January/February 2010 On Technical/Customizing E1

JDE® Cache Design Patterns

By Brian Oster

E1***Editor's Note:*** *How much do you know about JDE Cache implementation? If your answer is "not as much as I think," then get ready for a crash course courtesy of Brian Oster. Brian demonstrates how various API calls fit together and when you would want to apply them, with emphasis on the two main design patterns used for Cache implementations: Shared Cache and Multiple Instances, along with lots of code you can use. He's even put together a code zip file that you can apply to your 9.0 environment. So get ready to "cache" in on this great article.*

Abstract

I have been doing JDE development for almost eight years and I have come in contact with quite a few JDE Developers. While most have a basic understanding of JDE Cache, I have found that very few actually use it or truly understand how it is implemented in pristine code. And for good reason; while the JDE documentation does a pretty good job at documenting the individual API calls, it really doesn't give a developer who is unfamiliar with JDE Cache a real sense of how all the various API calls fit together or how to actually apply it to a real world situation. Hopefully this article will start to bridge that gap.

While most have a basic understanding of JDE Cache, I have found that very few actually use it or truly understand how it is implemented in pristine code.

To actually implement JDE Cache in your own applications will require the ability to write C Business Functions. As such, this article is targeted primarily at JDE Developers with a working knowledge of C BSFNs. However, the concepts presented here will be valuable to any JDE Developer, or anyone else for that matter, who simply wants a better understanding of JDE Cache.

This guide is NOT meant to be a complete, all encompassing replacement for the Oracle® API documentation, nor does it cover all the JDE Cache API calls. Rather, it is meant to supplement the Oracle documentation, with a specific emphasis on the two main design patterns used when implementing a JDE Cache process. I also throw in a few tips and traps that I have found when working with the API, along with some recommended best practices.

Preface

The API calls and concepts presented here should be compatible with EnterpriseOne® releases from Xe® to 9.0, although there may be some slight variations between releases in regards to Unicode compliance. Any actual code examples were created in E1 9.0 Standalone and should be ANSI C and JDE compliant. Any Pseudo code examples are not meant to represent actual C syntax, even though they may have that appearance.

One other note, I present two design patterns in this article: one is called the "Shared Cache" design pattern and the other is the "Multiple Instance" design pattern. I have seen the term "Shared Cache" in pristine code comments and other places and, as such, it may be recognized as an industry standard term. However, the term "Multiple Instance" as it refers to JDE Cache is a term I created. If some of you JDE veterans out there know of a more widely accepted term for the Multiple Instance Cache Pattern, please email me at the address listed in my bio and enlighten me

Glossary of Terms

Before we begin, let's review the terms used in this article, so that everyone understands them the same.

- **Cache** – JDE Cache
- **User Session** – The End User's JDE Session either on the local workstation or on the JDE Application (Batch) Server.
- **Client Application** – Any application or external process (APPL, UBE, BSFN, etc.) that calls the BSFN(s) that contain your JDE Cache processes.
- **Client Programmer** – Programmer creating the Client Application that will be using your BSFN(s) that use JDE Cache.
- **Public Function** – The actual function calls inside of a BSFN source file that can be called by ER code or by jdeCallObject.
- **Globally Unique Identifier (GUID)** – 128 bit number, that while not guaranteed to be unique, is so large that the probability of the same number being generated twice is extremely small. A GUID or Job Number can be used for unique cache records and/or cache names (See **Job Number**).
- **Job Number** (mnJobnumberA) – This is a unique next number per user session. It is generated by calling the JDE API JDB_GetInternalNextNumber. The number returned is unique per user and JDE session. In other words, when the user logs in, the first call to JDB_GetInternalNextNumber returns 1, then 2, and so on. If the user logs out and back in, it starts over with 1. It is used quite often to uniquely identify records in a Shared Cache design pattern or as part of the Cache Name in a Multiple Instance Cache design pattern (more on Design Patterns later on). Other unique identifiers such as a GUID could also be used. For the purpose of this article, Job Number and GUID will be used interchangeably.
- **Cache Structure** – Strictly speaking, this is the data structure for a record stored in cache. However I use the term not only as the data structure of the records in a cache instance, but also the key definitions. This is analogous to a database Table Structure or Table Definition, along with the primary and secondary key definitions.
- **DB Work Table** – Generally, this is a database table that temporarily holds records during a transaction or process. At the completion of the transaction or process, the records are deleted from the work table.

What Is JDE Cache?

JDE Cache is a powerful and handy set of API calls that you can use in your C BSFNs. It is used extensively within JDE pristine code.

JDE Cache has many uses. For example it can be used to share data or records among multiple applications or UBEs. As every JDE developer knows, APPL and UBE interconnects can be used to pass values. But what if you need to pass an array of values, multiple records, or data