Data Serialization and Deserialization

Serialization is the act of representing an object in a format that can be written to disk or sent via a network. De-serialization is the act of reconstructing an object from the serialized representation. For example, a hash table can be serialized to an array of key-value records.

Serializer Benchmarks

The .NET Framework comes with several generic serializers that can serialize and de-serialize user-defined types. This section weighs the advantages and disadvantages of each, benchmark serializers in terms serialization throughput and serialized message size.

First, we review the available serializers:

- `System.Xml.Serialization.XmlSerializer`
  - Serializes to XML, either text or binary.
  - Works on child objects, but does not support circular references.
  - Works only on public fields and properties, except those that are explicitly excluded.
  - Uses Reflection only once to code-generate a serialization assembly, for efficient operation. You can use the sgen.exe tool to pre-create the serialization assembly.
  - Allows customization of XML schema.
  - Requires knowing all types that participate in serialization a priori: It infers this information automatically, except when inherited types are used.
- `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter`
  - Serializes to a proprietary binary format, consumable only by .NET Binary Formatter.
  - Is used by .NET Remoting, but can also be used stand-alone for general serialization.
  - Works in both public and non-public fields.
  - Handles circular references.
  - Does not require a priori knowledge of types to be serialized.
  - Requires types to be marked as serializable by applying the `[Serializable]` attribute.
- `System.Runtime.Serialization.Formatters.Soap.SoapFormatter`
  - Is similar in capabilities to `BinaryFormatter`, but serializes to a SOAP XML format, which is more interoperable but less compact.
  - Does not support generics and generic collections, and therefore deprecated in recent versions of the .NET Framework.
- `System.Runtime.Serialization.DataContractSerializer`
  - Serializes to XML, either text or binary.
  - Is used by WCF but can also be used stand-alone for general serialization.
  - Serializes types and fields as an opt-in through the use of `[DataContract]` and `[DataMember]` attributes: If a class is marked by the `[Serializable]` attribute, all fields get serialized.
  - Requires knowing all types that participate in serialization a priori: It infers this information automatically, except when inherited types are used.
- `System.Runtime.Serialization.NetDataContractSerializer`
  - Is similar to DataContractSerializer, except it embeds .NET-specific type information in the serialized data.
  - Does not require foreknowledge of types that participate in serialization.
  - Requires sharing assemblies containing serialized types.

- `System.Runtime.Serialization.DataContractJsonSerializer`
    - Is similar to `DataContractSerializer`, but serializes to JSON format instead of XML format.

Figure 7-3 presents benchmark results for the previously listed serializers. Some serializers are tested twice for both text XML output and binary XML output. The benchmark involves the serialization and de-serialization of a high-complexity object graph, consisting of 3,600 instances of five types with a tree-like referencing pattern. Each type consists of `string` and `double` fields and arrays thereof. No circular references were present, because not all serializers support them; however, those serializers that support circular references ran significantly slower in their presence. The benchmark results presented here were run on .NET Framework 4.5 RC, which has slightly improved results over .NET Framework 3.5 for tests using binary XML, but otherwise there is no significant difference.

The benchmark results show that `DataContractSerializer` and `XmlSerializer`, when working with binary XML format, are fastest overall.

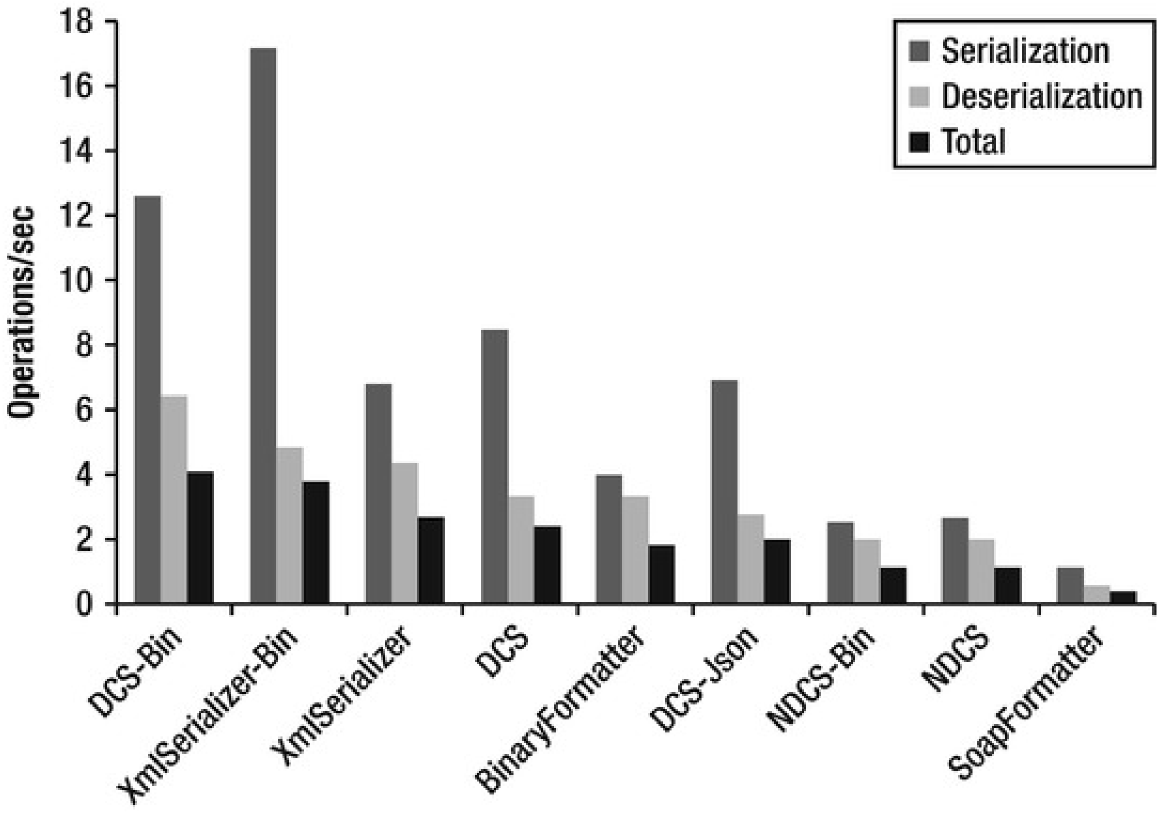

Figure 7-3. Serializer throughput benchmark results, in operations/sec

Next, we compare the serialized data size of the serializers (see Figure 7-4). There are several serializers that are very close to one another in this metric. This is likely because the object tree has most of its data in the form of strings, which is represented the same way across all serializers.

The most compact serialized representation is produced by the `DataContractJsonSerializer`, closely followed by `XmlSerializer` and `DataContractSerializer` when used with a binary XML writer. Perhaps surprisingly, `BinaryFormatter` was outperformed by most other serializers.

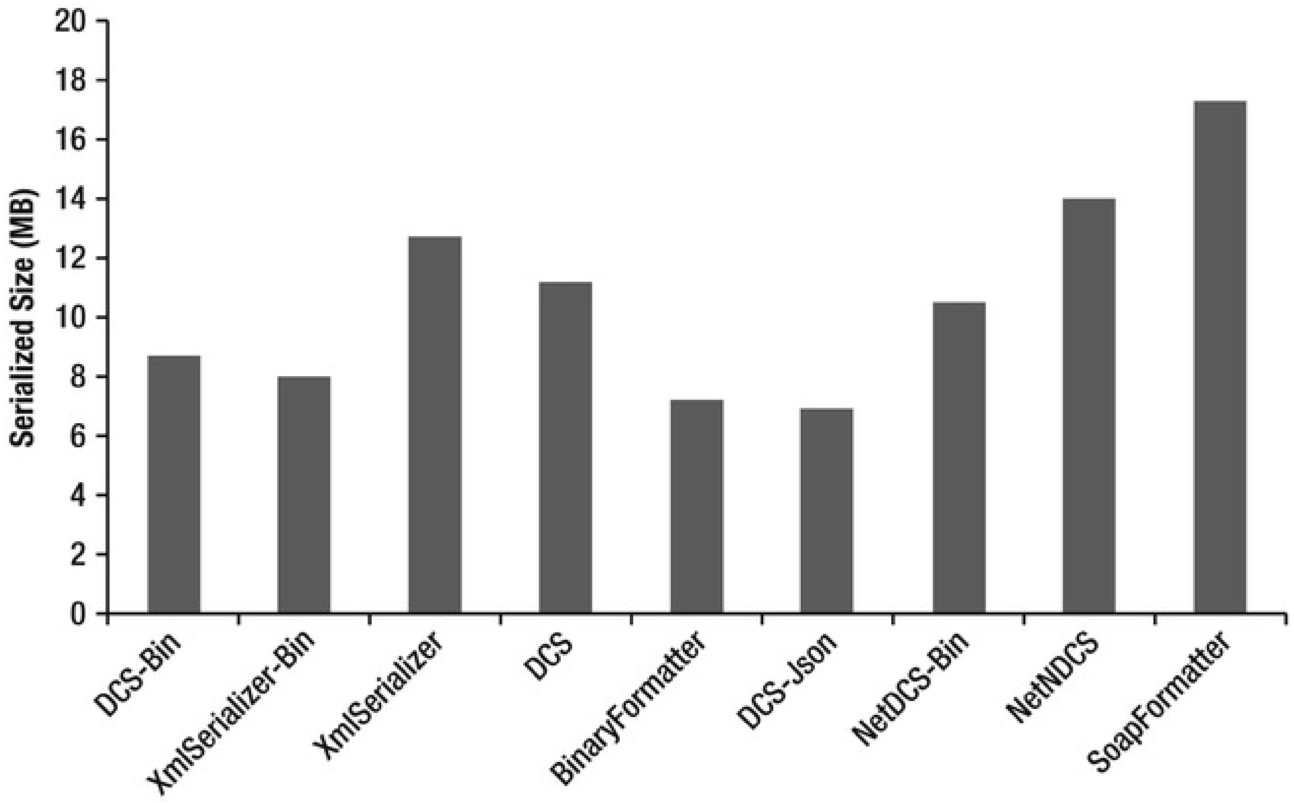

Figure 7-4. *Comparison of serialized data size*

DataSet Serialization

A `DataSet` is an in-memory cache for data retrieved from a database through a `DataAdapter`. It contains a collection of `DataTable` objects, which contain the database schema and data rows, each containing a collection of serialized objects. `DataSet` objects are complex, consume a large amount of memory, and are computationally intensive to serialize. Nevertheless, many applications pass them between tiers of the application. Tips to somewhat reduce the serialization overhead include:

- Call `DataSet.ApplyChanges` method before serializing the `DataSet`. The `DataSet` stores both the original values and the changed values. If you do not need to serialize the old values, call `ApplyChanges` to discard them.
- Serialize only the `DataTables` you need. If the `DataSet` contains other tables you do not need, consider copying only the required tables to a new `DataSet` object and serializing it instead.
- Use column name aliasing (`As` keyword) to give shorter names and reduce serialized length. For example, consider this SQL statement:`SELECT EmployeeID As I, Name As N, Age As A`