

WHITE
PAPER
SERIES

ObjectSpark Performance Features

DataTern, Inc.
330 Madison Ave 31st Floor
New York, NY 10017
(212) 210-6221
www.datatern.com



© 2009 Datatarn, Inc.

The information in this Document is the intellectual property of Datatarn, Inc. This Document may not be reproduced in whole or in part, by any means, without the written consent of DataTern, Inc. All Rights Reserved.

The Software described in this Document and all copies of the Software are the property of Datatarn, Inc. The Software is provided under a license agreement containing restrictions on use and disclosure. The software contains trade secrets of Datatarn, Inc. Reverse engineering of the Software is prohibited.

If provided to the U.S. Government, this Software and this Document are provided with Restricted Rights. Use, duplications, or disclosure by the Government is subject to restrictions set forth in FAR 52-227-19 (c) (2) (June 1987) or DOD FAR Supplement 252.227-7013 (c) (1) (ii) (June 1988) or the NASA FAR Supplement as applicable. Contractor/Manufacturer is Datatarn, Inc.

Trademarks

Datatarn is a registered trademark and ObjectSpark, the ObjectSpark logo, and the Datatarn logo are trademarks of Datatarn, Inc.

Microsoft and Visual Studio are registered trademarks and Windows, Windows NT, Visual Basic, Visual C++, SQL Server, and Microsoft Transaction Server are trademarks of Microsoft Corporation in the United States and/or other countries.

Oracle is a registered trademark of Oracle Corporation.

IBM and DB2 are trademarks of International Business Machines Corporation.

Rational Rose is a trademark or registered trademark of Rational Software, Inc., in the United States and/or other countries.

Sun, Sun Microsystems, the Sun logo and JavaBeans are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

All other product and company names mentioned in this document are trademarks or registered trademarks of their respective companies or organizations.

DataTern, Inc.
330 Madison Ave 31st Floor
New York, NY 10017
(212) 210-6221
www.datatarn.com

ObjectSpark Performance Features

This document provides an overview of:

- Key performance features offered by the ObjectSpark runtime

- Features that enable you to tune the performance of your data access mapping model, including the ability to leverage certain performance capabilities in your database in the behavior of your data components

- Client-side features that you can leverage when calling data component services

Data Component Server Performance Features

ObjectSpark was designed to deliver the fastest run-time performance possible within a multi-user environment. The ObjectSpark runtime has the following performance features:

- Dynamically optimizes the SQL used to access the database to minimize the number of tables accessed and JOIN operations performed

- "Lazy reads" and "just-in-time" data retrieval mechanisms to access the database only when needed

- An efficient system for state management that minimizes the number of network round trips and the extent to which shared resources are locked for exclusive access while still ensuring the integrity of the data, both on the client side and on the database side

- Takes advantage of existing transaction managers (like MTS/EJB), or provides its own.

Optimized Dynamic SQL Generation

At run time, the ObjectSpark runtime components use your mapping information model to dynamically construct one or more optimized SQL commands for the database to execute the requested read or write operation. The mapping information model includes information about attribute-to-column mappings, read/write accessibility of data, join information, write operation definitions, and table weights — all of which let the ObjectSpark runtime dynamically optimize the SQL code for the requested data access, particularly in cases where there are redundant data stores.

For example, if an attribute is mapped for read access to multiple columns containing redundant data values, the server uses the table weight information to read the data values from the smaller table.

In general, dynamic optimization tries to access as few tables as possible and to execute the minimum number of joins needed to return the information required to satisfy a request.

Use of Lazy Reads

The ObjectSpark runtime uses "lazy reads" to enhance the performance of applications at run time. When a query is defined to retrieve data for a given object from the database, the server does not issue the SQL

SELECT command immediately. Queries are not executed until an attempt has been made to read or modify an object attribute. This enables you to narrow the query before the database is accessed.

"Just-In-Time" Data Retrieval

The ObjectSpark runtime's use of lazy reads and optimized data retrieval behavior lets you optimize component performance by populating objects with only the information needed to support a given request. However, this approach runs the risk of having caller trying to read an attribute that has not yet been populated.

To ensure that information is always available when needed, the ObjectSpark runtime provides "just-in-time" data retrieval. This means that the server is always aware of which attributes have been populated and which have not. If an object does not have an attribute in memory when there is an attempt to use that attribute, the server automatically issues the specific SQL command needed to retrieve the additional attribute data from the database. This behavior is completely transparent to the caller.

Managing Concurrent Access to Data

In addition to network bandwidth and resource utilization, the type of support provided for managing concurrent access to data can significantly affect application performance. There are two possible concurrency models:

Pessimistic – The database locks the records being changed as soon as editing begins. The records are unlocked when all changes are complete. Only one user at a time can update a given record at a time.

Optimistic – The database locks the records being changed only when the changes are committed. Two or more users can access the same record at the same time. The database must be able to reconcile (or simply reject) changed records that have been edited by multiple users prior to commit.

For performance and scalability, ObjectSpark uses *optimistic* concurrency.

Caching object data in the ObjectSpark runtime is desirable from a performance perspective, but it requires *optimistic concurrency* so that database resources are free for others to use. This means you must have a way to detect and resolve any discrepancies resulting from one user saving changes to the records that another user is accessing.

Data Component Performance Tuning

The breadth and flexibility of the ObjectSpark ObjectSpark Designer mapping capabilities provides many ways to fine-tune the performance of your data services layer. Developers can use their understanding of the database schema and SQL logging information to optimize the mapping that controls a number of performance factors. For example, you can define which joins are performed in what order, how much data is read, and which tables are accessed.

Because the ObjectSpark Designer differentiates read mappings from write mappings, you can streamline the performance of read operations, while still allowing write operations to have all the complexity that the database requires to preserve the integrity of the data.

You can also create mappings that take advantage of database performance enhancements, such as stored procedures, triggers and cascading deletes, which may exist within the database. How data components are mapped can have a significant impact on their performance.

Differentiating Read and Write Behavior

ObjectSpark mapping models allow for differentiating the read and write behavior of components. This enables the ObjectSpark runtime to optimize read operations, while still supporting the complexity that is often required to perform write operations. You can even create constructs to define read-only classes that can be instantiated in memory outside the context of any transaction. Differentiating read and write mappings is especially advantageous with redundant data stores, which is often the case with production databases.

For *read operations*, the ObjectSpark runtime is designed to read the value of an attribute from any column to which the attribute is mapped for read access. At run time, the server uses the read mapping information (attribute maps, join information, attribute retrieval groups, table weights) to determine the optimal column from which to read the attribute value to satisfy the given request.

Write operations are often more complex than reads. To save modifications to an object, you can create mappings to update each of the data sources (columns in different tables) to which the object's attributes are mapped for write access.

In addition to providing mappings for saving attribute values, the ObjectSpark ObjectSpark Designer provides support for defining additional SQL commands to be executed when a change to an object or a change to an object relationship is saved. The ObjectSpark runtime uses this write mapping information when an object is saved or removed from the database.

Optimizing Data Retrieval Behavior

The ObjectSpark Designer enables you to optimize the data retrieval behavior of objects based on the database structure and component usage patterns. There are several ways you can tune the data retrieval behavior of your components.

- Edit the join information, either to define alternative join paths or to change the sequence in which joins are performed

- Mark attributes that contain BLOBs or data that stored in different tables to be retrieved only when requested.

- Map classes to views in cases where complex joins are required to populate certain objects

- Use stored procedures

Influencing How Much Data is Read

By default, the ObjectSpark runtime retrieves *all* attributes of an object whenever any one of the object's attributes is requested. However, the ObjectSpark ObjectSpark Designer lets you set the **Retrieve On Request** property on one or more attributes of a class to cause those attributes to only be retrieved when the caller reads that attribute value. This makes it possible to avoid retrieving large objects or to avoid executing JOINS or SELECTs on other tables to retrieve data that may not be needed.

For example, suppose that a `Person` class has four attributes: `personId`, `name`, `address`, and `photo`. The `Person.photo` is a binary object that is mapped to a column in a different table from the others. Therefore, the data component developer may elect to set the Retrieve On Request property for the `Person.photo` attribute. At run time, when the caller instantiates a `Person` object and reads the `name` attribute, the query that selects only the `personId`, `name`, and `address` values of the requested `Person` object. This avoids the performance hit that would result from retrieving the `photo` value when

the photo value is not needed. The photo value is automatically retrieved when the caller reads that attribute.

Influencing How Related Data is Accessed

The definition and sequence of joins used to access related data in a database has a significant impact on the performance of data retrieval operations for objects that are mapped to more than one table, or for objects that hold associations with other related objects. The ObjectSpark ObjectSpark Designer enables you to define both the *type* and the *sequence* of the joins used to retrieve related data. The ObjectSpark Designer also supports the concept of a *logical* join, which uses multiple SELECT commands to return the same result as a join in situations where a physical join is not possible, such as in accessing related data in other databases.

Coordination with Database Performance Features

ObjectSpark gives you a simple component interface while taking advantage of many of the performance tuning features that may exist in your database.

Calling Stored Procedures

Stored procedures are compiled SQL procedures that execute on the database server. They offer significant performance advantages to applications accessing the database. Stored procedures are typically useful in the definition of add, update, and remove operations.

ObjectSpark allows you to use stored procedures in the definition of read and write operations. You can map input and output stored procedure parameters to attributes of a class, columns of a table, constant values, or named variables. ObjectSpark also provides support for handling stored procedures that return a result set.

Using Timestamps

The ObjectSpark runtime can use timestamp information to check for dirty data at run time. If you mark an attribute as being mapped to a timestamp in your mapping model, only that field will be checked when the server compares your originally retrieved data to the current persistent state in the database.

Avoiding Legacy Database Implementation Conflicts

The ObjectSpark ObjectSpark Designer has many features for handling redundant data and maintaining the integrity of data relationships. However, databases also have mechanisms, such as triggers and cascading deletes, to perform similar functions. By capturing information regarding the presence of these mechanisms in the database, the ObjectSpark Designer gives you the information needed to avoid generating SQL commands that conflict with or duplicate existing database mechanisms.

For example, suppose a department name is stored in two tables, `TDepartment` and `TEmployee`. You could define a mapping model that causes the corresponding `TEmployee.deptName` fields to be updated whenever a given `TDepartment.name` is saved. However, if the database already has a trigger defined on `TDepartment.name` that is defined to update the redundant data values, you can have the `Department` object just update the value in `TDepartment.name` and let the database trigger handle the update of the related data.

Mapping Classes to Large Denormalized Tables

To enhance database performance by reducing the number of tables that applications need to open, legacy databases often have large tables containing data for many different types of related entities. ObjectSpark allows you to define a *where item* on a class that causes a subset of rows to be selected from the table for membership in a class. For example, if the database has a `Products` table that contains data for all types of products, the object model may each product type as a subclass of `Product`. The mapping for each subclass would then be defined to select the rows of the `Products` table that correspond to the given subclass.

Tuning Client Components for Performance

The **ObjectSpark Data Services API** provides a number of performance tuning features that you can use in developing application components that call your data services layer. For example, because ObjectSpark components use lazy reads, you can often refine queries before they are executed. This can have a significant impact on the overall performance of the system. Finally, developers can choose whether or not objects are instantiated within or outside the context of an application transaction.

Minimize Data Retrieval through Query Refinement

For each class in the object model, the ObjectSpark ObjectSpark Designer generates a list component, which can be used to retrieve collections of object data in the form of a list. By default, a list selects *all* objects of the type defined by the list component. The larger the number of records selected, the slower the performance. The ObjectSpark Data Services API provides support for refining the query expression to narrow the criteria used to select records *before* the `SELECT` command is sent to the database.

Query expressions can be relatively complex, including both logical and relational operators. For example, you can define a query expression that selects a narrower range of objects for membership in a list. This not only enhances the speed with which a result is returned, but it also makes better use of the memory where object data is cached.

Use of Transaction Contexts

ObjectSpark-generated data components enable clients to instantiate objects within or outside the context of a client transaction. For example, a business component can expose the interface to a `USAddresses` data component that holds a read-only list of zip codes, cities and states. The list can exist outside the context of any transaction and can be shared by multiple clients in a client drop-down list.

Another component can define a `Payroll` transaction that iterates through a number of `Account` data components to update the `PayAmount` on each object before committing the overall transaction. Committing the `Payroll` transaction causes the data components to commit the database transactions on the back end, which must all succeed before the `Payroll` transaction can succeed.

Working Within a High Performance Architecture

ObjectSpark is designed to work within the high-performance architecture defined by MTS/COM+ and (soon) the .NET platform, or EJB/J2EE application servers such as BEA WebLogic™ and IBM WebSphere™, and your database servers. It leverages the services of transaction coordination, threads management, and database connection pooling of the above servers.

You can use Universal Adapters with a variety of databases and, as previously discussed, it enables data component developers to take advantage of the performance features of databases.

The ObjectSpark Designer also provides flexibility in how you deploy your Universal Adapters and applications so you can balance the load on any given server based on usage patterns.

Leveraging Your Database Investment

The ObjectSpark architecture is database independent. ObjectSpark software works in a heterogeneous database environment and supports the integration of multiple physical databases even within a single object. You can optimize these implementations for performance by using logical joins and other mechanisms. Currently, ObjectSpark software uses OLE DB and JDBC providers for access to relational databases (depending on the target deployment platform).

Deployment Flexibility for Scalability and Load Balancing

ObjectSpark supports the design and configuration of components based upon your physical deployment strategies. You can deploy client applications, Web application servers, business components, data component servers, and databases in a wide variety of physical configurations.

You can deploy as many Data Component Servers you need support your application requirements. ObjectSpark imposes *no limitation* on the location or number of servers that can be deployed for scalability and performance. In a distributed application environment, the Data Component Servers usually reside near the business object servers. Whether or not your web server is deployed on the same physical system as your business components and data components depends on the capacity of your server. The database typically resides on its own system. You have the flexibility to try different configurations to optimize the performance of your application..

Leveraging Application Servers Performance Features

ObjectSpark is designed to leverage many of the performance features provided by application servers such as MTS, .NET, and EJB/J2EE Servers.

Transaction Management

ObjectSpark Universal Adapters leverage the transaction coordination services of the above mentioned servers to coordinate transactions across one or more data sources.

Threads Management

ObjectSpark-generated data components are designed to operate across multiple threads, as supported by the servers.

Database Connection Pooling

The ObjectSpark Universal Adapter takes full advantage of the server's specific services, including *connection pooling* to optimize the use of your database connections. Most servers pool database connections so that a connection can be opened once to serve many clients. Conceptually, the servers play "traffic cop" between the clients and the database. From the perspective of the Universal Adapter, all access to the database is managed transparently by the server, if one exists. In this scenario, the adapter does not have to open and close connections and does not have to perform any database transaction management.

If an application server is not present, the adapter will provide its own transaction management and connection pooling.