

# Jasmine Tips

Document Version: 1.00  
December 1998

Peter Fallon  
© Castle Software Australia Pty Ltd

## Introduction

This paper was created in response to questions I've seen posted, and to provide information I think Jasmine users need. Much of this is information that CA themselves is happy to discuss, but which does not appear to reach some of the forums and sites I visit for technical info.

There is no real organisation here, and most of it is in point form. There are three loose topic groups here:

- Design
- Implementing designs – Physical Design issues
- Performance

## Database Design

1. **OO Designs** are definitely NOT the same as relational designs. Forcing a relational approach on a OODB leads straight to poor performance, and difficulty coding. Find a good text on Object Oriented Analysis and Design, and study it!
2. OO Design goes through the same stages of **Logical** design and **Physical** design that you used for relational projects. Physical design must take the following factors into consideration: **Performance**, **Recoverability** and **Maintainability**. Minor issues include: Testing and Deployment (moving a system from one server/database to another).
3. **Performance Goals**: Maximise transaction rate for system, minimise response times per session, and maximise the number of concurrent users possible.
4. **Recoverability Goals**: Be able to cope with hardware failure (can you restore your entire database from backups? How recent are they?). Software failures – will your database handle programming errors, failed transactions, have you coded defensively? How well will the system cope with human error, or deliberate misuse?
5. **Maintenance Goals**: How extensible, maintainable documented is your database? Are you prepared to unload, move, reconstruct the schema, make changes and reload data at a moments notice? (There are more ideas on this further on)
6. **Naming Conventions**. The following are the naming conventions used in the Jasmine sample applications:

Data Type	Rule	Examples
Class Families	Capitalise the first letter, and add CF to the end of the name	CastleCF
Classes	Always use singular noun, Capitalise first letter of words	Garment, Person, Project
Properties	Singular, lower case first letter, then capitalise the first letter of each word in name	name, address, hoursWorked
Collection Properties	As above, but use plurals	garments, addresses
Object References	As per properties, but use the same name as the target class (if meaningful)	garment, garments
Methods	Use a verb as all or part of the name	addCustomer(), printReport()

7. **Scripts and Class Definitions A.** While the Database administrator is a great tool for ad-hoc database work, and querying schema information, NEVER use it for database definitions. The reason? You cannot reproduce what you just did, nor keep a record of it. If you have to recover the database at a later time, or reproduce any schema changes you have no documentation **unless you do it all with ODQL scripts using the ODQL Interpreter**. This is exactly the same reason why relational database administrator use scripts to modify their databases – they can always rollback or rollforward any structural changes at a moments notice!
8. **Scripts and Class Definitions B.** Keep your class definition scripts separate from method source/definitions. This makes it easier to make single changes to individual methods, add new methods... etc without redefining the class each time. Generally, keep your scripts as granular as possible.  
  
Remember, Jasmine is a server database. If you are going to write mission critical database applications using it, you have to use all of the same quality assurance procedures used on other systems. The specifics may change for the OO nature of Jasmine, but the general principles do not.
9. **Use Server side method execution** by default. Data processing is quicker, it reduces network traffic enormously, maximises code reuse, supports multiple client platforms, and is easier to maintain (no deployment of changes to clients required).
10. **Use Client Side methods when:** Very high server loads; complex user interface requirements (or data requires processing that must be tailored for the client machine); or when the raw data is substantially less complex than the final result, eg: data for a virtual reality display. The **Persistent Java** interface requires this approach. Generally avoid unless you have very specific reasons.

## Implementing Database Designs

1. The Jasmine server has to be running if you want to create, delete or otherwise manipulate Stores and Class Families. You do not need an active connection to the database, but you should have an ENV file defined for your default connection.
2. Try to keep **one class family per store**, or families pertinent to a single application in a single store. This makes it easier to track changes, and unload/loads are simpler. It also makes it easier to manage performance tuning (keeping different data in different stores, and thus on different driver potentially!)
3. The Database Administrator should keep regular track of store usage using **listStore**. The only message you'll get if a store fills up is when a transaction fails to save data.
4. Remember that deleteCF and deleteStore permanently remove data – there is no recovery from these operations except from tape.
5. Restoration from backup must be carefully managed –if you restore one store and its extents, does this old data refer/link to objects in other stores (eg: mediaCF)? If so, you'll have to rollback/restore data in those stores as well!
6. Never add your own classes or class families to the Jasmine **system** store.
7. Never add your own classes to **JadeLibCF**, or the sample database **CAStore**.
8. For maximum flexibility, create a duplicate of the multimedia class family and the JadeLibCF class family if you are using Studio (use the unload files on the install CD) for each **separate** application you create. Use Aliasing and the environment file to point jasmine to these copies when you run your application.

The reason? It makes backing up, restoring, migrating to another server, creating installs all so much easier, as you can install or remove this application from a *server without affecting any other part of the Jasmine installation*. If you are creating vertical market software, this is the only way to create and deploy applications. If you are a database administrator, this is the only way you can ensure loads, unloads, backups and restores don't accidentally cause cross application problems.

9. Use **Aliasing** for Class families if the database you are building will be implemented in stages. That provides you with an excellent mechanism to separate DEVELOPMENT, TEST and PRODUCTION

versions of your database. You simply define different connections (and ENV files) on the client for each of Development, Test and Production to use each class family as required.

## Performance Tuning

1. The administrator must keep an eye on disk usage in the controlled multimedia storage area. Use **Setmmarea** to change the directory once that disk is near full.
2. The administrator should use **Tidymmarea** regularly. Jasmine's Controlled storage will always verify files are where they should be, but occasionally files will be left when their instances are deleted from the database. Tidymmarea recovers this space. Be aware that it will take a LONG time to run on a large database, as it performs a very rigorous cross-check of all multimedia files and OID references.
3. Stores and extents should be managed with the view to getting best use out of your database server. Individual extents (and stores) can be on different drives, thus spreading drive loads. Remember that you will get different performance gains according to logical/physical drive mappings, whether physical drives are on the same drive card... etc. This is normal **server load management** stuff.
4. Specifying more than one filename for a single extent invokes **striping** in this extent. This is good for performance, but you don't have any finer control over the striping than this. This is one reason why its best to keep one Class Family per store.
5. Increase performance by keeping the Jasmine **System Store** on a different drive to the server's system files (**boot drive**).
6. Increase performance by keeping **data stores** on a separate drive to the System Store.
7. Increase performance by spreading your application stores on different drives. This is relevant when you have multiple Jasmine systems on the same server.
8. Increase performance by separating keeping the **Work store** and **Transaction store** for connections on a separate drive from the System Store. These are specified in the **Environment (ENV) file** on the Client, which has the same name as the connection they use in the connection dialog. Note that you can specify different server locations for each client if you want to get that fiddly!
9. Web connections (Jasmine Studio) are managed by the connection specified when you run the Deploy to Web option. Edit this connection's ENV file to alter Work and Transaction store locations.
10. The Default **page size** for a store is 8Kb. This is optimal for most situations, but should be enlarged if you have very large class definitions, and you are often reading data sequentially. If you do, you must be careful to specify this pagesize when required in later operations.
11. **Maintain dependency information on your database schema.** That is, if you are forced to RESTORE a store from tape because of a catastrophic failure, you must first restore not only ALL extents for that store (as you don't know how data is loaded across extents, so you don't know where the most recent transactions are stored). Secondly, you need to restore all related stores.

Eg: A Class in store MyStore contains text, numeric and Bitmap properties. The data for the Bitmaps will be stored in mediaCF, in a separate set of stores. If you bring back MyStore, then the data there is potentially out of sync with data in the mediaCF stores, so you have to restore that as well. If bringing the media stores back affects data in other stores, then you have to get them also... ad infinitum.

Be very careful here. Its not a problem – you simply have to manage things carefully, and have a properly designed disaster recovery plan, as per any mission critical database system! Obviously there are design issues here – its one of the reasons I suggested earlier that you keep copies of JadeLibCF and mediaCF for each separate system.

12. You can also get performance gains by carefully designing the **order of properties** in a class. Data for each instance is stored in the same order as properties in the class definition. Thus the first properties are always read in first... etc. As such, property order should be decided on the basis of frequency of use.

13. When Loading data, or running ODQL scripts in the interpreter, use **explicit transactions** – they greatly increase throughput. Consider grouping ODQL expression in blocks, as this also affects how transaction are interpreted.
14. Loads and Unloads require at least 150Kb free space in each of the stores they manipulate.
15. **READ Transactions** make data retrieval much quicker. Use for all queries and data access.
16. Make sure your server is set to **Production mode** (once you have finished all design changes). This increases speed, as the server engine doesn't have to continually monitor the schema for changes.
17. If you get E\_DMFO48\_CLEANUP\_QUEUE\_FULL in the Jasmine Error log, you have too many Read Transactions active (that haven't been committed). Consider changing high profile jobs to exclusive (use the setLockMode(EXCLUSIVE) ODQL command), or adjust the **cleanup\_queue\_length** parameter in your Jasmine configuration.
18. **Use Indexes** – they do speed queries. There is a minor performance hit for writes however.
19. Indexes have to be explicitly created for a class and its subclasses.
20. For important queries, experiment with different approaches, sequences and orders. You may find a small change makes a great different in performance. Break a complex query into smaller parts, and execute it in a chain (eg: extract main objects first, then run another query on the result to extract specific properties, run methods, get properties of contained classes... etc). the reason this is fast, is that the results of the first query remain on the server, and aren't transmitted to the client. the server has everything it needs locally!

For example:

---

```
ls = Track.parent.playingTime() from Track where Track.parent == r;
```

---

is OK, but a far quicker result is obtained by:

---

```
tracks = Track from Track where Track.parent == r;
ls = tracks.parent.playingTime() from tracks;
```

---

21. Using methods in the WHERE clause is very powerful, but it is much slower. However, it is FAR preferable to loading data to the client for further complex sub-queries.
22. Never use relational style key lookups (ie: simulating table joins). This is horribly slow.
23. ODQL code for collections should use **directAdd()** and **directRemove()** methods instead of the add() and remove() methods. The latter result in the creation of a new collection each time, which soaks up disk space in the work store.
24. Collections created via queries and other activities during a session are not removed, or garbage collected until the session finishes. This is extremely important if a session (connection) will be active for a long time, and large collections are being manipulated. Careful and judicious use of Collection level delete() method and the **clear** command is called for. You can also increase the size of the Work store in the ENV file.
25. Adjusting the **DBMS Server Cache** has a small, but beneficial effect (this is server memory dependent). Eg: increase from 256 to 2048.
26. Increase the **DMF cache** (memory used for transient data).
27. If you have a lot of connections performing data updates, experiment with the **System Maxlocks** setting (normally 10). This controls the number of instance that must be locked in a class before the server escalates it to a class lock. Increase this setting to improve concurrency!

### **Copyright Notice**

No part of this paper may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the authors. For information please contact: Castle Software Australia Pty Ltd.

This paper is based on materials from CA-World 98, presentations given to The Melbourne Jasmine Interest

Group, and the experiences of Castle Software Australia. The information and material contained in this document are provided "as is", without warranty of any kind, express or implied, including without limitation any warranty concerning the accuracy, adequacy, or completeness of such information or material or the results to be obtained from using such information or material.

### ***Trademark Acknowledgements***

Jasmine™ is a trademark of Computer Associates, CA-Visual Objects 2.0® is registered trademark of Computer Associates, Microsoft® Visual Basic® is a registered trademark of Microsoft and Microsoft® Office 97™ is a trademark of Microsoft.

All other product names and services identified throughout these notes are trademarks or registered trademarks of their respective companies. They are used throughout these notes in editorial fashion only and for the benefit of such companies. No such uses, or the use of any trade name, is intended to convey endorsement or other affiliation with the notes.