1. *Dynamic-Data Movement Pattern Analysis*. For efficiency reasons, a large part of data distribution policy is aimed at minimizing data movement within a data grid. Replication of a data atom across a data region to the physical nodes where the data are accessed most often minimizes network traffic. There are numerous methods for achieving this goal, one of which is to monitor data movement within a data region and continually evaluate and redistribute the data according to data access and usage patterns. This implies a continual feedback control loop that evaluates the following:

- *Input*—data location, data request points, data movement, distance that data travel on the network, frequency of data requests, and other parameters
- *Logic*—an algorithm that best estimates the placement of physical data locality to minimize or eliminate future data movement within the data region
- *Control Commands*—the ability to manage data movement in the data grid to manually tune system performance

The "control commands" can be manual, involving people analyzing "macroscale" data patterns and manually redistributing data over long periods of time. Alternatively, the process can be automated via a programmatic analytical process causing "microscale" data distributions and redistributions over short periods of time. Finally, a combination of both automated (microscale) and manual (macroscale) data distributions may be used.

2. *Implied Properties of Data Distribution*. A number of implied properties of data distribution are important to system behavior, including the distribution policy and its interactions with other policies, data management features, and the external systems. The implied properties that affect the abovementioned system behaviors are

- *Locality*—the position of the data atom.
- *Manipulation* of the data atom position.
- *Query* of the data atom position.
- *Distribution policy* that includes all data atoms and their replicas.

3. *Locality*. The data distribution policy implies an intimate knowledge of the exact location of each data atom (including its replicas) in the data region. This is key to the successful implementation of any data distribution policy. The data distribution policy determines where each of the data atoms is to be physically located in the data region. Knowledge of exactly where a data atom resides will result in an efficient use of the network resource, since data atom movement will be minimized during the operation of or access to data in the region by the applications or services it supports. This concept is known as *data affinity*, which will be addressed in the chapters that follow.

4. *Inclusion of Replicas.* The distribution of data in a data region must also include all the data atoms and the replicas as determined by the data replication

policy. Only in this way can maximum data affinity and data grid resilience (data grid high availability) be achieved.

5. *Location Manipulation.* Data distribution policy inherently implies that implementation will provide the ability to manipulate the exact physical location of each data atom in the data region.

6. *Query of Locality.* To implement the data management policies of a data grid and support a full range of data management features, data affinity, for example, all the administrative attributes of a data atom must be known and can be managed. These attributes include physical location in the data region as well as history of data movement (location and time of access).

Data Distribution Policy Expression. The data distribution expression defines the key parameters for the distribution of data atoms within a data region identified in the formula

$$DataDistributionPolicy = DDP\begin{bmatrix} PolicyName, \\ Region, \\ Scope(), \\ Pattern() \end{bmatrix}$$

where

- *PolicyName* = logical name for this policy. This is the logical name for this instance of a data distribution policy. Since there may be many distribution policies, this name provides a unique identification. Depending on the implementation of the data grid, this name may or may not be unique.
- *Region* = primary region name. This is the primary data region to which this data distribution is applied. A data distribution of identical characteristics [as determined by the *Scope*( ) and *Pattern*( ) attributes] can be applied to other regions in the data grid.
  - *Scope*( ) = *F*(*All*, *List*(*DataAtoms*) = *NULL*). The scope of the data distribution can span the entire data region as indicated by the "all" attribute or apply only to a specific range of data atoms identified in the supplied list. Note that these parameters are mutually exclusive.
  - *Pattern*( ) = *Function*(*Automatic*/*Specified*, *DP*( )), where

$$DistributionPolicy = F\begin{bmatrix} PatternName, \\ DeployPattern(), \\ DataAtom, \\ Currentlocation, \\ NewLocation, \\ Move/Add/Delete \end{bmatrix}$$

This distribution pattern is used to apply to the data atoms identified in the Policy's *Scope*( ). The pattern can be automatic in nature, one that follows a predetermined

mathematical principle such as randomness or Gaussian distribution. In this case, the pattern expression is the *DeployPattern*( ) parameter and the parameters of data atom, including its current and new locations, and operation (*Move*/*Add*/*Delete*) are not required. The second option is to manipulate the exact physical location of a specific data atom manually. In this case, the *DeployPattern*( ) parameter is not required. However, the parameters of the data atom, including its location (current and new), and operation (*Move*/*Add*/*Delete*) are essential.

Data Replication Policy

Within a data region, data atoms are distributed on the basis of policies. The distribution policies can be grounded in mathematical formula or heuristic usage patterns of data movements within the data region through time. The distribution policy determines the physical location where each data atom will be cached within the data region. The data replication policy goes hand in hand with the data distribution policy. Data replication addresses the number of "copies" of a data atom that exist within a data region.

Both the data distribution and the replication policies should be statistically tied to the physical size of the data grid; or, more specifically, to the physical size of the data region within the data grid. The physical size of the data region—for example, in a peer-to-peer topology—is the number of compute nodes assigned to a data region. The number of nodes that can execute the tasks of a service or services supported by the data region determines the physical size of a data region. (*Note*: More sophisticated data regions can be constructed and maintained to span nodes where there is no intersection of task execution capability. However, for this discussion we will consider only the simple case of the intersection of nodes to execute tasks for a single or multiple services.) Therefore, the maximum size of a data region is the entire grid of T nodes. Typically, as the grid grows to support more services, the data region size will be R number of nodes less than or equal to those of T. As R increases in number, a sophisticated model (e.g., statistical, heuristic) for data distribution policy becomes possible and preferable. Also, the data replication policy can not only reflect the size R of the region but also take into account the geographic/network topology that the data region spans. As the data region size R shrinks to a minimum of one or two nodes, the data distribution policy begins to look like the data replication policy.

The combination of data distribution and replication policies characterizes the data region's ability to support a task or service with minimal data movement and thus minimum network traffic with a region; adding in the data synchronization policy, the robustness of the data region to any failures is then defined.

Figure 9.3 shows an example of a modified data atom synchronized with its peers and replicas. This is not to suggest that all data synchronization relationships represent a single master/replica orientation. Various data atom synchronization relationships can be supported as part of the data synchronization policy.

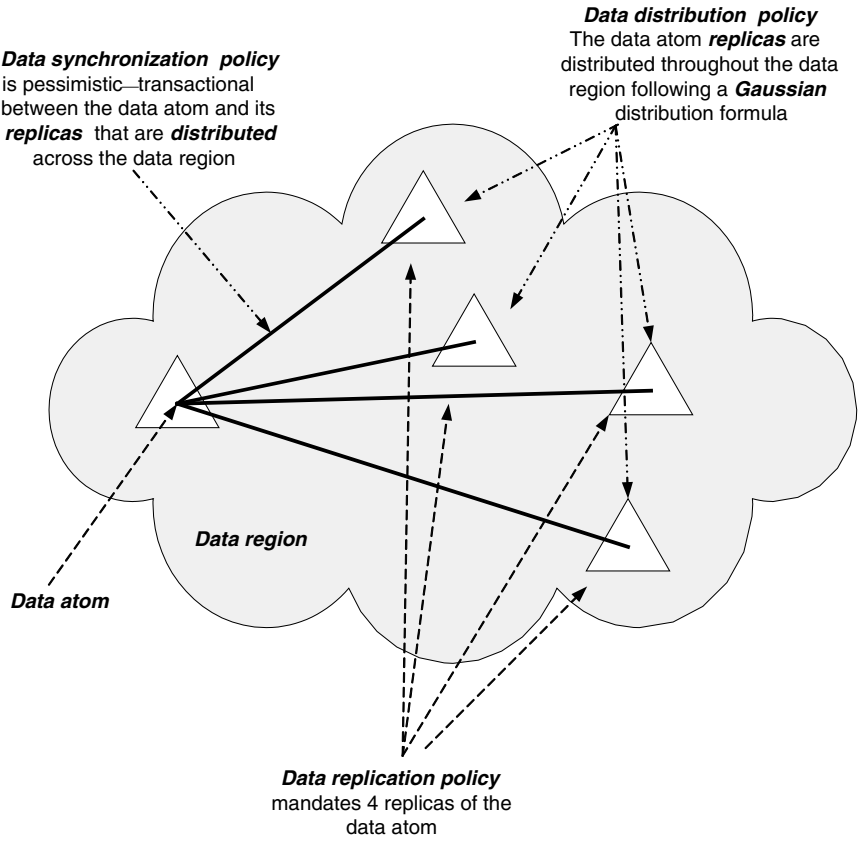

Figure 9.3. Relationship between data replication, distribution, and synchronization policies.

Data Replication Policy Expression. The data replication policy expression defines the key parameters for the replication of data atoms within a data region. The expression can be expressed as

$$DataReplicationPolicy = DRP \begin{bmatrix} PolicyName, \\ Region, \\ Quantity, \\ Scope( ) \end{bmatrix}$$

where *DRP* is the data replication policy. The following parameters influence the policy:

- *PolicyName* = logical name for this policy. This is the logical name for this instance of a data replication policy. Depending on the implementation of the data grid, this name may or may not be unique.

- *Region* = primary region name. This is the primary data region to which this data replication policy is applied. A data replication policy of identical characteristics [as determined by the *Scope*( ) and *Quantity*( ) attributes] can be applied to other regions in the data grid.
- *Quantity* = number of replicas. This is the number of replicas that a data atom will have in the data region.
- *Scope*( ) = *F*(*All*, *List*(*DataAtoms*) = *NULL*). Another function, *Scope*( ) of the data replication policy, can apply to all the data atoms of the entire data region as indicated by the *All* attribute or apply only to a specific range of data atoms identified in the supplied list. Note that these parameters are mutually exclusive.

Synchronization Policy

Synchronized regions ensure that all data "atoms" associated with a particular region are available everywhere. Replication of data "atoms" falls into replicating all or replicating a subset of categories. Total replication assumes that all the data for the region is available and copied everywhere. Partial replication combines a distribution policy, such as a round-robin distribution, with total replication of some data "atoms." Synchronization policy is discussed in more detail in a later chapter.

Load-and-Store Policy

A load-and-store (load/store) policy is needed when data are integrated from data sources external to the application data region. These external sources can include other data regions, legacy systems, databases, middleware (i.e., messaging), and files. The data load policies will define how and when data are to be obtained for the data region, for example, whether the data are to be "preloaded" or "loaded on demand." The data store policies will determine when and how data are to be pushed out of the data region to the appropriate external data destinations, for example, persisting to an external database. Data store policies can be transactional as well as nontransactional. Both load and store policies can also follow a data distribution policy.

One way to think of the data load/store policies is as an interregion synchronization policy. Virtualization of the external data sources and the interaction properties of those data sources allows the application that is running in synchronization to behave similarly to the synchronization policies of the data region. One must take into account whether the external data sources support such policies. A load/store policy is needed when data are integrated from data sources external to the application and its associated data regions. Those external sources that require load and/or store could include

- Other data regions
- Legacy systems

- Databases
- Middleware (i.e., messaging)
- Files
- Custom APIs

The data load policies will define how and when data are to be brought into the data region. Examples of data load policies are:

- *Preloaded*—load the data region with the external data before the application needs it.
- *Load on demand*—data from the external source are to be loaded to the data region on request from the application.

The data store policies will determine how and when data are to be pushed out of the data region to the appropriate external data source or sources. For example

- *Store all*—all or portions of the data in the data region is predefined for storage to the external systems with the data remaining in the data region after the store operation is executed.
- *Store on demand*—specific data atoms in the data region are stored or copied to the external systems as the application requires. The data remain in the data region after the operation completes. In this situation, the application controls the storage.
- *Purge*—data atoms are removed from the data region.

There are a number of implied properties to data load/store policies, including

- Granularity
- Grouping/frequency
- Invocation

Granularity. One property of the data load/store policies is granularity. Note that there are policy attributes that define just how many atoms are to be loaded into and stored out of the data region. The "xxx on demand" policy attribute implies a varying level of granularity from the smallest data atom to a grouping of any size as defined by the application. The "xxx-all" policy attribute implies that the data pertaining to a specific external system are to be loaded into or out of the data region in a complete block.

Grouping/Frequency. The mechanics of data load and store into and out of the data region involve, either as directed by the application or transparent to the application, when the operation physically is to take place. Each invocation of a load or store (typically for the "on demand" operations) can be done one at a time, or the policy can define a store-and-execute strategy that will group loads

and updates for execution at a later time. This strategy is particularly useful for performance optimizations when

Data atoms are fine-grained

The frequency of the load/store operations are greater than the time period of the external system's ability to transact the operation

Invocation. A third implied property of the data load and store policies is the invocation of the required enterprise information integration (EII) or enterprise application integration (EAI) adapters. The policies must tie the data atom grouping and manage the invocation of the physical data movement into and out of the region via the respective adapters.

Just as there is a relationship between data distribution and replication policies, there is a relationship between data synchronization and data load/store policies. Figure 9.4 illustrates this relationship.

The data management policies of the data grid are interconnected. The data synchronization policy determines the transactional level of the system, and the data

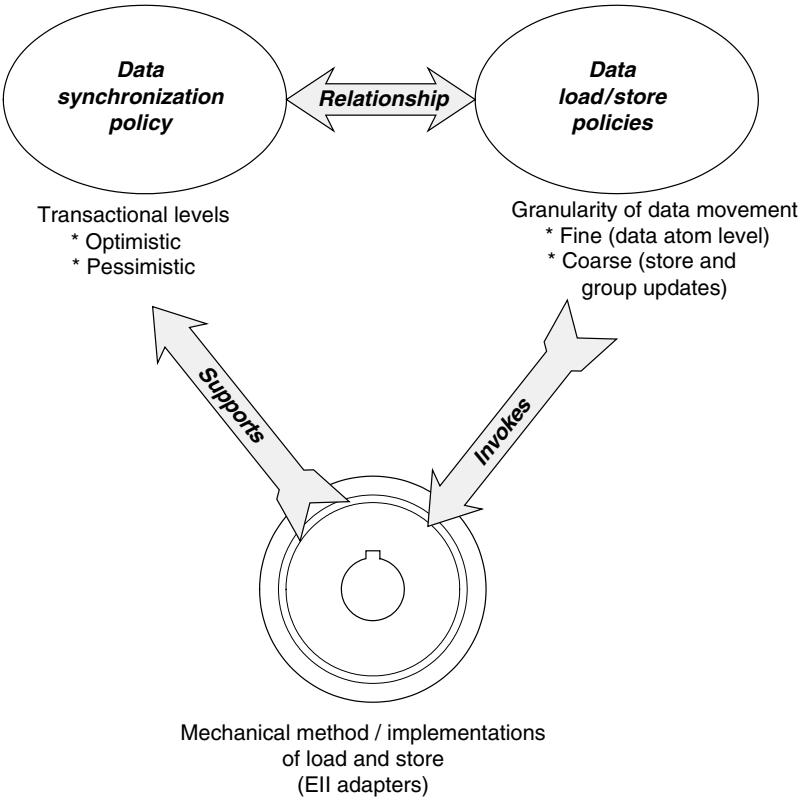

Figure 9.4. Data synchronization and load/store policy relationship.

load/store policies define the granularity and frequency of the process. There is, however, a third part of the equation: the EAI/EII adapter. The adaptor must be able to support the synchronization policy set in the region. For example, if the synchronization policy is optimistic but the EII adapter is a XA transactional, then there is a policy/implementation impedance mismatch. The end result is a system behavior that will not meet the application requirements for data management, performance, and throughput. The physical implementations must support the data management policy set in order for the data grid to operate properly in accordance with the set policies.

Data load/store policies are integral parts of enterprise information integration and are discussed in more depth in a later chapter.

Data Load Policy Expression. The data load policy expression defines the key parameters for loading data into a data region from external sources:

$$DataLoadPolicy = DLP\begin{bmatrix} PolicyName, \\ Region, \\ Granularity(), \\ Adapter() \end{bmatrix}$$

where

- *DPL* is the data load policy function.
- *PolicyName* = logical name of this policy. This is the logical name for this instance of a data load policy. Depending on the implementation of the data grid, this name may or may not be unique.
- *Region* = primary region name. This identifies the primary data region to which this data load policy is applied. A data load policy of identical characteristics—as determined by the *Granularity*( ), *Scope*( ), and *Operation*( ) attributes—can be applied to other regions in the data grid.
- *Granularity* = *F*(*Grouping*( ), *Frequency*( )). The granularity of a data load is defined by the parameters of grouping and frequency. The *Grouping*( ) parameter defines the number of updates that are to be grouped in the data region as a result of queries before the queued updates are applied to the data region. The *Frequency*( ) parameter indicates the minimum frequency or time interval for data load into the data region. Both the grouping and frequency parameters can be static numbers and user-defined functions based on application/service requirements for data load. A *Frequency*( ) of zero identifies a one-time data load into the data region, thereby preloading the data region. Frequency of any other value (negative values do not apply) indicates a load on demand operation (e.g., a frequency of 2 means updating every 2 s).
- *Adapter*( ) = *EIIAdapter*(. . .). The *Adapter* refers to the physical enterprise integration information (EII) or enterprise application integration (EAI)

adapters that will physically perform the loading of data into the data region. Included in these adapters can be the data atom schema and translation logic from the external source into the data atom of the region being loaded. The parameters of the *EIIAdapter*( ) attribute can vary from adapter to adapter implementation as required to perform the required function.

Data Store Policy Expression. The data store policy expression defines the key parameters for loading data into an external data store:

$$DataStorePolicy = DSP \begin{pmatrix} PolicyName, \\ Region, \\ Granularity(), \\ Operation(), \\ Adapter() \end{pmatrix}$$

where

- *DSP* is the data store policy function.
- *PolicyName* = logical name of this policy. This is the logical name for this instance of a data load policy. Depending on the implementation of the data grid, this name may or may not be unique.
- *Region* = primary region name. This is the primary data region to which this data store policy is applied. A data store policy of identical characteristics [as determined by the *Granularity*( ), *Scope*( ), and *Operation*( ) attributes] can be applied to other regions in the data grid.
- *Granularity* = *F*(*Grouping*( ), *Frequency*( )). The granularity of a data store is defined by the parameters of grouping and frequency. The *Grouping*( ) parameter defines the number of updates to be exported out of the data region that are to be queued before the queued exports are applied. The *Frequency*( ) parameter indicates the minimum frequency or time (maximum time interval) with or during which any data export out of the data region must occur. Both the *Grouping*( ) and *Frequency*( ) parameters can be static numbers and user-defined functions based on application/service requirements for data store. A *Frequency*( ) of zero identifies a one-time data export out of the data region. A frequency of any other value (negative values do not apply) indicates a store on demand at a defined interval.
- *Operation* = *F*(*Store*/*Purge*). This defines the resulting state of the data region after a data atom has been exported. *Store*( ) leaves the data region populated with the data atom in the last known state at the time of the store. The *Purge*( ) attribute deletes the data atom from the region after it has been stored.
- *Adapter*( ) = *EIIAdapter*(. . .). The *Adapter*( ) refers to the physical EII, or EAI adapters that will physically perform the export of data out of the data region. Included in the adapter can be the data atom schema and translation logic from the data atom to the external source to which the data are being exported.

The parameters of the *EIIAdapter*( ) attribute can vary among implementations as required, thus meeting the required functions.

Event Notification Policy

Event notification policy is a common paradigm—a tool common to real-time event processing or straight-through processing (STP). Most databases and middleware products support event notification in some way. Databases support event notification through "triggers." The triggering mechanism monitors the state of a database at varying degrees of granularity: a table within a database, a row within a table, or a field within a row. When a monitored entity changes state (due to an event that modifies its state, via an insert, update, or delete action), the database will toggle all "registered triggers" for this event, which will execute the respective registered triggered operations. These operations are typically user-defined. The trigger passes into the user-defined operation (or function) that describes, in detail, what event invoked the trigger. Typical examples of triggers and corresponding user-defined functions are:

- Internal database operations when a row in table A is deleted. The first thing that happens is to find the rows in related tables for which this action effects and then take the appropriate action to other tables maintain referential.
- External operations to database: user-defined programs that will cause a cascade of events or state changes to systems external to the database.

In straight-through processing, or more specifically with message-based middleware, which is a concept of publish and subscribe, an external program will "subscribe" to a published event in the middleware. Any single published event can have many subscribing programs. The published event may trigger cascading events by the invocation of all subscribed programs.

Similarly, within the data grid, event notification plays an integral role. When a data region's state has changed, the following could occur:

- A data atom is added to or deleted from a data region.
- A data management policy is changed.
- A data atom is changed.

Other data regions within the data grid, applications/services, or legacy systems, can register interest in order to be notified should an event take place. The event notification policy is a standard interface that describes how events are to be handled within the data region. Some of the key parameters described via the event notification policy are the events themselves. Programs can register interest in events, and exactly how the invocations are to be managed.

As with the data load and data store policies, event notification policy is an integral part of information integration into and out of the data grid. This will be discussed in more detail in a later chapter.

Event Notification Policy Expression. The event notification policy expression defines the key parameters for the management of events within a data region:

$$EventNotificationPolicy = ENP\begin{pmatrix} PolicyName, \\ Region(), \\ Scope(), \\ Operation() \end{pmatrix}$$

where

- *ENP* is the event notification policy.
- *PolicyName* = logical name for this policy. The *PolicyName* is the logical name for this instance of an event notification policy. Depending on the implementation of the data grid, this name may or may not be unique.
- *Region* = primary region name. This is the primary data region to which this policy is applied. An event notification policy of identical characteristics [as determined by the *Scope*( ) and *Operation*( ) attributes] can be applied to other regions in the data grid.
- *Scope*( ) = *F*(*All*, *List*(*DataAtoms*) = *NULL*). The *Scope*( ) of the event notification policy can be applied to the data atoms of the entire data region as indicated by the *All* attribute or apply only to a specific range of data atoms identified in the supplied list. Note that these parameters are mutually exclusive.
- *Operation* = *F*( ). This is the user-defined function to be invoked on the occurrence of an event.

QUALITY-OF-SERVICE (QoS) LEVELS

Throughout the our discussions on data grid—and its management policies of synchronization, distribution, replication, and load and store—we will be using the terms *quality of service* (QoS) and *QoS levels*. The objective of a distributed data management system, through its data management policies, is to provide a level of end-user support found with traditional data management systems. It is necessary to identify QoS levels on the basis of application requirements when both the business applications and services are running in a distributed grid computing environment. Various categories of QoS levels must be accounted for, including

- Service availability
- Service performance
- Geographic boundary (desktop, data center, cross data center, etc.)
- Data management
- Others

Our discussions will focus on the QoS levels for data management. Some of the QoS features for data management in grid are computing

- Traditional data management service levels, such as support for transactions, query, and embedded logic (i.e., stored procedures).
- Data grid management service levels—for example, regionalization and synchronization.

There are numerous ways to implement the data grid, depending on the user application's requirements as well as the level of service that it demands from a data management system in order for it function properly; this will determine the type of data grid and the QoS level that the data grid must provide.

In Chapter 5 we discussed ways in which to express an application in an equation format via the definition of its parameters and to quantify its data management needs (QoS levels) for the data grid.

The definition for an application in a distributed environment is

$$Application(Work(), Data(), Time(), Geography(), Query())$$

where the various functions can have the following parameters:

$$Work(batch/atomic, synchronous/nonsynchronous)$$
$$Data(overallsize, atomicsize, transactional, transient, queryable)$$
$$Time(Real\text{-}Time, NotReal\text{-}Time, NearReal\text{-}Time)$$
$$Geography(Topology, NetworkBandwidth)$$
$$Query(basic, complex)$$

This equation is broken up into functions: *Work*( ), *Data*( ), *Time*( ), *Geography*( ), and *Query*( ). Each of these parameters quantifies a level of service that a data grid must provide to support the service or application. The QoS level that a data grid provides is determined by its management policies within the data region. These policies are mentioned in this chapter, but will be discussed in more detail in the chapters to follow.

As with most service-oriented architectures, the service is judged by the customer's satisfaction level as a result of using the service. The most fundamental aspect to customer satisfaction is the service's ability to meet the business need. However, there are additional key factors necessary to measure customer satisfaction, such as

- Availability when needed during peak as well as off-peak times
- Cost of the service
- Performance time to run the service

Therefore, the responsibility of the data grid is to meet the QoS demands on all levels in order to accommodate and manage the overall customer experience.