Object-Oriented Database Management Systems Revisited

An Updated DACS State-of-the-Art Report

Contract Number SP0700-98-4000
Subcontract No. 1800313 IAW SOW dated 18 December 1997
(Data & Analysis Center for Software)

Prepared for:
Air Force Research Laboratory -
Information Directorate (AFRL/IF)
525 Brooks Road
Rome, NY 13441-4505

Prepared by:
Gregory McFarland, Andres Rudmik, and David Lange
Modus Operandi, Inc.
122 Fourth Ave.
Indialantic, FL 32903

Under subcontract to:
DoD Data & Analysis Center for Software (DACS)
ITT Industries - Systems Division
Griffiss Business & Technology Park
775 Daedalian Drive
Rome, NY 13441-4909

Unclassified and Unlimited Distribution

DoD Data & Analysis Center for Software (DACS)
P.O. Box 1400
Rome, NY 13442-1400
(315) 334-4905, (315) 334-4964 - Fax
cust-laisn@dacs.dtic.mil
http://www.dacs.dtic.mil

The Data & Analysis Center for Software (DACS) is a Department of Defense (DoD) Information Analysis Center (IAC), administratively managed by the Defense Technical Information Center (DTIC) under the DoD IAC Program. The DACS is technically managed by Air Force Research Laboratory Information Directorate (AFRL/IF) Rome Research Site. ITT Industries - Systems Division manages and operates the DACS, serving as a source for current, readily available data and information concerning software engineering and software technology.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection is estimated to average 1 hour per response including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports. 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project, (0704-0188). Washington, DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	31 January 1999	N/A

4. TITLE AND SUBTITLE
An Updated State-of-the-Art-Report
Oriented Database Management Systems Revisited

5. FUNDING NUMBERS
SP0700-98-4000

6. AUTHORS

Gregory McFarland, Andres Rudmik, and David Lange
Modus Operandi, Inc

7. PERFORMING ORGANIZATIONS NAME(S) AND ADDRESS(ES)

Modus Operandi, Inc, 122 Fourth Ave.
Indialantic, FL 32903

8. PERFORMING ORGANIZATION REPORT NUMBER
DACS-SOAR-99-4

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Technical Information Center (DTIC)/ AI
8725 John J. Kingman Rd., STE 0944, Ft. Belvoir, VA 22060
and Air Force Research Lab/IFTD
525 Brooks Rd., Rome, NY 13440

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
N/A

11. SUPPLEMENTARY NOTES
Available from: DoD Data & Analysis Center for Software (DACS)
775 Daedalian Drive, Rome, NY 13441-4909

12a. DISTRIBUTION/ AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

12b. DISTRIBUTION CODE
UL

13. ABSTRACT *(Maximum 200 words)*

This report reviews the state of the art of Object-Oriented Database Management Systems (OODBMS). The objective of this report is to provide the reader with an understanding of the issues relevant to OODBMS technology and to describe where commercial products stand on these issues. A further objective is to describe the current state of commercial OODBMS in regard to a set of expected facilities for databases in general and object-oriented databases in particular. It is expected that this report will be used as the first step in an evaluation aimed at selecting an OODBMS for use in a given application development effort.

This report describes a broad range of OODBMS evaluation criteria. It then evaluates five commercial OODBMS products in a subset of the overall criteria, focusing on issues relevant to applications development and advanced object-oriented database concepts (e.g., versioning, schema evolution). An appendix to this report defines a template to be used as a guide for performing an evaluation of OODBMS.

14. SUBJECT TERMS
Data, Datasets, OODBMS, Databases, Object-Oriented

15. NUMBER OF PAGES
145

16. PRICE CODE
N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Trademarks

Objectivity and Objectivity/DB are registered trademarks of Objectivity, Inc.

ONTOS, ONTOS DB, and ONTOS Object SQL are trademarks of ONTOS, Inc.

VERSANT, VERSANT ODBMS, and VERSANT View are trademarks of Versant Object Technology Corporation.

Object Design, Inc. and ObjectStore and trademarks of Object Design, Inc.

Object Design and SchemaDesigner are registered trademarks of Object Design, Inc.

GemStone is a registered trademark of Servio Corporation.

Opal and GeODE are trademarks of Servio Corporation.

ITASCA is a trademark of Itasca Systems, Inc.

UNIX is a registered trademark of AT&T.

DEC, DECstation, VAX, VMS, and ULTRIX are trademarks of Digital Equipment Corporation.

Sun is a registered trademark of Sun Microsystems, Inc.

Sun3, Sun4, SunOS, and Solaris are trademarks of Sun Microsystems, Inc.

HP is a registered trademark of Hewlett-Packard Company.

HP/UX is a trademark of Hewlett-Packard Company.

IBM is a registered trademark of International Business Machines Corporation.

AIX and RISC System/6000 are trademarks of International Business Machines Corporation.

Silicon Graphics and IRIS are registered trademarks of Silicon Graphics, Inc.

IRIX is a trademark of Silicon Graphics, Inc.

NCR is a trademark of NCR Corporation.

Other companies mentioned own numerous registered trademarks. All company and product names are registered trademarks of the individual companies.

Table of Contents

Trademarks ____ i

Abstract & Ordering Information ____ 1.

1. Introduction ____ 2.

2. Overview of OODBMS Technology ____ 4.
   - *2.1 The Need for Object-Oriented Databases* .... *4.*
   - *2.2 The Evolution of Object-Oriented Databases* .... *5.*
   - *2.3 Characteristics of Object-Oriented Databases* .... *6.*
   - *2.4 Application of an OODBMS* .... *8.*
   - *2.5 Pragmatics of Using an OODBMS* .... *10.*
   - *2.6 Summary* .... *11.*

3. Evaluation Criteria ____ 12.
   - *3.1 Functionality* .... *13.*
   - 3.1.1 Basic Object-Oriented Modeling .... 14.
     - 3.1.1.1 Complex Objects .... 14.
     - 3.1.1.2 Object Identity .... 16.
     - 3.1.1.3 Classes .... 16.
     - 3.1.1.4 Attributes .... 16.
     - 3.1.1.5 Behaviors .... 17.
     - 3.1.1.6 Encapsulation .... 17.
     - 3.1.1.7 Inheritance .... 17.
     - 3.1.1.8 Overriding Behaviors and Late Binding .... 18.
     - 3.1.1.9 Persistence .... 18.
     - 3.1.1.10 Naming .... 18.
   - 3.1.2 Advanced Object-Oriented Database Topics .... 18.
     - 3.1.2.1 Relationships & Referential Integrity .... 19.
     - 3.1.2.2 Composite Objects .... 20.
     - 3.1.2.3 Location Transparency .... 21.
     - 3.1.2.4 Object Versioning .... 21.
     - 3.1.2.5 Work Group Support .... 21.
     - 3.1.2.6 Schema Evolution .... 22.
     - 3.1.2.7 Runtime Schema Access/Definition/Modification .... 23.
     - 3.1.2.8 Integration with Existing DBs and Applications .... 24.
     - 3.1.2.9 Active vs. Passive Object Management System .... 24.
   - 3.1.3 Database Architecture .... 25.
     - 3.1.3.1 Distributed Client - Server Approach .... 25.
     - 3.1.3.2 Data Access Mechanism .... 26.
     - 3.1.3.3 Object Clustering .... 26.
     - 3.1.3.4 Heterogeneous Operation .... 26.
   - 3.1.4 Database Functionality .... 26.
     - 3.1.4.1 Access to Unlimited Data .... 27.
     - 3.1.4.2 Integrity .... 27.
     - 3.1.4.3 Concurrency .... 27.
     - 3.1.4.4 Recovery .... 28.
     - 3.1.4.5 Transactions .... 28.

Table of Contents Continued

3.1.4.6 Deadlock Detection .... 29.
3.1.4.7 Locking .... 29.
3.1.4.8 Backup and Restore .... 29.
3.1.4.9 Dump and Load .... 30.
3.1.4.10 Constraints .... 30.
3.1.4.11 Notification Model .... 30.
3.1.4.12 Indexing .... 30.
3.1.4.13 Storage Reclamation .... 31.
3.1.4.14 Security .... 31.
3.1.5 Application Programming Interface .... 31.
3.1.5.1 DDL/DML Language .... 32.
3.1.5.2 Computational Completeness .... 32.
3.1.5.3 Language Integration Style .... 32.
3.1.5.4 Data Independence .... 32.
3.1.5.5 Standards .... 33.
3.1.6 Querying an OODBMS .... 33.
3.1.6.1 Associative Query Capability .... 34.
3.1.6.2 Data Independence .... 34.
3.1.6.3 Impedance Mismatch .... 34.
3.1.6.4 Query Invocation .... 34.
3.1.6.5 Invocation of Programmed Behaviors .... 34.
3.2 Application Development Issues .... 34.
3.2.1 Developer's View of Persistence .... 35.
3.2.2 Application Development Process .... 35.
3.2.3 Application Development Tools .... 35.
3.2.3.1 Database Administration Tools .... 35.
3.2.3.2 Database Design Tools .... 36.
3.2.3.3 Source Processing Tools .... 36.
3.2.3.4 Database Browsing Tools .... 36.
3.2.3.5 Debugging Support .... 36.
3.2.3.6 Performance Tuning Tools .... 36.
3.2.4 Class Library .... 36.
3.3 Miscellaneous Criteria .... 36.
3.3.1 Product Maturity .... 37.
3.3.2 Product Documentation .... 37.
3.3.3 Vendor Maturity .... 37.
3.3.4 Vendor Training .... 37.
3.3.5 Vendor Support & Consultation .... 37.
3.3.6 Vendor Participation in Standards Activities .... 38.

4. Evaluation Targets ________ 387.
4.1 Objectivity/DB .... 38.
4.2 ONTOS DB .... 40.
4.3 VERSANT .... 40.
4.4 ObjectStore .... 41.
4.5 GemStone .... 42.
4.6 ObjectStore PSE Pro .... 43.
4.7 IBM San Francisco .... 44.

Table of Contents Continued

5. Evaluation Reports ____ 44.

5.1 Evaluation of Objectivity/DB 2.0 .... 45.

5.1.1 Application Development Issues .... 45.

5.1.1.1 Developer's View of Persistence .... 45.

5.1.1.2 Application Development Process .... 46.

5.1.1.3 Application Development Tools .... 46.

5.1.1.3.1 Database Administration Tools .... 46.

5.1.1.3.2 Database Design Tools .... 47.

5.1.1.3.3 Source Processing Tools .... 47.

5.1.1.3.4 Database Browsing Tools .... 48.

5.1.1.3.5 Debugging Support .... 49.

5.1.1.3.6 Performance Tuning Tools .... 50.

5.1.1.4 Class Library .... 51.

5.1.2 Advanced Topics .... 51.

5.1.2.1 Transaction and Concurrency Model .... 51.

5.1.2.2 Relationships & Referential Integrity .... 52.

5.1.2.3 Composite Objects .... 53.

5.1.2.4 Location Transparency .... 54.

5.1.2.5 Object Versioning .... 54.

5.1.2.6 Work Group Support .... 55.

5.1.2.7 Schema Evolution .... 55.

5.1.2.8 Runtime Schema Access/Definition/Modification .... 57.

5.2 Evaluation of ONTOS DB 2.2 .... 57.

5.2.1 Application Development Issues .... 57.

5.2.1.1 Developer's View of Persistence .... 57.

5.2.1.2 Application Development Process .... 58.

5.2.1.3 Application Development Tools .... 59.

5.2.1.3.1 Database Administration Tools .... 59.

5.2.1.3.2 Database Design Tools .... 59.

5.2.1.3.3 Source Processing Tools .... 60.

5.2.1.3.4 Database Browsing Tools .... 61.

5.2.1.3.5 Debugging Support .... 61.

5.2.1.3.6 Performance Tuning Tools .... 62.

5.2.1.4 Class Library .... 63.

5.2.2 Advanced Topics .... 64.

5.2.2.1 Transaction and Concurrency Model .... 64.

5.2.2.2 Relationships & Referential Integrity .... 65.

5.2.2.3 Composite Objects .... 66.

5.2.2.4 Location Transparency .... 66.

5.2.2.5 Object Versioning .... 67.

5.2.2.6 Work Group Support .... 67.

5.2.2.7 Schema Evolution .... 67.

5.2.2.8 Runtime Schema Access/Definition/Modification .... 67.

5.3 Evaluation of VERSANT Release 2 .... 68.

5.3.1 Application Development Issues .... 68.

5.3.1.1 Developer's View of Persistence .... 68.

5.3.1.2 Application Development Process .... 69.

5.3.1.3 Application Development Tools .... 70.

Table of Contents Continued

5.3.1.3.1 Database Administration Tools .... 70.
5.3.1.3.2 Database Design Tools .... 71.
5.3.1.3.3 Source Processing Tools .... 71.
5.3.1.3.4 Database Browsing Tools .... 71.
5.3.1.3.5 Debugging Support .... 72.
5.3.1.3.6 Performance Tuning Tools .... 73.
5.3.1.4 Class Library .... 74.
5.3.2 Advanced Topics .... 74.
5.3.2.1 Transaction and Concurrency Model .... .74
5.3.2.2 Relationships & Referential Integrity .... 76.
5.3.2.3 Composite Objects .... 76.
5.3.2.4 Location Transparency .... 76.
5.3.2.5 Object Versioning .... 77.
5.3.2.6 Work Group Support .... 78.
5.3.2.7 Schema Evolution .... 79.
5.3.2.8 Runtime Schema Access/Definition/Modification .... 80.
5.4 Evaluation of ObjectStore 2.0 .... 80.
5.4.1 Application Development Issues .... 81.
5.4.1.1 Developer's View of Persistence .... 81.
5.4.1.2 Application Development Process .... 81.
5.4.1.3 Application Development Tools .... 82.
5.4.1.3.1 Database Administration Tools .... 82.
5.4.1.3.2 Database Design Tools .... 83.
5.4.1.3.3 Source Processing Tools .... 84.
5.4.1.3.4 Database Browsing Tools .... 84.
5.4.1.3.5 Debugging Support .... 85.
5.4.1.3.6 Performance Tuning Tools .... 85.
5.4.1.4 Class Library .... 86.
5.4.2 Advanced Topics .... 87.
5.4.2.1 Transaction and Concurrency Model .... 87.
5.4.2.2 Relationships & Referential Integrity .... 88.
5.4.2.3 Composite Objects .... 89.
5.4.2.4 Location Transparency .... 89.
5.4.2.5 Object Versioning .... 90.
5.4.2.6 Work Group Support .... 91.
5.4.2.7 Schema Evolution .... 91.
5.4.2.8 Runtime Schema Access/Definition/Modification .... 92.
5.5 Evaluation of GemStone Version 3.2 .... 92.
5.5.1 Application Development Issues .... 93.
5.5.1.1 Developer's View of Persistence .... 94.
5.5.1.2 Application Development Process .... 95.
5.5.1.3 Application Development Tools .... 96.
5.5.1.3.1 Database Administration Tools .... 96.
5.5.1.3.2 Database Design Tools .... 97.
5.5.1.3.3 Source Processing Tools .... 97.
5.5.1.3.4 Database Browsing Tools .... 99.
5.5.1.3.5 Debugging Support .... 100.
5.5.1.3.6 Performance Tuning Tools .... 100.

Table of Contents Continued

5.5.1.4 Class Library .... 101.
5.5.2 Advanced Topics .... 101.
5.5.2.1 Transaction and Concurrency Model .... 101.
5.5.2.2 Relationships & Referential Integrity .... 103.
5.5.2.3 Composite Objects .... 103.
5.5.2.4 Location Transparency .... 103.
5.5.2.5 Object Versioning .... 104.
5.5.2.6 Work Group Support .... 104.
5.5.2.7 Schema Evolution .... 104.
5.5.2.8 Runtime Schema Access/Definition/Modification .... 105.
5.6 Evaluation of Object Design ObjectStore PSE Pro .... 106.
5.6.1 Application Development Issues .... 106.
5.6.1.1 Developer's View of Persistence .... 106.
5.6.1.2 Application Development Process .... 106.
5.6.1.3 Application Development Tools .... 106.
5.6.1.3.1 Database Administration Tools .... 106.
5.6.1.3.2 Database Design Tools .... 106.
5.6.1.3.3 Source Processing Tools .... 106.
5.6.1.3.4 Database Browsing Tools .... 107.
5.6.1.3.5 Debugging Support .... 107.
5.6.1.3.6 Performance Tuning Tools .... 107.
5.6.1.4 Class Library .... 107.
5.6.2 Advanced Topics .... 108.
5.6.2.1 Transaction and Concurrency Model .... 108.
5.6.2.2 Relationships and Referential Integrity .... 108.
5.6.2.3 Composite Objects .... 108.
5.6.2.4 Location Transparency .... 108.
5.6.2.5 Object Versioning .... 108.
5.6.2.6 Work Group Support .... 108.
5.6.2.7 Schema Evolution .... 109.
5.6.2.8 Runtime Schema Access/Definition/Modification .... 109.
5.7 IBM San Francisco .... 109.
5.7.1 Application Development Issues .... 109.
5.7.1.1 Developer's View of Persistence .... 110.
5.7.1.2 Application Development Process .... 110.
5.7.1.3 Application Development Tools .... 110.
5.7.1.3.1 Database Administration Tools .... 110.
5.7.1.3.2 Database Design Tools .... 110.
5.7.1.3.3 Source Processing Tools .... 111.
5.7.1.3.4 Database Browsing Tools .... 111.
5.7.1.3.5 Debugging Support .... 111.
5.7.1.3.6 Performance Tuning Tools .... 111.
5.7.1.4 Class Library .... 111.
5.7.2 Advanced Topics .... 113.
5.7.2.1 Transaction and Concurrency Model .... 113.
5.7.2.2 Relationships and Referential Integrity .... 114.
5.7.2.3 Composite Objects .... 114.
5.7.2.4 Location Transparency .... 114.

Table of Contents Continued

5.7.2.5 Object Versioning .... 115.
5.7.2.6 Work Group Support .... 115.
5.7.2.7 Schema Evolution .... 115.
5.7.2.8 Runtime Schema Access/Definition/Modification .... 115.

6. Conclusions ____ 116.

About the Authors ____ 119.

References ____ 120.

Bibliography ____ 123.

Appendix A – Evaluation Survey Template ____ 124.

A.1 Functionality .... 124.

A.1.1 Basic Object-Oriented Modeling .... 124.
A.1.1.1 Complex Objects .... 124.
A.1.1.2 Object Identity .... 124.
A.1.1.3 Classes .... 124.
A.1.1.4 Attributes .... 125.
A.1.1.5 Behaviors .... 125.
A.1.1.6 Encapsulation .... 125.
A.1.1.7 Inheritance .... 125.
A.1.1.8 Overriding Behaviors and Late Binding .... 125.
A.1.1.9 Persistence .... 125.
A.1.1.10 Naming .... 125.
A.1.2 Advanced Object-Oriented Database Topics .... 126.
A.1.2.1 Relationships & Referential Integrity .... 126.
A.1.2.2 Composite Objects .... 126.
A.1.2.3 Location Transparency .... 126.
A.1.2.4 Object Versioning .... 126.
A.1.2.5 Work Group Support .... 126.
A.1.2.6 Schema Evolution .... 127.
A.1.2.7 Runtime Schema Access/Definition/Modification .... 127.
A.1.2.8 Integration with Existing DBs and Applications .... 127.
A.1.3 Database Architecture .... 127.
A.1.3.1 Distributed Client - Server Approach .... 127.
A.1.3.2 Data Access Mechanism .... 127.
A.1.3.3 Object Clustering .... 127.
A.1.3.4 Heterogeneous Operation .... 127.
A.1.4 Database Functionality .... 128.
A.1.4.1 Access to Unlimited Data .... 128.
A.1.4.2 Integrity .... 128.
A.1.4.3 Concurrency .... 128.
A.1.4.4 Recovery .... 128.
A.1.4.5 Transactions .... 128.
A.1.4.6 Deadlock detection .... 129.
A.1.4.7 Locking .... 129.
A.1.4.8 Backup and Restore .... 129.
A.1.4.9 Dump and Load .... 129.

Table of Contents Continued

A.1.4.10 Constraints .... 129.
A.1.4.11 Notification Model .... 129.
A.1.4.12 Indexing .... 129.
A.1.4.13 Storage Reclamation .... 130.
A.1.4.14 Security .... 130.
A.1.5 Application Programming Interface .... 130.
A.1.5.1 DDL/DML Language .... 130.
A.1.5.2 Computational Completeness .... 130.
A.1.5.3 Language Integration Style .... 130.
A.1.5.4 Data Independence .... 130.
A.1.5.5 Standards .... 130.
A.1.6 Querying an OODBMS .... 130.
A.1.6.1 Associative Query Capability .... 130.
A.1.6.2 Data Independence .... 131.
A.1.6.3 Impedance Mismatch .... 131.
A.1.6.4 Query Invocation .... 131.
A.1.6.5 Invocation of Programmed Behaviors .... 131.
A.2 Application Development Issues .... 131.
A.2.1 Developer's View of Persistence .... 131.
A.2.2 Application Development Process .... 131.
A.2.3 Application Development Tools .... 131.
A.2.3.1 Database Administration Tools .... 131.
A.2.3.2 Database Design Tools .... 131.
A.2.3.3 Source Processing Tools .... 131.
A.2.3.4 Database Browsing Tools .... 131.
A.2.3.5 Debugging Support .... 131.
A.2.3.6 Performance Tuning Tools .... 132.
A.2.4 Class Library .... 132.
A.3 Miscellaneous Criteria .... 132.
A.3.1 Product Maturity .... 132.
A.3.2 Product Documentation .... 132.
A.3.3 Vendor Maturity .... 132.
A.3.4 Vendor Training .... 132.
A.3.5 Vendor Support & Consultation .... 132.
A.3.6 Vendor Participation in Standards Activities .... 132.

Appendix B – Schema-Mapping Tools ________ 133.
B.1 Mapping Techniques .... 133.
B.2 Mapping Inheritance Trees .... 133.
B.3 Mapping Object Relationships .... 133.

Table of Contents Continued

List of Figures

Figure 1: The Evolution of Object-Oriented Databases .... 6.

Figure 2: Makeup of an Object-Oriented Database .... 7.

Figure 3: Exploded Product Parts Data Model .... 9.

Figure 4: Example of Object Attributes and Operations .... 9.

List of Tables

Table 3: Evaluation Criteria Overview .... 13.

Table 3.1: Functional Evaluation Criteria .... 15.

Table 3.2: Application Development Evaluation Criteria .... 35.

Table 3.3: Miscellaneous Evaluation Criteria .... 37.

Object-Oriented Database Management Systems Revisited

Abstract and Ordering Information

Abstract

This report reviews the state of the art of Object-Oriented Database Management Systems (OODBMS). The objective of this report is to provide the reader with an understanding of the issues relevant to OODBMS technology and to describe where commercial products stand on these issues. A further objective is to describe the current state of commercial OODBMS in regard to a set of expected facilities for databases in general and object-oriented databases in particular. It is expected that this report will be used as the first step in an evaluation aimed at selecting an OODBMS for use in a given application development effort.

This report describes a broad range of OODBMS evaluation criteria. It then evaluates five commercial OODBMS products in a subset of the overall criteria, focusing on issues relevant to applications development and advanced object-oriented database concepts (e.g., versioning, schema evolution). An appendix to this report defines a template to be used as a guide for performing an evaluation of OODBMS.

Ordering Information:

A bound version of this report, is avaliable for $50 from the DACS Product Orderform on-line or you may order it by contacting:

DACS Customer Liaison

775 Dadaelian Drive

Griffiss Business Park

Rome, NY 13441-4909

(315) 334-4905; Fax: (315) 334-4964;

cust-liasn@dacs.dtic.mil

http://www.dacs.dtic.mil/forms/orderform.shtml

Acknowledgements:

The author would like to gratefully acknowledge comments on an earlier draft by Mr. Robert Vienneau and the help of Mr. Lon R. Dean in producing this report.

1. Introduction

This report provides an update to our report of the state of the art of Object-Oriented Database Management Systems (OODBMS), which was issued in 1993 (McF93). The objective of this report is to provide the reader with an understanding of the issues relevant to OODBMS technology and to describe where commercial products stand on these issues. A further objective is to describe the current state of commercial OODBMS in regard to a set of expected facilities for databases in general and object-oriented databases in particular. It is expected that this report will be used as the first step in an evaluation aimed at selecting an OODBMS for use in a given application development effort. It is essential that any such evaluation be customized to consider application specific requirements. For these reasons it is impossible to use this report as the sole information source when selecting an OODBMS. In particular, it is not a goal of this report to identify the best OODBMS product. Each OODBMS will be architected based on a set of assumptions which make it more or less suited for particular application domains and usage patterns. Thus, we do not expect a single OODBMS to be best in all situations. This report may also be used as an introduction to object-oriented database technology. Additional in-depth information may be found in the references and bibliographies.

An evaluation of OODBMS must include analysis in four areas:

- functionality,
- usability,
- platform, and
- performance.

An analysis of functional capabilities is performed to determine if a given OODBMS provides sufficient capabilities to meet the current and future needs of a given development effort. Functional capabilities include basic database functionality such as concurrency control and recovery as well as object-oriented database features such as inheritance and versioning. Each evaluation will have to identify and weight a set of functional requirements to be met by the candidate OODBMS. Weighting is an important consideration since application workarounds may be possible for missing functionality.

Usability deals with the application development and maintenance process. Issues include development tools and the ease with which database applications can be developed and maintained. How a developer perceives the database and the management of persistent objects might also be considered under the category of usability. Other issues to be considered are database administration, product maturity, and vendor support. Evaluation of usability is likely to be highly subjective.

Perhaps the most easily measurable evaluation criteria is platform. An OODBMS either is or is not available on the application's target hardware and operating system. Heterogeneous target environments require that the OODBMS transparently interoperates within that environment. An OODBMS is typically a multi-processed software system communicating over a local area network. Platforms upon which database server processes, client application processes, additional administration processes (e.g., lock servers), and development tools can be hosted must be considered. Network requirements should also be evaluated.

Performance may represent the most important evaluation criteria. [Lai91], [Rot92], and [Ver92] describe the issues relevant to analyzing the performance of an OODBMS and describe the difficulties in building general purpose benchmarks. The University of Wisconsin has performed a benchmarking of OODBMS, known as the 007 benchmark [Car93]. A general purpose benchmark is only effective in predicting the performance of an OODBMS for an application which closely mirrors the behavior of that benchmark.

An effective benchmark must consider the number of interactive users, the rate of database updates and accesses, the size of the databases, the hardware and network configurations, and the general access patterns of the database applications. Thus, in order to provide useful information, a benchmark must be modeled to closely mimic the expected behavior of the application being developed.

Providing a fair and substantive evaluation of OODBMS is a difficult task. Issues regarding accuracy of marketing information and technical documentation, completeness of implementation, usability of implementation, performance, and feature interaction (regarding completeness, usability, and performance) must be considered when performing the evaluation. [Ste92] proposes that an evaluation begin by a thorough review of a product's technical documentation (instead of marketing information), and be completed by actual use and benchmarking of the products. The objective of this report is to perform the first part of this evaluation process by performing an extensive analysis based on technical product documentation. In particular:

- Functional capabilities have been identified by examination of the product's technical manuals as supplied by the vendor. Discussions with technical representatives of the vendor have been used to clarify our understandings and descriptions of the evaluated products.
- Usability has been derived by analyzing the documentation for the vendor supplied tools and by reviewing the application programming interface in order to understand how an application interacts with the database.
- Information regarding platform and heterogeneous operation has been supplied by the product vendors.
- Performance is not addressed as part of this evaluation.

This report is the result of an effort aimed at evaluating the current state of OODBMS. Readers can use this report as the start of an evaluation process leading to the selection of an OODBMS for a particular application or system development effort. Users of this report must continue the evaluation process by operationally verifying the described capabilities and by developing an application specific benchmark. This report will be considered a success if it helps its readers in performing a fair and objective evaluation of OODBMS products.

The remainder of this report is broken into the following sections:

- Chapter 2, Overview of OODBMS Technology, discusses the reasons for the emergence of OODBMS and the application areas they are expected to benefit most.
- Chapter 3, Evaluation Criteria, is a detailed discussion of evaluation criteria for OODBMS. This section focuses on functionality and usability issues. This section provides the reader an in-depth overview of OODBMS technology.
- Chapter 4, Evaluation Targets, identifies and briefly describes the commercial OODBMS that were evaluated as part of this effort. These products are:
    - Objectivity/DB developed by Objectivity, Inc.
    - ONTOS DB developed by ONTOS, Inc.
    - VERSANT developed by Versant Object Technology Corp.
    - ObjectStore developed by Object Design, Inc.
    - GemStone developed by Servio Corp.
    - ObjectStore PSE Pro by Object Design, Inc.
    - IBM San Francisco

- Chapter 5, Evaluation Reports, is a detailed evaluation of each OODBMS listed in Chapter 4. The evaluations are based on a subset of the criteria identified in Chapter 3, focusing on application development issues and advanced object-oriented database topics.
- Chapter 6, Conclusions, presents some concluding remarks regarding this evaluation report and the current state of OODBMS.
- Appendix A: Evaluation Survey, presents a template that may be used to direct an evaluation of OODBMS. The template was derived from the evaluation criteria presented in Section 3 of this report.
- Appendix B: Schema-mapping Tools, describes the salient mapping techniques and features.

2. Overview of OODBMS Technology

2.1 The Need for Object-Oriented Databases

The increased emphasis on process integration is a driving force for the adoption of object-oriented database systems. For example, the Computer Integrated Manufacturing (CIM) area is focusing heavily on using object-oriented database technology as the process integration framework. Advanced office automation systems use object-oriented database systems to handle hypermedia data. Hospital patient care tracking systems use object-oriented database technologies for ease of use. All of these applications are characterized by having to manage complex, highly interrelated information, which is a strength of object-oriented database systems.

Clearly, relational database technology has failed to handle the needs of complex information systems. The problem with relational database systems is that they require the application developer to force an information model into tables where relationships between entities are defined by values. Mary Loomis, the architect of the Versant OODBMS compares relational and object-oriented databases. "Relational database design is really a process of trying to figure out how to represent real-world objects within the confines of tables in such a way that good performance results and preserving data integrity is possible. Object database design is quite different. For the most part, object database design is a fundamental part of the overall application design process. The object classes used by the programming language are the classes used by the ODBMS. Because their models are consistent, there is no need to transform the program’s object model to something unique for the database manager." [Loo92b]

An initial area of focus by several object-oriented database vendors has been the Computer Aided Design (CAD), Computer Aided Manufacturing (CAM) and Computer Aided Software Engineering (CASE) applications. A primary characteristic of these applications is the need to manage very complex information efficiently. Other areas where object-oriented database technology can be applied include factory and office automation. For example, the manufacture of an aircraft requires the tracking of millions of interdependent parts that may be assembled in different configurations. Object-oriented database systems hold the promise of putting solutions to these complex problems within reach of users.

Object-orientation is yet another step in the quest for expressing solutions to problems in a more natural, easier to understand way. Michael Brodie in his book On Conceptual Modeling states "the fundamental characteristic of the new level of system description is that it is closer to the human conceptualization of a problem domain. Descriptions at this level can enhance communication between system designers, domain experts and, ultimately, system end-users." [Bro84]

The study of database history is centered on the problem of data modeling. "A data model is a collection of mathematically well defined concepts that help one to consider and express the static and dynamic properties of data intensive applications. " [Bro84] A data model consists of:

- static properties such as objects, attributes and relationships,
- integrity rules over objects and operations, and
- dynamic properties such as operations or rules defining new database states based on applied state changes.

Object-oriented databases have the ability to model all three of these components directly within the database supporting a complete problem/solution modeling capability. Prior to object-oriented databases, databases were capable of directly supporting points 1 and 2 above and relied on applications for defining the dynamic properties of the model. The disadvantage of delegating the dynamic properties to applications is that these dynamic properties could not be applied uniformly in all database usage scenarios since they were defined outside of the database in autonomous applications. Object-oriented databases provide a unifying paradigm that allows one to integrate all three aspects of data modeling and to apply them uniformly to all users of the database.

2.2 The Evolution of Object-Oriented Databases

Object-oriented database research and practice dates back to the late 1970's [Loc79] and had become a significant research area by the early 1980's, with initial commercial product offerings appearing in the late 1980's. Today, there are many companies marketing commercial object-oriented databases that are second generation products. The growth in the number of object-oriented database companies has been remarkable. As both the user and vendor communities grow there will be a user pull to mature these products to provide robust data management systems.

OODBMS' have established themselves in niches such as e-commerce, engineering product data management, and special purpose databases in areas such as securities and medicine. The strength of the object model is in applications where there is an underlying need to manage complex relationships among data objects. Today, it's unlikely that OODBMS' are a threat to the stranglehold that relational database vendors have in the market place. Clearly, there is a partitioning of the market into databases that are best suited for handling high volume low, complexity data and databases that are suited for high complexity, reasonable volume, with OODBMS filling the need for the latter.

Object-oriented databases are following a maturation path similar to relational databases. Figure 1 depicts the evolution of object-oriented database technologies. On the left, we have object-oriented languages that have been extended to provide simple persistence allowing application objects to persist between user sessions. Minimal database functionality is provided in terms of concurrency control, transactions, recovery, etc. At the mid-point, we have support for many of the common database features mentioned above. Database products at the mid-point are sufficient for developing reasonably complex data management applications. Finally, database products with declarative semantics have the ability to greatly reduce development efforts, as well to enforce uniformity in the application of these semantics. OODBMS products today are largely in the middle with a few products exhibiting declarative semantics such as constraints, referential integrity rules, and security capabilities. In most OODBMS products, most of the database semantics are defined by programmers using low-level services provided by the database.

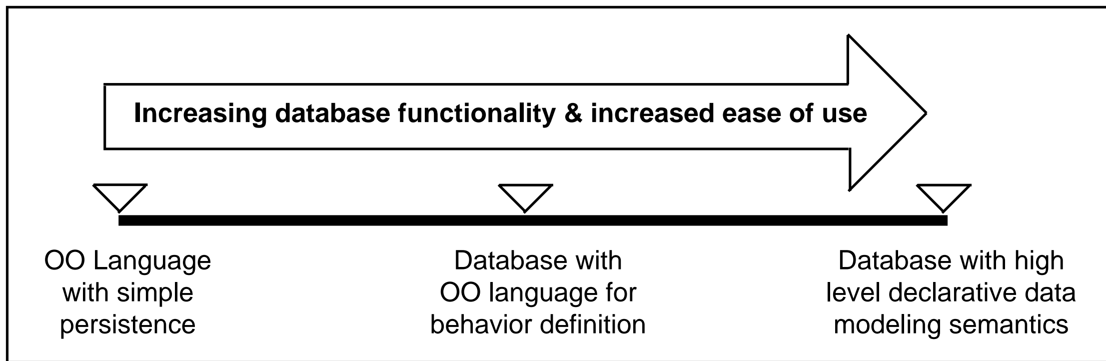

Figure 1. The evolution of object-oriented databases

The next stage of evolution is more difficult. As one moves to the right the database does more for the user requiring less effort to develop applications. An example of this is that current OODBMS provide a large number of low-level interfaces for the purpose of optimizing database access. The onus is entirely on the developer for determining how to optimize his application using these features. As the OODBMS database technology evolves, OODBMS will assume a greater part of the burden for optimization allowing the user to specify high level declarative guidance on what kinds of optimizations need to be performed.

A general guideline for gauging database maturity is the degree to which functions such as database access optimization, integrity rules, schema and database migration, archive, backup and recovery operations can be tailored by the user using high level declarative commands to the OODBMS. Today, most object-oriented database products require the application developer to write code to handle these functions.

Another sign of maturation of a new technology is the establishment of industry groups to standardize on different aspects of technology. Today we see a significant interest in the development of standards for object-oriented databases. For example, the Object Management Group (OMG) is a non-profit industry sponsored association whose goal is to provide a set of standard interfaces for interoperable software components. Interfaces are to be defined in areas of communications (Object Request Broker), object-oriented databases, object-oriented user interfaces, etc. An OODBMS application programmers interface (API) specification is currently being developed (by ODMG, Object Database Management Group, a group of OODBMS vendors) thus allowing portability of applications across OODBMS [Sol92]. Another standards body X3H7, a technical committee under X3, has been formed to define OODBMS standards in areas such as object-models and object-extensions to SQL.

Today, OODBMS vendors are adding more database features to their products to provide the functionality one would expect from a mature database management system. This evolution moves us to the mid-point of the evolutionary scale shown in Figure 1.

2.3 Characteristics of Object-Oriented Databases

Object-oriented database technology is a marriage of object-oriented programming and database technologies. Figure 2 illustrates how these programming and database concepts have come together to provide what we now call object-oriented databases

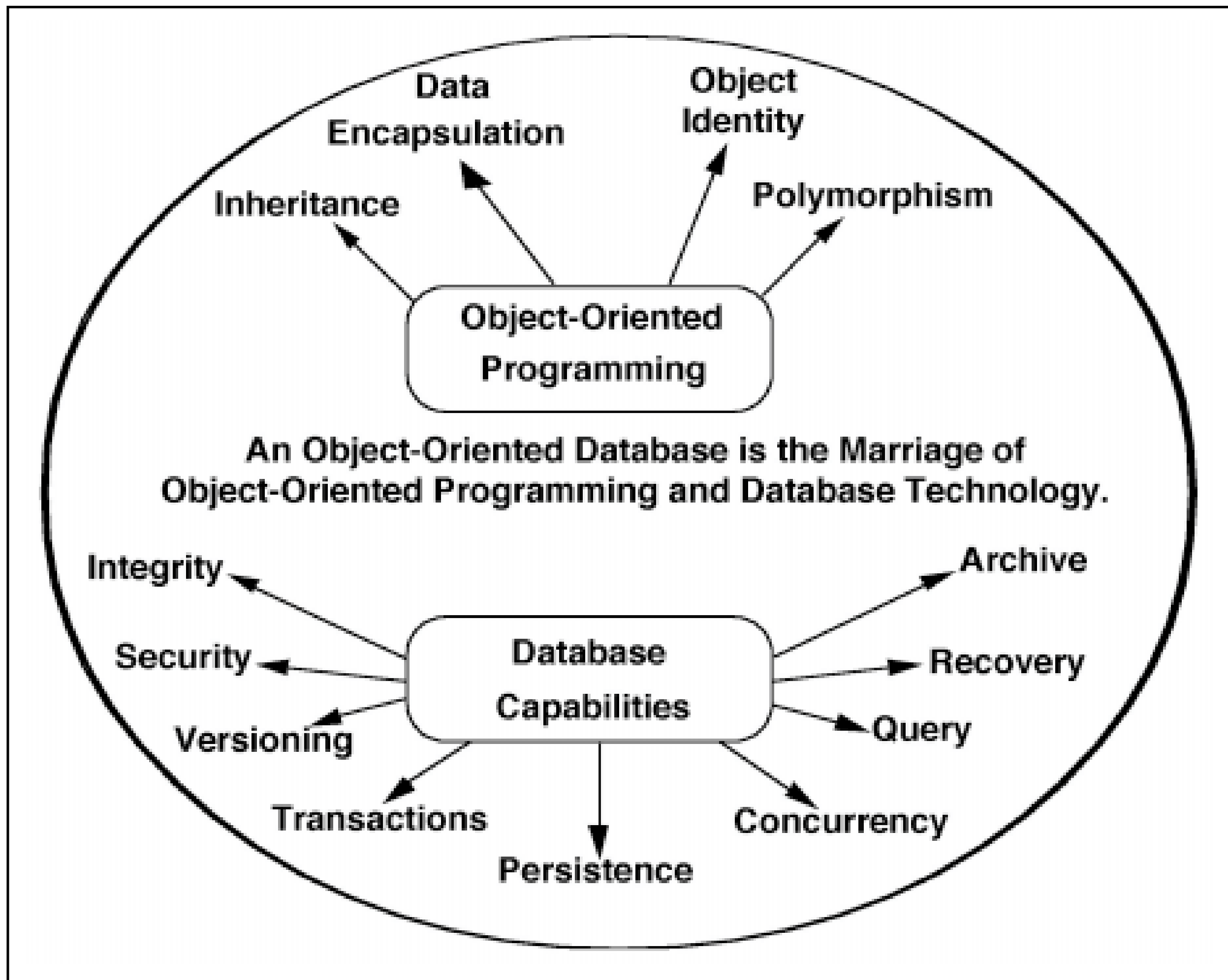

Figure 2. Makeup of an Object-Oriented Database

Perhaps the most significant characteristic of object-oriented database technology is that it combines object-oriented programming with database technology to provide an integrated application development system. There are many advantages to including the definition of operations with the definition of data. First, the defined operations apply ubiquitously and are not dependent on the particular database application running at the moment. Second, the data types can be extended to support complex data such as multi-media by defining new object classes that have operations to support the new kinds of information.

Other strengths of object-oriented modeling are well known. For example, inheritance allows one to develop solutions to complex problems incrementally by defining new objects in terms of previously defined objects. Polymorphism and dynamic binding allow one to define operations for one object and then to share the specification of the operation with other objects. These objects can further extend this operation to provide behaviors that are unique to those objects. Dynamic binding determines at runtime, which of these operations is actually executed, depending on the class of the object requested to perform the operation. Polymorphism and dynamic binding are powerful object-oriented features that allow one to compose objects to provide solutions without having to write code that is specific to each object. All of these capabilities come together synergistically to provide significant productivity advantages to database application developers.

A significant difference between object-oriented databases and relational databases is that object-oriented databases represent relationships explicitly, supporting both navigational and associative access to information. As the complexity of interrelationships between information within the database increases, the greater the advantages of representing relationships explicitly. Another benefit of using explicit relationships is the improvement in data access performance over relational value-based relationships.

A unique characteristic of objects is that they have an identity that is independent of the state of the object. For example, if one has a car object and we remodel the car and change its appearance – – the engine, the transmission, the tires so that it looks entirely different, it would still be recognized as the same object we had originally. Within an object-oriented database, one can always ask the question, is this the same object I had previously, assuming one remembers the object's identity. Object-identity allows objects to be related as well as shared within a distributed computing network.

All of these advantages point to the application of object-oriented databases to information management problems that are characterized by the need to manage:

- a large number of different data types,
- a large number of relationships between the objects, and
- objects with complex behaviors.

Application areas where this kind of complexity exists includes engineering, manufacturing, simulations, office automation and large information systems.

2.4 Application of an OODBMS

The design of an object-oriented data model is the first step in the application of object-oriented databases to a particular problem area. Developing a data model includes the following major steps:

- Object identification
- Object state definition
- Object relationships identification
- Object behavior identification
- Object classification

The following is a cursory overview of these steps.

As one begins to define an object-oriented data model, the first step is to simply observe and record the objects in the solution space. There are many techniques that aid this process. For example, one can formulate a description of the solution and identify the nouns that are candidates for being the objects in the data model. Next, one identifies the characteristics of these objects. These characteristics become the object attributes. In a similar manner, examining the logical dependencies among objects identifies different kinds of association. For example, the parts relationship can be identified by analyzing the system decomposition into subparts. Next, one begins to enumerate the different responses that an object has to different stimuli. Finally, one classifies objects into an inheritance structure to factor out common characteristics and behaviors. All of these steps are performed iteratively until one has a complete data model.

A number of textbooks (e.g., [Coa90], [Wir90]) describe different variations of the above approach. In all cases, these methods culminate in a data model consisting of objects, attributes, relationships, behavior and a classification structure. The methods vary in terms of targeted audience, the level of rigor, and the number and kinds of intermediate steps required to arrive at a data model. Some methods are targeted to

people whose background is structured analysis while other methods appeal to accomplished object-oriented developers. The practitioner has to select the methods that best match his experience and the target application.

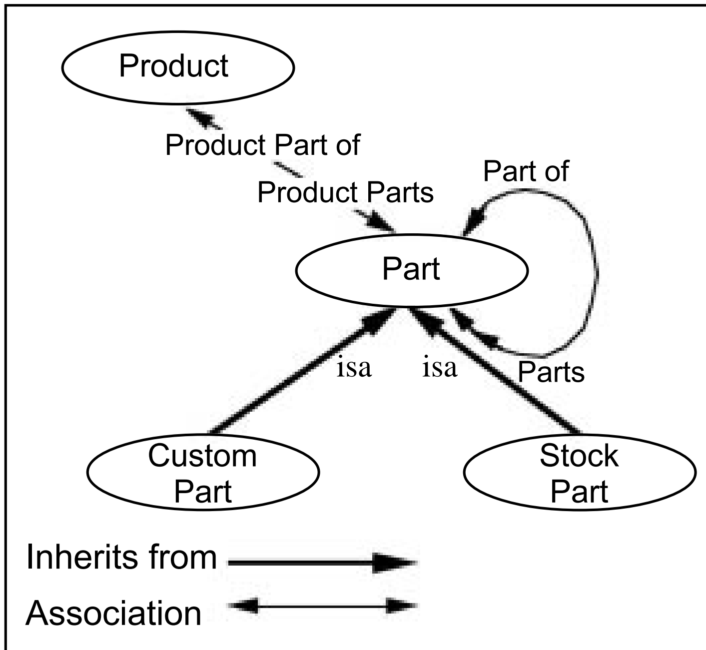

Figure 3. Exploded product parts data model

Figure 3 illustrates a data model for a product and its decomposition into parts. Each part, in turn, may decompose into subparts. These associations are relationships (bi-directional relationships in this example) between objects. In an object-oriented database, relationships are maintained between objects using the object's unique identity, which means that one can change the attribute values of objects and not affect the relationships between the objects.

A significant difference between databases and object-oriented programming languages such as C++, is that databases typically provide high level primitives for defining relationships among objects. Typically, the implementation of relationships is managed by the OODBMS to maintain referential integrity. In addition, the OODBMS may allow one to define relationship cardinality and object existence constraints. Semantic richness of relationships is well suited for the management of complex, highly interrelated information. Unfortunately, these capabilities are not provided uniformly by different object-oriented database products.

An object-oriented data model also defines attributes and operations for each object as shown in Figure 4.

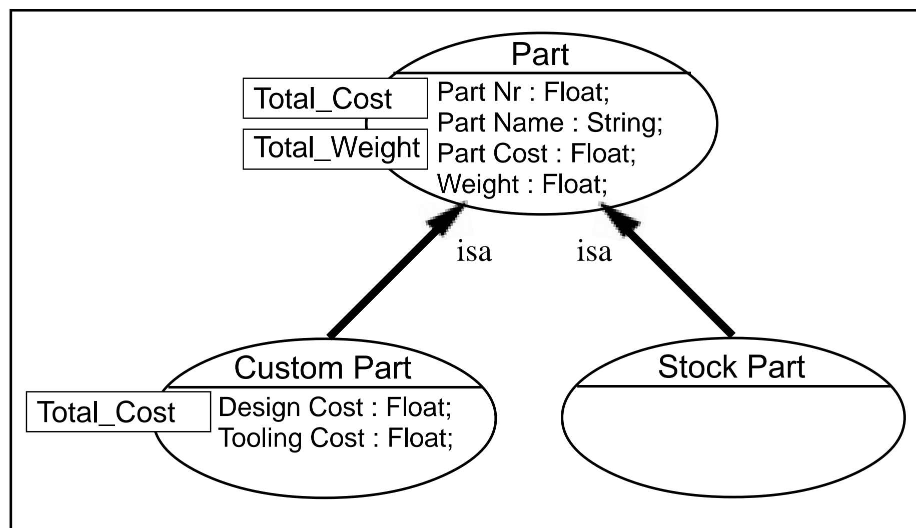

Figure 4. Example of object attributes and operations

In our example, the Custom Part and Stock Part inherit the attributes and operations from Part. The Custom Part object defines additional attributes and overrides the Total Cost operation. A significant advantage of inheritance is that it allows one to generalize objects by factoring common attributes and operations into some common object and then, using inheritance, share these common properties. As one adds more objects, relationships, and operations, inheritance helps to reduce the complexity of defining and maintaining this information.

In the real world, the data model is not static and will change as organizational information needs change and as missing information is identified. Consequently, the definition of objects must be changed periodically and existing databases migrated to conform to the new object definitions. Object-oriented databases are semantically rich introducing a number of challenges when changing object definitions and migrating databases. Object-oriented databases have a greater challenge handling schema migration because it's not sufficient to simply migrate the data representation to conform to the changes in class specifications. One must also update the behavioral code associated with each object. Improved facilities to manage this kind of change are appearing in a number of products, making it easier to maintain OODBMS-based solutions over time.

2.5 Pragmatics of Using an OODBMS

One needs to weigh a broad range of issues when considering an object-oriented database as a solution to an information management problem. These issues include: object models, data modeling tools, application design and development tools, testing and debugging tools, monitoring and tuning tools, and database maintenance tools. A database application, like any software system, has a life cycle and requires a complete set of life cycle support tools. The following is a brief overview of the kinds of capabilities one needs to look for in these areas.

First, one needs to construct an object model of the information problem to be solved by the database. With current object-oriented databases, there are significant variations in the modeling capabilities of these products. For example, relationships in some databases are supported by high level declarative capabilities that allow one to define self-maintaining properties of relationships. In other database products, one must program the semantics of relationships explicitly. Some database products support a significantly richer data model providing powerful data management services that are defined as part of the data model. For example, one may define the existence of some objects to be dependent on a particular relationship. When one retracts the relationship, then the related object is also deleted. Other examples of more advanced database semantics include relationship cardinality constraints, attribute value constraints, uniqueness properties of attribute values, initial and default values, object versioning, and composite objects. All of these data model issues affect how easily one can define their data model using tools provided by the database vendor. Another benefit is that the richer the data modeling facilities the less work that is required to implement the database application.

An area that many OODBMS evaluators overlook is the tools to support the development of the database applications. Since an OODBMS includes a language for specifying object behaviors, one needs to understand how such behaviors are developed and tested for a given OODBMS. Testing is particularly important since one needs to integrate the compiler testing and debugging tools with the database persistent object storage manager.

During database maintenance, one often needs to introduce incremental changes into the database and the database applications. This is one of the most difficult areas since existing objects must be migrated to a new state conforming to the new schema definitions. This is a common problem with all database applications and needs to be anticipated in the design of the database and its applications. Today most OODBMS provide low level services for migrating databases, making the process of changing the schema a major challenge for database developers.

Finally, the problem of optimizing a database implementation for a particular application is a difficult one. First, there is little experience with OODBMS on which to base optimization strategies. Second, many of the optimization features are at a low level, requiring the application developer to design the application around these features. Both database instrumentation and monitoring tools are needed, as well as facilities for tuning an existing application. As with relational databases, object-oriented database systems also require extensive monitoring and tuning to extract the maximum performance for a given application.

2.6 Summary

The main objective of an OODBMS is to provide consistent, data independent, secure, controlled and extensible data management services to support the object-oriented modeling paradigm. Today's OODBMS provide most of these capabilities. Many of these products are second generation OODBMS that have incorporated the lessons learned from the first generation products. Interpreting the database evolution diagram in Figure 1, we are about half way along the path to having feature rich, powerful OODBMS in the market place. Given the high degree of interest in object-oriented technologies, there is a substantial market pull to put OODBMS products on a fast track where features and capabilities will continue to advance at a rapid rate.

A major strength of the OODBMS technology is its ability to represent complex behaviors directly. By incorporating behaviors into the database, one substantially reduces the complexity of applications that use the database. In the ideal scenario, most of the application code will deal with data entry and data display. All the functionality associated with data integrity and data management would be defined within the basic object model. The advantages of this approach are:

- all operations are defined once and reused by all applications, and
- changes to an operation affect all applications, simplifying database maintenance (although most databases require the applications to be recompiled).

The benefits of object-oriented database applications development are an increase in productivity resulting from the high degree of code reuse and an ability to cope with greater complexity resulting from incremental refinement of problems. One also gets increased design flexibility due to polymorphism and dynamic binding. Finally, both developers and users will experience benefits resulting from the naturalness and simplicity of representing data as objects.

These strengths need to be weighed against the organizational changes introduced by this new and different way of engineering solutions. Different engineering considerations contribute to performance and reliability than for relational DBMS's. Projects need to be managed differently. Clearly, one needs to approach this new technology with eyes open, recognizing that the benefits will be realized after a considerable investment has been made to learn how to use it effectively.

3. Evaluation Criteria

This section is a detailed discussion of evaluation criteria that may be considered when evaluating OODBMS. These criteria are broken into three main areas:

- o 3.1 Functionality,
- o 3.2 Application Development Issues, and
- o 3.3 Miscellaneous Criteria.

Section 3.1, Functionality, defines evaluation criteria based on functional capabilities provided by the OODBMS. Subsections include Basic Object-Oriented Modeling, Advanced Object-Oriented Database Topics, Database Architecture, Database Functionality, Application Programming Interface, and Querying an OODBMS. The topics discussed in the Basic Object-Oriented Modeling subsection are those functions that are expected to be similar in all OODBMS products, such as the ability to define complex objects using classes, attributes, and behaviors. In fact, these topics are the object-oriented features found in most OO technologies (e.g., languages, design methods). The Advanced Object-Oriented Database Topics subsection describes functionality that is somewhat unique to object-oriented databases. It is expected that OODBMS products will differ significantly in these areas. The Database Functionality subsection describes the features that distinguish a database from a persistent storage manager (e.g., concurrency control and recovery).

Section 3.2, Application Development Issues, considers issues regarding the development of applications on top of an OODBMS. Section 3.3, Miscellaneous Criteria, identifies a few non-functional and non-developmental evaluation issues. These issues deal with vendor and product maturity, vendor support, and current users of the OODBMS product.

Although the evaluation criteria identified in this chapter will be an important part of any OODBMS selection process, the issues of platform and performance will most likely dominate the selection process. Although missing functionality can often be managed at the application level, inadequate performance cannot be overcome (assuming optimal use of database facilities and features). The issue of platform and performance as evaluation criteria will not be further addressed in this chapter.

The list of evaluation criteria defined in this chapter is quite extensive. This list was developed as a means of covering the spectrum of issues that might merit consideration during an OODBMS evaluation task. It is not expected that any OODBMS evaluation effort would attempt to consider all of the listed criteria. Instead, an evaluation effort must select the criteria relevant to a particular set of application requirements. Table 3, Evaluation Criteria Overview, is a road map of the criteria evaluation categories that are described in the remainder of this section.

Table 3. Evaluation Criteria Overview

Evaluation Area	Criteria Categories
3.1 Functionality	3.1.1 Basic Object-Oriented Modeling 3.1.2 Advanced Object-Oriented Database Topics 3.1.3 Database Architecture 3.1.4 Database Functionality 3.1.5 Application Programming Interface 3.1.6 Querying an OODBMS
3.2 Application Development Issues	3.2.1 Developer's View of Persistence 3.2.2 Application Development Process 3.2.3 Application Development Tools 3.2.4 Class Library
3.3 Miscellaneous Criteria	3.3.1 Product Maturity 3.3.2 Product Documentation 3.3.3 Vendor Maturity 3.3.4 Vendor Training 3.3.5 Vendor Support & Consultation 3.3.6 Vendor Participation in Standards Activities

3.1 Functionality

In the process of identifying functional evaluation criteria, we provide an overview of OODBMS capabilities that are typically found in commercial products. The list of such capabilities was derived from the many books, reports, and articles that have appeared in the literature over the past few years. We have specifically avoided evaluation criteria checklists that might be considered partial to one particular vendor (e.g., [Bar93] provides a detailed evaluation checklist which was developed while the author was at ITASCA Systems, [Ver92] is an evaluation guide published by Versant).

This section not only identifies the functionality evaluation criteria but also provides a high-level overview of DBMS and OODBMS concepts. Readers already familiar with these concepts can skip this section. Readers new to these technologies can gain an overview of the issues from this section and are referred to the numerous references and bibliographies listed at the end of this report. In particular [Bro91], [Cat91], [Ban92], [Kim90], and [Par89] are texts on object-oriented and intelligent databases.

Evaluation criteria for this section are broken into the following subsections:

- o Basic Object-Oriented Modeling
- o Advanced Object-Oriented Database Topics
- o Database Architecture
- o Database Functionality
- o Application Programming Interface
- o Query Capabilities

Table 3.1, Functional Evaluation Criteria, is a road map of the categories and specific topics covered in the functional evaluation criteria.

3.1.1 Basic Object-Oriented Modeling

The evaluation criteria in this section distinguish a database as an object-oriented database. The selected criteria are taken from [Atk92], [Cat91], and [Lec92]. Topics in this section cover the basic object-oriented (OO) capabilities typically supported in any OO technology (e.g., programming language, design method). These basic capabilities are expected to be supported in all commercial OODBMS. The topics are given a cursory overview here for readers new to OO technology.

3.1.1.1 Complex Objects

OO systems and applications are unique in that the information being maintained is organized in terms of the real-world entities being modeled. This differs from relational database applications that require a translation from the real-world information structure to the table formats used to store data in a relational database. Normalizations upon the relational database tables result in further perturbation of the data from the user's perceptual viewpoint. OO systems provide the concept of complex objects to enable modeling of real-world entities. A complex object contains an arbitrary number of fields, each storing atomic data values or references to other objects (of arbitrary types). A complex object exactly models the user perception of some real-world entity.

Table 3.1. Functional Evaluation Criteria

Criteria Categories	Criteria
Basic Object-Oriented Modeling	• Complex Objects • Object Identity • Classes • Attributes • Behaviors • Encapsulation • Inheritance • Overriding Behaviors and Late Binding • Persistence • Naming
Advanced Object-Oriented Database Topics	• Relationships & Referential Integrity • Composite Objects • Location Transparency • Object Versioning • Work Group Support • Schema Evolution • Runtime Schema Access/Definition/Modification • Integration with Existing DBs and Applications • Active vs. Passive Object Mgmt. System
Database Architecture	• Distributed Client-Server Approach • Data Access Mechanism • Object Clustering • Heterogeneous Operation
Database Functionality	• Access to Unlimited Data • Integrity • Concurrency • Recovery • Transactions • Deadlock Detection • Locking • Backup and Restore • Dump and Load • Constraints • Notification Model • Indexing • Storage Reclamation • Security
Application Programming Interface	• DDL/DML Language • Computational Completeness • Language Integration Style • Data Independence • Standards
Querying an OODBMS	• Associative Query Capability • Data Independence • Impedance Mismatch • Query Invocation • Invocation of Programmed Behaviors

3.1.1.2 Object Identity

OO databases (and programming languages) provide the concept of an object identifier (OID) as a means of uniquely identifying a particular object. OIDs are system generated. A database application does not have direct access to the OID. The OID of an object never changes, even across application executions. The OID is not based on the value stored within the object. This differs from relational databases, which use the concept of primary keys to identify a particular table row (i.e., tuple). Primary keys are based upon data stored in the identified row. The concept of OIDs makes it easier to control the storage of objects (e.g., not based on value) and to build links between objects (e.g., they are based on the never changing OID). Complex objects often include references to other objects, directly or indirectly stored as OIDs.

The size of an OID can substantially affect the overall database size due to the large number of inter-object references typically found within an OO application.

When an object is deleted, its OID may or may not be reused. Reuse of OIDs reduces the chance of running out of unique OIDs but introduces the potential for invalid object access due to dangling references. A dangling reference occurs if an object is deleted, and some other object retains the deleted object's OID, typically as an inter-object reference. This second object may later use the OID of the deleted object with unpredictable results. The OID may be marked as invalid or may have been re-assigned. Typically, an OODBMS will provide mechanisms to ensure dangling references between objects are avoided (see Section 3.1.2.1, Relationships & Referential Integrity).

3.1.1.3 Classes

OO modeling is based on the concept of a class. A class defines the data values stored by, and the functionality associated with, an object of that class. One of the primary advantages of OO data modeling is this tight integration of data and behavior through the class mechanism.

Each object belongs to one, and only one, class. An object is often referred to as an instance of a class. A class specification provides the external view of the instances of that class. A class has an extent (sometimes called an extension), which is the set of all instances of the class [Cat91]. Implementation of the extent may be transparent to an application, but minimally provides the ability to visit every instance of the class.

Within an OODBMS, the class construct is normally used to define the database schema. Some OODBMS use the term type instead of class. The OODBMS schema defines what objects may be stored within the database.

3.1.1.4 Attributes

Attributes represent data components that make up the content of a class. Attributes are called data members in the C++ programming language. Instance attributes are data components that are stored by each instance of the class. Class attributes (static data members in C++) are data values stored once for all instances of the class. Attributes may or may not be visible to external users of the class (see Sections 3.1.1.6, Encapsulation, and 3.1.5.4, Data Independence).

Attribute types are typically a subset of the basic data types supported by the programming language that interfaces to the OODBMS. Typically this includes enumeration types such as characters and booleans, numeric types such as integers and floats, and fixed length arrays of these types such as strings. The OODBMS may allow variable length arrays, structures (i.e., records) and classes as attribute types.

Pointers are normally not good candidates for attribute types since pointer values are not valid across application executions.

An OODBMS will provide attribute types that support inter-object references (see Section 3.1.2.1, Relationships & Referential Integrity). OO applications are characterized by a network of inter-connected objects. Object inter-connections are supported by attributes that reference other objects.

Other types that might be supported by an OODBMS include text, graphic, and audio. Often these data types are referred to as binary large objects (BLOBS).

Derived attributes are attributes that are not explicitly stored but instead calculated on demand. Derived attributes require that attribute access be in-distinguishable from behavior invocation (see Section 3.1.5.4, Data Independence).

3.1.1.5 Behaviors

Behaviors represent the functional component of a class. A behavior describes how an object operates upon its attributes and how it interacts with other related objects. Behaviors are called member functions in the C++ programming language. Behaviors hide their implementation details from users of a class (see Section 3.1.1.6, Encapsulation).

3.1.1.6 Encapsulation

Classes are said to encapsulate the attributes and behaviors of their instances. Behavior encapsulation shields the clients of a class (i.e., applications or other classes) from seeing the internal implementation of a behavior. This shielding provides a degree of data independence so that clients need not be modified when behavior implementations are modified (they will have to be modified if behavior interfaces change).

A class's attributes may or may not be encapsulated. Attributes that are directly accessible to clients of a class are not encapsulated (public data members in C++ classes). Modifying the definition of a class's attributes that are not encapsulated requires modification of all clients that access them. Attributes that are not accessible to the clients of a class are encapsulated (private or protected data members in C++ classes). Encapsulated attributes typically have behaviors that provide clients some form of access to the attribute. Modifications to these attributes typically do not require modification to clients of the class.

3.1.1.7 Inheritance

Inheritance allows one class to incorporate the attributes and behaviors of one or more other classes. A subclass is said to inherit from one or more superclasses. The subclass is a specialization of the superclass in that it adds additional data or behaviors, or overrides behaviors of the superclass. Superclasses are generalizations of their subclasses. Inheritance is recursive. A class inherits the attributes and behaviors from its superclasses, and from its superclass's superclasses, etc.

In a single inheritance model, a class may directly inherit from only a single other class. In a multiple inheritance model a class may directly inherit from more than one other class. Systems supporting multiple inheritance must specify how inheritance conflicts are handled. Inheritance conflicts are attributes or behaviors with the same name in a class and its superclass, or in two superclasses.

Inheritance is a powerful OO modeling concept that supports reuse and extensibility of existing classes. The inheritance relationships between a group of classes define a class hierarchy. Class hierarchies improve the ability of users to understanding software systems by allowing knowledge of one class (a superclass) to be applicable to other classes (its subclasses).

3.1.1.8 Overriding Behaviors and Late Binding

OO applications are typically structured to perform work on generic classes (e.g., a vehicle) and at runtime invoke behaviors appropriate for the specific vehicle being executed upon (e.g., Boeing 747). Applications constructed in such a manner are more easily maintained and extended since additional vehicle classes may be added without requiring modification of application code.

Overriding behaviors is the ability for each class to define the functionality unique to itself for a given behavior. Late binding is the ability for behavior invocation to be selected at runtime based on the class of an object (instead of at compile time).

3.1.1.9 Persistence

Persistence is the characteristic that makes data available across executions. The objective of an OODBMS is to make objects persistent.

Persistence may be based on an object's class, meaning that all objects of a given class are persistent. Each object of a persistent class is automatically made persistent. An alternative model is that persistence is a unique characteristic of each object (i.e., it is orthogonal to class). Under this model, an object's persistence is normally specified when it is created. A third persistence model is that any object reachable from a persistent object is also persistent. Such systems require some way of explicitly stating that a given object is persistent (as a means of starting the network of inter-connected persistent objects).

Related to the concept of persistence is object existence. OODBMS may provide a means by which objects are explicitly deleted. Such systems must ensure that references to deleted objects are also removed (see Section 3.1.2.1, Relationships & Referential Integrity). An alternative strategy is to maintain an object as long as references to the object exist. Once all references are removed, the object can be safely deleted.

3.1.1.10 Naming

OO applications are characterized as being composed of a network of inter-connected objects. An application begins by accessing a few known objects and then traverses to additional objects via relationships from the known objects. As objects are created they are linked (i.e., related) to other existing objects. Given this scenario, the database must provide some mechanism for identifying one or more objects at application start-up without using relations from existing objects. This is typically accomplished by allowing objects to be named and providing a retrieval mechanism based upon name. An application begins by loading one or two 'high-level' objects that it knows by name and then traverses to other reachable objects.

Object names apply within some name scope. Within a given scope, names must be unique (i.e., the same name can not refer to two objects). The simplest scope model is for the entire database to act as a single name scope. An alternative scope model is for the application to identify name scopes. Using multiple name scopes will reduce the chance for name conflicts.

3.1.2 Advanced Object-Oriented Database Topics

Functional capabilities identified in this section are those that are somewhat unique to object-oriented database systems. We expect that these topics represent the most interesting evaluation topics and will provide the greatest diversity among the evaluated OODBMS.

3.1.2.1 Relationships & Referential Integrity

Relationships are an essential component of the object-oriented modeling paradigm. Relationships allow objects to refer to each other and result in networks of inter-connected objects. Relationships are the paths used to perform navigation-based data access typical of programmed functionality. The ability to directly and efficiently model relationships is one of the major improvements of the object-oriented data model over the relational data model. (Note: Critics claim this reduces data independence by relying on the existence of specific relationships and indexes. See Section 3.1.6, Querying an OODBMS.)

Conceptually, relationships can be thought of as abstract entities that allow objects to reference each other. An OODBMS may choose to represent relationships as attributes of the class (from which the relationships emanate), as independent objects (in which case relationships may be extensible and allow attributes to be added to a relationship), or as hidden data structures attached to the owning object in some fashion.

Relationships are often referred to as references, associations, or links. Sometimes the term relation is used to mean the schema definition of the potential for inter-connections between objects, and the term relationship is used to mean actual occurrences of an inter-connection between objects. In this document we will use the term relationship interchangeably for both the schema definition and the object level existence of connections between objects.

Relationships can be characterized by a number of different independent parameters, leading to a large number of different relationship behaviors:

- Relationships may be uni-directional or bi-directional. A uni-directional relationship exists in only a single direction, allowing a traversal from one object to another but no traversal in the reverse direction. A bi-directional relationship allows traversal in both directions. When a relationship is established along a bi-directional relationship, that relationship is automatically created in both directions (i.e., the application explicitly creates the relationship in one direction and the OODBMS implicitly sets the relationship in the opposite direction).
- Relationships have a cardinality, typically either one-to-one, one-to-many, or many-to-many. A one-to-one relationship allows one object to be related to one other object (e.g., spouse might typically be modeled as a one-to-one relationship). Setting a one-to-one relationship deletes any previously existing relationship. A one-to-many relationship allows a single object to be related to many objects in one direction, in the reverse direction an object may be related to only a single object (e.g., if modeling a house, the house might be composed of many rooms, each room is part of only a single house). One-to-one and one-to-many relationships may be uni-directional or bi-directional. A many-to-many relationship, which must be bi-directional, allows each object to be related to many objects in both directions of the relationship (e.g., modeling the relationship between parents and children might use a many-to-many, bi-directional relationship. A person may have many children; children may have more than one parent.).
- Relationships may have ordering semantics. Ordered relationships are typically considered lists (the objects are ordered by the operations that build the relations, not by the values stored in the related objects, a la indexes). Unordered relationships are either sets or bags. Sets do not allow duplication; bags do.
- Relationships may support the concept of composite objects. See Section 3.1.2.2, Composite Objects.

The existence of relationships gives rise to the need for referential integrity. Referential integrity ensures that objects do not contain references to deleted objects. Referential integrity can be automatically provided for bi-directional relationships. Given a bi-directional relationship, when an object is deleted, all related objects can be found and have their relationships to the deleted object removed. Uni-directional relationships can not be assured of referential integrity (short of performing complete scans of the database). If an object which is the target of a uni-directional relationship is deleted, there is no efficient mechanism to identify all objects that reference the deleted object (and delete the relationships to the deleted object). Application level solutions to this problem exist (e.g., maintenance of existence dependency lists), but may result in poor performance and are effectively duplicating much of the work done by bi-directional relationships. Alternatively, if the OODBMS does not reuse object identifiers, then a deleted object may be tombstoned, meaning a mark is left denoting that the object has been deleted. When a reference to a deleted object is made it can then be trapped as an error, or simply ignored, with an appropriate update of the referencing object's relationship (to no longer relate to the deleted object). Some OODBMS products provide a similar capability by keeping reference counts to objects, and only deleting an object when all references have been removed.

Relationship implementation [see Cat91] provides a major differentiator between OODBMS products. On disk, relationships are typically modeled using object identifiers. Once brought into memory, relationships may remain as object identifiers or be swizzled into virtual memory pointers. Swizzling is a process that converts disk-based relationship representations into memory pointers for all related objects that are in memory. This may be done on object load or on demand when a particular relationship is traversed. Swizzling trades the overhead of performing the conversion to a memory-based pointer traversal in the hopes that multiple future accesses across the relationship will result in an overall speed improvement. Systems that do not swizzle require an object identifier lookup for all relationship traversal. This lookup can be performed efficiently through the use of lookup tables. Hybrid approaches are also possible. Each application will have to consider its expected object access and relationship traversal patterns to determine if a swizzled or object identifier-based relationship approach will best suit its needs.

3.1.2.2 Composite Objects

Composite objects are groupings of inter-related objects that can be viewed logically as a single object. Composite objects are typically used to model relationships that have the semantic meaning is-part-of. (e.g., rooms are part of a house). Composite objects are connected by the relationship mechanisms provided by the OODBMS. Operations applied to the 'root' object of such a grouping can be propagated to all objects within that group. Operations that might be applied on composite objects include:

- equality,
- copy,
- delete, and
- lock.

[Sar92] defines Identifier-Equality, Shallow-Equality, and Deep-Equality operations. These three different forms of equality checks compare object identifiers, attribute values, and attribute values of component objects, respectively. Also defined are Shallow-Copy and Deep-Copy operations. A Shallow-Copy makes a new object and copies attribute values. A Deep-Copy makes a new object, copying non-relationship attribute values, and then recursively creates new objects for related objects (recursively applying the Deep-Copy operation). Deep-Copy is an example of an operation being propagated across a composite object. Propagation of delete and lock operations means that if the root object is deleted or locked all of its component objects are also deleted or locked.

3.1.2.3 Location Transparency

Location transparency is the concept that an object can be referenced (i.e., can use the same syntactic mechanism) regardless of what database it resides in and where on the network that database is located. Objects should be able to be moved programmatically and have all references to the object remain intact (a form of referential integrity). (The ability to move an object to a new database location will also be considered as part of database administration capabilities.)

3.1.2.4 Object Versioning

Object versioning is the concept that a single object may be represented by multiple versions (i.e., instances of that object) at one time [Loo92a]. [Col92] defines two forms of versioning, each driven by particular requirements of the applications which are driving the need for OODBMS products:

- Linear Versioning is the concept of saving prior versions of objects as an object changes. In design-type applications (e.g., CASE, CAD) prior versions of objects are essential to maintain the historical progression of a design and to allow designers to return to earlier design points after investigating and possibly discarding a given design path. Under linear versioning, only a single new version can be created from each existing version of an object.
- Branch Versioning supports concurrency policies where multiple users may update the same data concurrently. Each user's work is based upon a consistent, non-changing base version. Each user can modify his version of an object (as he proceeds along some design path in a CAD application for example). At some future point in time, under user/application support, the multiple branch versions are merged to form a single version of the object. Branch versioning is important in applications with long transactions so that users are not prevented access to information for long periods of time. Under branch versioning, multiple new versions may be created for an object.

Associated with the idea of versioning is that of configuration. A configuration is a set of object versions that are consistent with each other. In other words, it is a group of objects whose versions all belong together. OODBMS need to provide support so that applications access object versions that belong to the same conceptual configuration. This may be achieved by controlling the relationships that are formed for versioned objects (i.e., they may be duplicated in the new object or replaced with relationships to other objects).

An OODBMS may provide low level facilities which application developers use to control the versioning of objects. Alternatively, the OODBMS may implement a specific versioning policy such as automatically creating a new object version with each change. An automatic policy may result in rapid and unacceptable expansion of the database and requires some automated means of controlling this growth [Cel92].

[Cel92] has suggested versioning of databases instead of objects. Within a database, objects are not versioned and can be changed at will. An application can perform versioning by defining new databases in which separate work may proceed. Under the described approach, each new database is linked to the database from which it was derived and stores only objects that have been changed.

3.1.2.5 Work Group Support

In addition to versioning, an OODBMS might support group applications in other manners. The ability to designate shared and private databases with the concept of checking data in and out of these data spaces. Some databases may allow segments of the database to be taken off-line, perhaps on a portable computer,

used autonomously, and then brought back on-line at a later time. Long transactions are another mechanism on top of which group applications can be built (see Section 3.1.4.5, Transactions).

3.1.2.6 Schema Evolution

Schema evolution is the process by which existing database objects are brought into line with changes to the class definitions of those objects (i.e., schema changes require all instances of the changed class to be modified so as to reflect the modified class structure). Schema evolution is helpful although not essential during system development (as a means of retaining test data, for example). Schema evolution is essential for maintenance and/or upgrades of fielded applications. Once an application is in the field (and users are creating large quantities of information), an upgrade or bug fix can not require disposal of all existing user databases. Schema evolution is also essential for applications that support user-level modeling and/or extension of the application (see Section 3.1.2.7, Runtime Schema Access/Definition/Modification).

[Ban87] has defined a framework for schema modifications in an object-oriented database. Included in this framework are invariants which must be maintained at all times (e.g., all attributes of a class must have a distinct name), rules for performing schema modifications (e.g., changing the type of an attribute in a given class must change the type of that attribute in all classes which inherit that attribute), and a set of schema changes that should be supported by an object-oriented database. This set of schema change operations, modified from [Ban87] to use terminology of this report, is:

- oo Changes to Definition of a Class:
  - o Changes to an Attribute of a Class (applies to both instance and class attributes):
  - Add an attribute to a class.
  - Remove an attribute from a class.
  - Change the name of an attribute.
  - Change the type of an attribute.
  - Change the default value of an attribute.
  - Alter the relationship properties for relationship attributes.
  - o Changes to a Behavior of a Class:
  - Add a new behavior to the class.
  - Remove a behavior from the class.
  - Change the name of a behavior.
  - Change the implementation of a behavior.
- oo Changes to the Inheritance of a Class:
  - o Add a new superclass to a class.
  - o Remove a superclass for a given class.
  - o Change the order of superclasses for a class (it is expected that superclass ordering will be used to handle attribute and behavior inheritance conflicts).
- oo Changes to the Existence of a Class:
  - o Add a new class.
  - o Remove an existing class.
  - o Change the name of a class.

Schema changes will require modification of instances of the changed class as well as applications that referenced the changed class. Some of these changes cannot be performed automatically by the OODBMS. Deleting attributes and superclasses are examples of schema changes that could be performed automatically. Adding attributes and superclasses can only be performed if default values are acceptable for the initial state of new attributes. This is not likely, especially for relationship attributes. An OODBMS should provide tools and/or support routines for aiding programmed schema evolution.

A manual evolution approach requires instance migration to be performed off-line, probably through a dump of the database and a reload of the data through an appropriate transformation filter.

Systems may perform an aggressive update by automatically adjusting each instance after each schema change. This approach may be slow due to the overhead of performing the update on all instances at a single time. This approach is the easiest for an application to implement since multiple versions of the schema need not be maintained indefinitely. (Note: although this approach is considered to be the easiest to implement, it is by no means expected to be an easy task!)

Schema changes may be performed in background mode, thus spreading the update overhead over a longer period of time. A lazy evaluation approach defers updating objects until they are accessed and found to be in an inconsistent state. Both the background and lazy approaches require extended periods where multiple versions of the schema exist and will be complicated by multiple schema modifications.

Applications and stored queries will have to be updated manually as a result of schema changes. Some forms of schema changes will not require updates to applications and queries due to data independence and encapsulation of a class's data members.

It is expected that all OODBMS products will support some form of schema evolution for static schema changes. By static, we mean the schema is changed by manipulations of class definitions outside of application processing (i.e., by reprocessing database schema definitions and modifying application programs). Dynamic schema modification, meaning modification of the schema by the application, is more complex and potentially inconsistent with the basic C++ header file approach used for schema definitions in many current commercial products. Dynamic schema modification is only needed in applications that require user definable types. See Section 3.1.2.7, Runtime Schema Access/Definition/ Modification.

3.1.2.7 Runtime Schema Access/Definition/Modification

An OODBMS typically makes use of a database resident representation of the schema for the database. The existence of such a representation, and a means of accessing it, provide applications with direct access to schema information. Access to schema information might be useful in building custom tools which browse the structure and contents of a database (browsing tools are typically provided with the OODBMS; see Section 3.2.3.4, Database Browsing Tools). Modification and definition of the schema at runtime allows the development of dynamically extensible applications. The authors of this report have constructed a custom object-oriented modeling system in which class definitions are created and modified dynamically [Rud93]. Users of this modeling system define the classes, attributes, and behaviors of the information that they wish to model. Once defined, instances of these classes may be created and manipulated.

The idea of dynamic schema definition is foreign to the C++ programming language. In C++, class definitions are defined statically in header files. An application may not alter these class definitions at runtime. OODBMS can provide access and modification of their schemata by storing the schema as

instances of predefined classes and then allowing applications to create, modify, and query the instances which model the schema. An application might wish to modify the schema in order to extend an application to store new information or to display information in alternative presentations.

3.1.2.8 Integration with Existing DBs and Applications

Numerous papers have appeared in the literature describing the need for integration of object-oriented and relational database technologies [Loo92c] [Loo92d] [Ras92] [Vas93]. Newly developed object-oriented applications will need to access existing relational databases. Data stored in object-oriented databases must be accessible to existing Standard Query Language (SQL) applications. Some applications will require access to both relational and object-oriented databases.

[Loo93a] describes the process of accessing a relational database from an object program. Defining a mapping of the relational schema into an object model is the first task. A simple approach is to represent each relation by a class and replace foreign key fields (in the relational schema) by relationships (in the object-oriented schema). Given a mapping of tables to classes, a means of invoking SQL operations from the object program must be defined. This can be provided by defining methods on the mapped classes for creating, updating, and deleting instances. These methods are responsible for interacting with the relational database. Additional methods are required to provide an interface to query operations that translate the tuples returned from a query into a set of objects accessible by the object program. Database interface generator products, providing automated support for interfacing object programs to relational databases, are currently being developed. These products may work from a user-developed set of class definitions, or from the relational database's data definitions language (DDL). In either case, the result is a set of method specifications and implementations that provide access to a relational database from an object program.

OODBMS vendors are moving to support the need for SQL access to their databases. This need arises due to the large experience base of SQL users and the desire for existing applications to be continually supported, as OODBMS systems become part of the information infrastructure. The basic approach to this task is to incorporate an SQL interface in the OODBMS application programming interface (API) and to provide an SQL query processing capability (see Section 3.1.6, Querying an OODBMS).

3.1.2.9 Active vs. Passive Object Management System

OODBMS may be characterized as being an active or a passive object management system. A passive OODBMS means that the database does not store the implementation of the methods defined for a class. Applications built on a passive OODBMS provide in their executable image the code for each method defined in the system. Application execution results in each object that is accessed during that execution, being moved from the database server process to the client application process. Once the object resides in the application's address space, a message may be sent to that object resulting in the object executing one of its methods. Note that the process of moving the object from the database server to the application's address space is typically transparent to the application programmer.

An active OODBMS means that the database stores the implementation of object behaviors (i.e., methods) in the database. This allows objects to execute those behaviors (i.e., respond to messages) in the database server process. Advantages of an active data model include:

- Objects may be accessed and manipulated by non object-oriented programs. These programs may access objects in the database through a standard programming language interface. Each such access may result in a long series of messages being sent between many different objects that are cooperating to provide some useful service on behalf of the requesting application.

- o Object behaviors are stored in a single location (the database) which makes it easier to be sure that all applications have the latest version of those behaviors. This also tends to isolate those applications from changes to the object behaviors.
- o Each object accessed by the application need not be transferred to the application's address space.
- o Consistency checks (i.e., constraints) can be automatically maintained by the database. As an object's state is changed, the database server process can automatically execute consistency checks to ensure that the new state does not violate some constraint.

One of the significant differences between an active and a passive OODBMS becomes apparent when considering the implications of traversing 10,000 objects as part of a query or other database operation. Using a passive database requires that each of those 10,000 objects be moved from the database server to the client application prior to invoking the methods that access the objects. An active database can be programmed so that the traversal and method invocation occurs in the database server process, eliminating the need to transfer each object across the network to the application process.

3.1.3 Database Architecture

This section provides an overview of architectural issues relevant to an OODBMS. Readers interested in further details can see [Cat91] for a discussion of implementation issues and techniques. [Tha86] describes an implementation of a persistent memory system upon which object-oriented databases may be built. [Ska86] describes an implementation of a shared database server architecture suitable as a back-end for an object-oriented database.

3.1.3.1 Distributed Client - Server Approach

Advances in local area network and workstation technology have given rise to group design type applications driving the need for OODBMS (e.g., CASE, CAD, Electronic Offices). OODBMS typically execute in a multiple process distributed environment. Server processes provide back-end database services such as management of secondary storage and transaction control. Client processes handle application specific activities such as access and update of individual objects. These processes may be located on the same workstation or on different workstations. Typically a single server will be interacting with multiple clients servicing concurrent requests for data managed by that server. A client may interact with multiple servers to access data distributed throughout the network.

[DeW92] evaluates and benchmarks three alternative workstation-server architectures that have been proposed for use with OODBMS:

Object server approach - the unit of transfer from server to client is an object. Both machines cache objects and are capable of executing methods on objects. Object-level locking is easily performed. The major drawback of this approach is the overhead associated with the server interaction required to access every object and the added complexity of the server software which must provide complete OODBMS functionality (e.g., be able to execute methods). Keeping client and server caches consistent may introduce additional overheads.

Page server approach - the unit of transfer from server to client is a page (of objects). Page level transfers reduce the overhead of object access since server interaction is not always required. Architecture and implementation of the server is simplified since it need only perform the backend database services. A possible drawback of this approach is that methods can be evaluated only on the client, thus all objects accessed by an application must be transferred to the client. Object level locking will be difficult to implement.

File server approach - the OODBMS client processes interact with a network file service (e.g., Sun's NFS) to read and write database pages. A separate OODBMS server process is used for concurrency control and recovery. This approach further simplifies the server implementation since it need not manage secondary storage. The major drawback of this approach is that two network interactions are required for data access, one to the file service and one to the OODBMS server.

[DeW92] identified no clear winner when benchmarking the three approaches. The page server approach seemed best with large buffer pools and good clustering algorithms. The object server approach performed poorly if applications scanned lots of data, but was better then the page server approach for applications performing lots of updates and running on workstations with small buffer pools.

3.1.3.2 Data Access Mechanism

An evaluation of OODBMS products should consider the process necessary to move data from secondary storage into a client application. Typically this requires communication with a server process, possibly across a network. Objects loaded into a client's memory may require further processing, often referred to as swizzling, to resolve references to other objects which may or may not already be loaded into the client's cache. The overhead and process by which locks are released and updated objects are returned to the server should also be considered.

3.1.3.3 Object Clustering

OODBMS which transfer units larger than an object do so under the assumption that an application's access to a given object implies a high probability that other associated objects may also be accessed. By transferring groups of objects, additional server interaction may not be necessary to satisfy these additional object accesses. Object clustering is the ability for an application to provide information to the OODBMS so that objects which it will typically access together can be stored near each other and thus benefit from bulk data transfers.

3.1.3.4 Heterogeneous Operation

[Loo92e] describes the role of an OODBMS as an application integration technology. An OODBMS provides a mechanism for applications to cooperate, by sharing access to a common set of objects. A typical OODBMS will support multiple concurrent applications executing on multiple processors connected via a local area network. Often, the processors will be from different computer manufacturers; each having its own data representation formats. For applications to cooperate in such an environment, data must be translated to the representation format suitable for the processor upon which that data is stored (both permanently by a server and temporarily by a client wishing to access the data). To be an effective integration mechanism, an OODBMS must support data access in a heterogeneous processing environment.

3.1.4 Database Functionality

The primary benefit an application derives from a database is that application data persists across application executions. Additional benefits offered by a database are the ability to share data between applications, provision of concurrent access to the data by multiple applications, and providing an application the access to a data space larger than its process address space. This section reviews the issues that distinguish an OODBMS from a language with persistence (e.g., Smalltalk). A language with persistence typically provides for data to exist across executions but does not provide the additional benefits outlined above.

Within this section we provide a brief overview of the database topic with additional issues relevant to object-oriented databases. Additional information on the topics listed here is available from any good text on database management systems (e.g., [Dat86]).

3.1.4.1 Access to Unlimited Data

A database provides an application the ability to access a virtually unlimited amount of data. In particular, the application can address more data than would fit within the application process address space. Some databases may support the notion of transient data that is maintained during program execution but not saved in the database. This is useful for providing access to very large transient data objects that do not map easily into the application's address space.

3.1.4.2 Integrity

A database is required to maintain both structural integrity and logical integrity. Structural integrity ensures that the database contents are consistent with its schema. Logical integrity ensures that constraints specifying logical properties of the data are always true. [But92] describes new concerns for integrity in OODBMS. A major concern is that OODBMS architectures map data directly into the application's address space (unlike a typical relational database that provided direct access to the data only in a separate database server process). Mapping data into the application's address space yields significant performance improvements over server-only data access, especially for the application areas being targeted by OODBMS [Cat91]. However, once data is mapped into an application's address space there is no way to guarantee it is not inadvertently or maliciously tampered with. This limits a database's ability to guarantee integrity of the data.

Structural integrity mechanisms include assurances that references to deleted objects do not exist (see Section 3.1.2.1, Relationships & Referential Integrity) and that all instances are consistent with their class definitions. Logical integrity can be supported by encapsulating all the data members of a class and providing access to information content of an object only through behaviors defined by the class. Additional integrity constraints can be supported if the database provides a mechanism for specifying application-level constraints and for ensuring the execution of those constraints before and/or after behavior invocation (see Section 3.1.4.10, Constraints). Constraints executed before a behavior can test the consistency of the request and the input parameters. Constraints executed after a behavior can test for logical consistency of the resulting state of the object and any output parameters.

3.1.4.3 Concurrency

Databases provide concurrency control mechanisms to ensure that concurrent access to data does not yield inconsistencies in the database or in applications due to invalid assumptions made by seeing partially updated data. The problems of lost updates and uncommitted dependencies are well documented in the database literature. Relational databases solve this problem by providing a transaction mechanism that ensures atomicity and serializability. Atomicity ensures that within a given logical update to the database, either all physical updates are made or none are made. This ensures the database is always in a logically consistent state, with the DB being moved from one consistent state to the next via a transaction. Serializability ensures that running transactions concurrently yields the same result as if they had been run in some serial (i.e., sequential) order.

Relational databases typically provide a pessimistic concurrency control mechanism. The pessimistic model allows multiple processes to read data as long as none update it. Updates must be made in isolation, with no other processes reading or updating the data. This concurrency model is sufficient for applications that have short transactions, so that applications are not delayed for long periods due to access conflicts.

For applications being targeted by OODBMS (e.g., multi-person design applications), the assumption of short transactions is no longer valid. Optimistic concurrency control mechanisms [Kun81] are based on the assumptions that access conflicts will rarely occur. Under this scenario, all accesses are allowed to proceed and, at transaction commit time, conflicts are resolved. OODBMS have incorporated the idea of optimistic concurrency control mechanisms for building applications that will have long transaction times. Handling of conflicts at commit time cannot simply abort a transaction, however, since one designer may be losing days or weeks of work. OODBMS must provide techniques to allow multiple concurrent updates to the same data and support for merging these intermediate results at an appropriate time (under application control). Most systems use some form of versioning system in order to handle this situation (see Section 3.1.2.4, Object Versioning).

An alternative policy is to allow reading and a single update to occur in parallel. Readers are made aware that the data they are reading may be in the midst of an update. Thus readers may be viewing slightly outdated information. Implementation of this approach fits well in the client-server architecture typical of an OODBMS. Each client application gets its own local copy of the data. If an update is made to the data, the server does not permanently store it until all concurrent read transactions are completed. Thus, all read transactions execute seeing a consistent data set, albeit one that is in the process of being updated. Once all readers have completed, the write transaction is allowed to complete modifying the permanent copy of the data. Some OODBMS may, at transaction commit, inform reading clients that the data they just read is in the process of being updated.

3.1.4.4 Recovery

Recovery is the ability for a database to return to a consistent state after a software or hardware failure. Similar to concurrency, the transaction concept is used to implement recovery and to define the boundaries of recovery activity. One or more forms of database journaling, backup, checkpointing, logging, shadowing, and/or replication are used to identify what needs to be recovered and how to perform a recovery.

Databases must typically respond to application failures, system failures, and media failures. Application failures are typically trapped by the transaction mechanism and recovery is implemented by rolling back the transaction. System failures, such as loss of power, may require log and/or checkpoint supported rollback of uncommitted transactions and roll forward of transactions that were committed but not completely flushed to disk. Media failures, such as a disk head crash, require restoration of the database from a backup version, and replaying of transactions that have been committed since the backup.

The ability of a database to recover from failures results in a heavy processing and storage overhead. In the process of evaluating an OODBMS, its ability to recover from faults, and the overhead incurred to provide that recovery capability, must be carefully considered Applications envisioned for OODBMS (e.g., CASE tools) often do not have the same strict recovery requirements as do relational database applications (e.g., banking systems). In addition, the amount of data stored in such systems may result in unacceptable storage overheads for many forms of recovery. For these reasons, an OODBMS evaluation effort must carefully select the recovery capabilities needed based on both the functional and performance requirements of the application.

3.1.4.5 Transactions

Transactions are the mechanism used to implement concurrency and recovery. Different transaction policies have been described in Section 3.1.4.3, Concurrency, under the topics of pessimistic, optimistic, and multiple readers/single write concurrency control policies. Within a transaction, data from anywhere

in the (distributed) database must be accessible. A feature found in many OODBMS products is to commit a transaction but to allow the objects to remain in the client cache under the expectation that they will soon be referenced again.

Some OODBMS have incorporated the concept of long and/or nested transactions. A long transaction allows transactions to last for hours or days without the possibility of system generated aborts (due to lock conflicts for example). System generated aborts must be avoided for applications targeting OODBMS since a few hours or days of work cannot be simply discarded. Long transactions may be composed of nested transactions for purposes of recovery [Cat91].

Nested transactions [Mos85] allow a single (root) transaction to be decomposed into multiple subtransactions. Each subtransaction can be committed or aborted independently of the other subtransactions contained in the scope of the root transaction. Each subtransactions commit is dependent upon its immediate superior committing and the root transaction committing. Nested transactions improve upon the basic transaction mechanism by providing finer-grained control over the failure properties of software. Using nested transactions, a portion of a computation can fail and be retried without affecting other parts of that same computation. Nested transactions were developed to provide concurrency controls for distributed computing systems. Again, an OODBMS evaluation must carefully consider whether the target application requires nested transactions and the performance and/or storage impacts of using this facility.

3.1.4.6 Deadlock Detection

Database systems use deadlock detection algorithms to determine when applications are deadlocked due to conflicting lock requests. Relational DBMS typically select an arbitrary transaction and abort it in an attempt to let the remaining transactions complete. The aborted transaction is normally restarted automatically by the system. This scheme works well for the class of applications supported by relational DBMS. As described in the previous section, applications being targeted by OODBMS cannot afford a system-aborted transaction resulting in the potential loss of hours, days, or weeks of work. OODBMS provide alternative concurrency control and transaction mechanisms to reduce or avoid the possibility of deadlocks. Regardless of concurrency control protocol, an OODBMS must still detect and resolve deadlocks.

3.1.4.7 Locking

Locking of database entities is the typical approach to implementing transactions. OODBMS may provide locking at the object and/or the page level. Object-level locking may result in high overheads due to lock management. Page-level locking may reduce concurrency, especially for write locks, since not all objects on a page may be used in the transaction that holds the lock. OODBMS clustering facilities will aid in reducing the loss of concurrency due to page-level locking.

An OODBMS will implicitly acquire and release locks as data is accessed by an application. Application support may be necessary for specifying the lock mode (e.g., all locks may be acquired in read mode by default and the application must specify that write access to the object is necessary). The OODBMS may also provide an interface for explicitly locking data.

3.1.4.8 Backup and Restore

Backup is the process of copying the state of the database to some other storage medium in case of subsequent failure or simply for historical record. Ideally backups can be performed while the database is in use. Backups may be performed only on specific sections of the database. Incremental backup

capabilities reduce the amount of information that must be saved by storing only changes since a prior backup. Obviously, a database must be able to be restored from a backup.

3.1.4.9 Dump and Load

A database may be dumped into a human readable ASCII format. Specific segments of the database may be dumped. A database may be recreated by loading from a dump file.

3.1.4.10 Constraints

Constraints are application-specific conditions that must always hold true for a database. Logical database integrity can be supported by providing the application developer with the ability to define constraints and for those constraints to be automatically executed at the appropriate times. Although constraints can be directly encoded in behaviors, having a separate constraint mechanism reduces duplication and ensures execution of the constraints. Some OODBMS define a model for constraint definition and invocation but require applications to explicitly invoke the constraint executions.

Constraints executed at behavior invocation can test for validity of the input parameters and the requested operation. Constraints executed at the completion of a behavior invocation can test for logical consistency of the result values and resulting database state.

3.1.4.11 Notification Model

An OODBMS may provide a passive and/or an active notification model. A notification model allows an application to be informed when an object is modified or when some other database event occurs. Passive notification systems require that the application query the database for state changes. A passive system minimally provides the logic that determines if an object has changed state or if an event has occurred. The application is responsible for informing the database of the objects and events in which it is interested and for periodically querying the database to see if those objects have been updated or if particular events have occurred.

An active notification system is one in which application routines are invoked automatically as a result of some object being updated or some database event occurring. Active notification systems are similar to database triggers and constraint systems (whereby constraints are executed when one of their operands changes state). Much work has been done on database triggers and constraint systems [Par89] [Zdo86] [Hud86] [Bun79] [Bor81] [Buc86]. Presently there is no clear indication that such mechanisms can be efficiently supported in general purpose database management systems.

3.1.4.12 Indexing

Databases make use of indexing to provide optimized data retrieval based on some aspect of that data. In particular, query evaluations can be dramatically improved by the presence of indexes. Query optimizers must dynamically determine if an index is present and, if so, use that index to provide an efficient query execution. [Mai86b] and [Kim92] discuss indexing of OODBMS systems.

Indexes in an OODBMS are typically built to provide lookup of all objects of a given class and its subclasses based on one or more data members of that class. Indexes may be segregated to some segment of a database or may span the entire database.

Different indexing implementations result in different time and space performance. Hashing and b-trees are two common index implementation techniques. Indexing adds to the overhead associated with creating, deleting, and modifying objects and thus must be used judiciously.

3.1.4.13 Storage Reclamation

Data stored within a database is dynamically created and destroyed by the applications that access that database. Data that is no longer accessible, whether determined implicitly by the system or as the result of an explicit delete by an application, results in storage space that is no longer in use. Reclamation of unused space must be performed so that a database does not continuously grow. Space reclamation should be performed incrementally or as a background activity so that performance hiccups are not encountered by the applications.

3.1.4.14 Security

Secure database systems protect their data from malicious misuse. Security requirements are similar to data integrity requirements that protect data from accidental misuse. Secure databases typically provide a multi-level security model where users and data are categorized with a specific security level. Mandatory security controls ensure that users can access data only at their level and below. Discretionary security controls provide access control based on explicit authorization of a user's access to data. Applications targeted by OODBMS often do not require strict security controls, although discretionary access controls seem desirable for work-group design type applications. [Thu92a] and [Thu92b] discuss security concerns relevant to OODBMS. Little work has been done to add security mechanisms to OODBMS.

3.1.5 Application Programming Interface

A major distinction between an object-oriented database application and a relational database application is the relation between the data manipulation language (DML) and the application programming language. Large scale commercial database applications are developed in a standard programming language (e.g., C, C++). In a relational database, SQL is used as the DML, providing the means to create, update, query, and delete data from the database. Thus, in a relational database application, a significant amount of effort is spent transforming data between the two different languages. Critics claim that this impedance mismatch adds to the complexity of building relational database applications. In an object-oriented database application, the DML is typically very near a standard programming language. For example, many OODBMS support C++ as their DML, with the result that C++ can be used for building the entire application. OODBMS proponents claim that by using a single language, the impedance mismatch is removed (or significantly reduced) and application development is simplified.

The OODBMS approach is not without its critics. [Sto90] claims that OODBMS application programming is a step backward (from relational) since the code typically is built by navigating relationships in the schema. OODBMS applications traverse the network of inter-connected objects to discover information. This hand-coded navigation of the schema cannot automatically take advantage of indexes that might be added and requires modification of the application code as the schema changes. SQL, on the other hand, is an associative retrieval language, not requiring any information describing how information is to be found. SQL statements can be recompiled and/or optimized at runtime to find the best access path to the requested data.

It is clear that both relational and object-oriented database approaches have merit and will be used for developing particular classes of applications. It is interesting to note that relational vendors are currently adding object-oriented features to their products and that OODBMS vendors are adding SQL-like query capabilities to theirs. In addition, third party companies are developing automated integration tools that allow SQL applications to access OODBMS and OODBMS to access relational databases.

This section covers issues relevant to the application programming interface of an OODBMS. As described above, OODBMS aim to provide a tight integration between the data definition/manipulation language and a standard programming language in an attempt to ease the application development task.

3.1.5.1 DDL/DML Language

An OODBMS provides one or more languages in which data definitions can be specified and applications can be constructed. A common example is to use C++ header files for describing the class structures (i.e., schema) and then to implement the behaviors of those classes and the remainder of the application in C++. The data description component of the language often is an extension of some standard programming language to support specification of relationships and other object-oriented features.

3.1.5.2 Computational Completeness

Relational database query languages, such as SQL, are typically not computationally complete, meaning general purpose control and computation structures are not provided. For this reason, applications are built by embedding the query language statements in a standard programming language. OODBMS that use a standard programming language (or an extension of one) for data definition and data manipulation provide the application developer with a computationally complete language in which database manipulations and general purpose processing can be accomplished.

3.1.5.3 Language Integration Style

A number of mechanisms may be used for providing access to the database from the programming language. These include library interfaces, language extensions, or for true object-oriented languages, definition of behaviors for construction, destruction, and access via member functions. Most authors define a loose language integration as one where the database operations are explicitly programmed for example by library calls. This is common for OODBMS interfaces from the C programming language. A tight language integration makes the database operations transparent, typically through inherited behaviors in a class hierarchy. For example, a C++ integration might use the standard class constructor and destructor constructs to create and delete objects from the database. Even in a tight integration, additional parameters will often be added to control database activity (e.g., clustering) and some database operations will be provided by library calls.

3.1.5.4 Data Independence

Data independence is the ability for the schema of the database to be modified without impacting the external (i.e., application) view of that schema. As described in Section 3.1.1.6, Encapsulation, this is accomplished by the schema definition language encapsulating the internals of data members and behaviors. Applications that rely only on the public interfaces of classes will be protected from changes to the private portions and the implementations of those classes.

The concept of derived attributes also adds to data independence. A derived attribute is one that is not stored, but instead calculated on demand. An application cannot determine whether an attribute is derived or not (i.e., if it is stored or computed), thus changes to the schema do not affect the application. Typically, programming languages do not easily support the concept of derived attributes. In C++ for example, a function call is syntactically different than an attribute reference, so providing transparent derived attributes is not possible. Eiffel [Mey88] is an object-oriented programming language that directly supports the concept of a derived attribute.

3.1.5.5 Standards

Applications can be portable across OODBMS products if all such products agree on a standard application programming interface (API). The Object Database Management Group (ODMG) is a working group of OODBMS vendors tasked with defining a set of interface specifications aimed at ensuring application portability and interoperability. All member OODBMS vendors have agreed to support the standard specification once it is developed (initial version expected in the fall of 1993). One of these specifications will be a C++ binding for object definition, manipulation, and query. Currently no standard exists and all OODBMS have a custom API, thus application programs are not portable across databases. A concern is that any existing applications will have to be modified when a standard is defined and implemented by the OODBMS vendors.

OODBMS typically integrate with a standard programming language such as C, C++, or Smalltalk. One issue is whether the OODBMS works with a standard version of a programming language (e.g., ANSI C, AT&T compatible C++). A second is whether the OODBMS uses a custom-built compiler or can use any third party compiler (e.g., from a compiler vendor).

3.1.6 Querying an OODBMS

Query languages provide access to data within a database without having to specify how that data is to be found. Thus, query languages provide a level of data independence. An application or user need not understand the structures storing the data in order to access the data. This is in strong contrast to network and hierarchical databases that require programmed navigation of the database. [Sto90] describes capabilities of the next generation database management system (beyond current relational systems). Included in these requirements are a flexible-type system, inheritance, association of behaviors with data, and the use of rules or constraints. As previously mentioned, [Sto90] claims that a navigational-based data access mechanism, as provided by most OODBMS, is a step backward from the associative access mechanisms of the relational DBMS generation. In particular, hand-coded navigational access has been shown to be less optimal than an optimized query approach, and changes to schema and indexing require changes to the navigational segments of application code. In light of this, it is clearly important for OODBMS to provide a declarative query language that is closely integrated with an object-oriented programming language.

3.1.6.1 Associative Query Capability

Standard Query Language (SQL), as defined by the ANSI X3H2 committee is being revised to incorporate object-oriented features. In the short term, OODBMS vendors are providing their own object-oriented extensions to SQL in order to provide reasonable access to their object databases from an SQL-like language.

[Cat91] identifies a range of query capabilities for OODBMS:

- No query language support - all data access must be via programmed database navigation.
- Collection-based queries - queries operate on some predefined collection of objects, selecting individual objects based on some predicate, yielding a resulting collection of objects.
- General queries - which can have a result of any type (e.g., value, object, or collection).

An additional capability is for a query to return textual information that is suitable for report generation.

3.1.6.2 Data Independence

As described above, the basic motivation for using a declarative query language is to support data independence. Query implementations are expected to make optimal use of schema associations and indexes at runtime.

3.1.6.3 Impedance Mismatch

One of the main criticisms of relational database programming is the impedance mismatch between the data manipulation language (DML), normally SQL, and the application programming language, typically some general purpose language such as C. Relational database applications have an impedance mismatch, in that database access via the query language is table-based while application programming is individual value-based. Extra code and intellectual hurdles are required to translate between the two.

A presumed benefit of OODBMS is that the application programming language and the DML are one and the same. However, as noted above, critics claim this eliminates data independence. Declarative query capabilities, which are being added to OODBMS, will support the concept of data independence. How well these query capabilities are integrated with the application programming language will dictate the level of impedance mismatch between the application programming language and the use of associative queries.

[Loo93b] identifies a range of techniques for integrating queries within an OODBMS:

- supply a Select method on all persistent classes,
- extend the object programming language to include SQL like predicates for filtering selection operations, and
- embed the SQL Select statement into the object programming language, providing a pre-processor which translates these statements into an appropriate set of runtime calls.

A tight integration of query invocation and query result with the selected OODBMS application language will decrease the impedance mismatch typical of database applications.

3.1.6.4 Query Invocation

As described throughout this section, the major emphasis is to provide an associative query capability from within programmed applications. An additional requirement is to provide a means for ad hoc query invocation, possibly from within a database browser tool.

3.1.6.5 Invocation of Programmed Behaviors

The query language should be able to invoke object behaviors as part of their predicates. Whether this can be done from programmed queries and/or ad hoc queries must be investigated.

3.2 Application Development Issues

As application developers, the authors of this report are interested in application development issues. Many of the topics covered in Section 3.1, Functionality, directly affect application development. For example language integration affects how developers perceive the use of the database. Another functional issue that is extremely important to the application development process is schema migration. The ability to migrate schema (i.e., to update objects in response to schema changes) affects the testing process and the ability to upgrade fielded versions of the software.

This section identifies more specific application development issues. Evaluation of application development issues is not as straightforward as that for functional issues. Functional evaluations can be

based on technical documentation and informally verified by reviewing interface specifications of the product. Application development issues are more abstract. Review of technical documentation will provide only a small glimpse of the application development process. Only through use of the product, on a large application, with a team of developers, will a true understanding of the application development process be derived.

Table 3.2, Application Development Evaluation Criteria, lists the following sections that define application development issues to be considered in performing a review of OODBMS products.

Table 3.2. Application Development Evaluation Criteria

Criteria Categories	Criteria
Developer's View of Persistence	
Application Development Process	
Application Development Tools	o Database Administration Tools o Database Design Tools o Source Processing Tools o Database Browsing Tools o Debugging Support o Performance Tuning Tools
Class Library	

3.2.1 Developer's View of Persistence

An evaluation should consider how a software developer perceives persistent objects and what coding constructs are used to access persistent objects. This issue is closely related to the language integration issue. Of particular interest is the need for explicit user code to access persistent objects, lock persistent objects, and to notify the database that a persistent object has been updated and must be stored back to the database.

3.2.2 Application Development Process

The application development process is the series of steps needed to define and structure databases, define database schema, process class behaviors, and to link, execute, and debug applications. An evaluation should consider the need to use vendor-supplied preprocessors and/or interactive schema development tools, should describe the integration with debuggers and program builders (e.g., Unix make), and should consider the issues relevant to multiple developer efforts. Refer to Section 3.1.2.6, Schema Evolution, to consider the issues relevant to maintenance of fielded applications.

3.2.3 Application Development Tools

Both vendor and third party tools must be supplied in the areas described in the following sections.

3.2.3.1 Database Administration Tools

Database administration tools are used for creating, deleting, and re-organizing (e.g., moving) databases. Database administration tasks should be available to applications (i.e., through a programmed interface) in order to allow applications to hide database administration tasks from the user.

3.2.3.2 Database Design Tools

Database design tools are used for interactively defining the schema or classes which are to be stored within a database. Third party object-oriented modeling tools may be available which generate source code suitable for use with the OODBMS.

3.2.3.3 Source Processing Tools

Source processing tools perform the transformation of textual descriptions of applications programs into executable code. These include pre-processors specific to the OODBMS as well as standard language compilers. Also included might be tools to aid in controlling the process of application building (e.g., Unix make facilities). Integration with application development environments should also be considered.

3.2.3.4 Database Browsing Tools

Interactive browsing tools allow database schemata and contents to be viewed graphically and possibly modified.

3.2.3.5 Debugging Support

An OODBMS vendor should supply tools and or utilities that are useful during the debugging process. These facilities should be invocable from third party debugging environments.

3.2.3.6 Performance Tuning Tools

An OODBMS should provide utilities which enable a developer to understand the performance parameters of an application and a means by which performance can be adjusted as a result of this analysis. In addition, any specific design considerations that can affect performance must be considered.

3.2.4 Class Library

Object-oriented development is based on building a reusable set of classes organized into a class hierarchy. The class hierarchy mechanism supports reuse of general data and behaviors in specialized classes. As described in Section 3.1.5.3, Language Integration Style, some OODBMS may provide their application interface through a class library. Inherited behaviors provide support for persistence, object creation, deletion, update, and reference traversal.

In addition to the possibility of providing database interface through the class library, an OODBMS may deliver application support classes. Such classes typically provide data abstractions such as sets, lists, dictionaries, etc. These classes should be extensible just like any other user-defined class. Ideally source code would be provided for these classes. Source is needed since documentation is often insufficient for determining the effect of invoking each method under each possible condition. Source is also useful in understanding performance characteristics and in repairing errors that may be found in the code.

3.3 Miscellaneous Criteria

A number of non-technical criteria should also be considered when evaluating an OODBMS. This section details some of these criteria, as listed in Table 3.3, Miscellaneous Evaluation Criteria.

Table 3.3 Miscellaneous Evaluation Criteria

Criteria
Product Maturity
Product Documentation
Vendor Maturity
Vendor Training
Vendor Support & Consultation
Vendor Participation in Standards Activities

3.3.1 Product Maturity

Product maturity may be measured by several criteria including:

- years under development,
- number of seats licensed,
- number of licensed seats actually in use,
- number of licensed seats in use for purposes other than evaluations (i.e., actual development efforts),
- number and type of applications being built with the OODBMS product, and
- number and type of shipped applications built with the OODBMS product.

3.3.2 Product Documentation

Product documentation should be clear, consistent, and complete. The documentation should include complete examples of typical programmed capabilities (e.g., what is the sequence of calls to access data from the database and to cause updates to that data to be made permanent in the database).

3.3.3 Vendor Maturity

Vendor maturity may be measured by several criteria including:

- company's size and age,
- previous experience of the company's lead technical and management personnel in the commercial database market, and
- financial stability.

3.3.4 Vendor Training

Availability and quality of vendor supplied training classes is an important consideration when selecting an OODBMS.

3.3.5 Vendor Support & Consultation

It is expected that significant support will be required during the OODBMS evaluation process and to overcome the initial learning curve. OODBMS vendors should provide a willing and capable support staff. Support should be available via phone and electronically. Consulting support might also be

appealing where the OODBMS vendor provides expert, hands-on assistance in product use, object-oriented application design (especially in regards to database issues), and in maximizing database application performance.

3.3.6 Vendor Participation in Standards Activities

The vendor should be active in standards efforts in the object-oriented, language, CASE, open software, and data exchange areas. In particular:

- Object Management Group (OMG) - an organization funded by over 80 international information systems corporations whose charter is to develop standards for inter-operation and portability of software. The OMG is focusing on object-oriented integration technologies such as Object Request Broker (ORB), OODBMS interfaces, and object interfaces for existing applications.
- Object Database Management Group (ODMG) - an organization of OODBMS vendors chartered to define a standard interface to OODBMS that will allow application portability and interoperability. Standards defined by the ODMG will be provided to OMG, ANSI, STEP, PCTE, etc. to aid in their respective standardization efforts.
- ANSI standardization efforts in languages (C, C++, Smalltalk), SQL, and object-oriented databases.
- Standards such as Portable Common Tool Environment (PCTE) and CASE Data Interchange Format (CDIF) providing for common data representations, data exchange formats and interoperation of tools.
- PDES/STEP - an effort aimed at standardizing an exchange format for product model data (product model data, such as CAD data, represents a prime application area for OODBMS).

4. Evaluation Targets

This chapter identifies the commercial OODBMS that were evaluated as part of this effort. For each evaluation target we identify:

- the platforms upon which that OODBMS is hosted,
- the level of heterogeneous operation supported by the OODBMS,
- the application interface languages provided by the OODBMS, and
- third party products that inter-operate in some way with the OODBMS.

Information provided in this section was provided directly by each vendor.

4.1 Objectivity/DB

Objectivity/DB is an object-oriented database product developed and marketed by Objectivity, Inc., 800 El Camino Real, Menlo Park, CA 94025, (415) 688-8024. Evaluation information provided in this report was obtained from the technical documentation set for Objectivity/DB Version 2.0 and from discussions with technical representatives of Objectivity, Inc.

Objectivity/DB may be executed by client applications hosted in a heterogeneous network of:

- o DECstation under Ultrix 4.2
- o Sun4/SPARC under SunOS 4.1, Solaris 2.0 or Solaris 2.1
- o VAX under Ultrix 4.2 or VMS
- o Hewlett Packard 9000 series 300 under HP/UX 8.0
- o Hewlett Packard 9000 series 700 or 800 under HP/UX 8.0 or HP/UX 9.0
- o IBM RISC System/6000 under AIX
- o Silicon Graphics Iris under IRIX 4.0
- o NCR System 3300 (i386) under SVR4 Version 2.0

Applications running on any of the above platforms, connected via a local area network, may share access to a single database. Objectivity, Inc., has announced and released a Beta test subset version of Objectivity/DB for Windows NT. A version running on DEC/ALPHA under OSF 1.0 is also in Beta testing..

Objectivity/DB provides application interfaces for:

- o AT&T compatible C++
- o ANSI C

Objectivity/DB was designed as an open product and is advertised to work with any ANSI C or AT&T compatible C++ compiler. Objectivity, Inc., has partnership agreements to develop tool integrations and/or be compatible with the following products:

- o Program Development Environments
  - SoftBench from Hewlett-Packard
  - ObjectCenter from CenterLine Software
  - FUSE from DEC
  - WorkBench 6000 from IBM
  - ObjectWorks from ParcPlace
- o RDBMS Gateways
  - Persistence Software
- o Object-oriented GUIs
  - UIM/X from Visual Software, Ltd.
  - XVT from XVT Software, Inc.
  - Integrated Computer Solutions, Inc.
  - Objective, Inc.
  - Micram Classify/DB
- o Analysis and Design Tools
  - PTech from Associative Design Technology
  - Paradigm Plus from ProtoSoft
  - ROSE from Rational
  - Softeam

An evaluation of Objectivity/DB is provided in Section 5.1 of this report.

4.2 ONTOS DB

ONTOS DB is an object-oriented database product developed and marketed by ONTOS, Inc., Three Burlington Woods, Burlington, MA 01803, (617) 272-7110. Evaluation information provided in this report was obtained from the technical documentation set for ONTOS DB 2.2 and from discussions with technical representatives of ONTOS, Inc.

ONTOS DB may be hosted on the following platforms:

- o IBM RISC System/6000 under AIX
- o IBM PC under OS2
- o Hewlett Packard 9000 series under HP/UX
- o SCO 386 Unix
- o Sun4/SPARC under SunOS 4.1

ONTOS DB does not support heterogeneous operation between any other target platforms.

ONTOS DB provides a C++ application interface. ONTOS DB can be used with AT&T compatible C++ compilers. ONTOS DB developers can debug using gdb or dbx Unix debugging environments.

ONTOS DB is compatible with the following products:

- o Program Development Environments
  - ObjectCenter from CenterLine Software
  - ObjectWorks from ParcPlace
- o Analysis and Design Tools
  - PTech from Associative Design Technology

An evaluation of ONTOS DB is provided in Section 5.2 of this report.

4.3 VERSANT

VERSANT is an object-oriented database product developed and marketed by Versant Object Technology Corp., 4500 Bohannon Drive, Menlo Park, CA 94025, (415) 329-7500. Evaluation information provided in this report was obtained from the technical documentation for VERSANT Release 2 and from discussions with technical representatives of Versant Object Technology Corporation.

VERSANT may be hosted on the following platforms:

- o Sun4/SPARC under SunOS 4.0
- o IBM RISC System/6000 under AIX
- o Hewlett Packard 9000 series under HP/UX
- o DECstation 3100 under Ultrix
- o Sequent under DYNIX/ptx
- o Silicon Graphics under IRIS
- o NeXT under NeXTstep
- o IBM PC under OS2

VERSANT supports heterogeneous operation between their Sun4, Hewlett Packard, and IBM RISC System/6000 platforms. Versant is adding additional platforms to their heterogeneous operation as an on-going activity. For example, the addition of OS2 platforms are expected before the end of this year.

VERSANT provides application interfaces for:

- o C++
- o ANSI C
- o Smalltalk

C++ compilers from AT&T, Sun, Hewlett Packard, Glockenspiel are compatible with VERSANT.

Versant Object Technologies has partnership agreements to develop tool integrations and/or be compatible with the following products:

- o Program Development Environments
  - ObjectCenter from CenterLine Software
  - ObjectWorks from ParcPlace
  - SoftBench from Hewlett-Packard
  - WorkBench 6000 from IBM
- o RDBMS Gateways
  - Persistence Software
- o Analysis and Design Tools
  - Paradigm Plus from ProtoSoft
  - ROSE from Rational
  - ACIS Geometric Modeler from Spatial Technology

An evaluation of VERSANT is provided in Section 5.3 of this report.

4.4 ObjectStore

ObjectStore is an object-oriented database product developed and marketed by Object Design, Inc., One New England Executive Park, Burlington, MA 01803, (617) 270-9797. Evaluation information provided in this report was obtained from the technical documentation set for ObjectStore Release 2.0 and from discussions with technical representatives of Object Design, Inc.

ObjectStore may be hosted on the following platforms:

- o Sun under Solaris 1.x and Solaris 2.x
- o Hewlett Packard under HP/UX
- o DEC under Ultrix
- o NCR under SVR 4
- o Univel under SVR 4
- o Olivetti under SVR 4
- o IBM RISC System/6000 under AIX
- o Silicon Graphics
- o IBM PC under Windows 3.1 and OS2

ObjectStore supports heterogeneous operation between their Sun, Hewlett-Packard, IBM RISC System/6000, and Silicon Graphics implementations. The next release of ObjectStore will support heterogeneous operation across all their implementations.

ObjectStore provides application interfaces for:

- o AT&T compatible C++
- o ANSI C

ObjectStore is advertised to work with any ANSI C or AT&T compatible C++ compiler. In addition, Object Design markets a compiler and other development tools for building ObjectStore applications. ObjectStore has completed partnership agreements to develop tool integrations and/or be compatible with the following products:

- o Program Development Environments
  - Borland C++ & Application Frameworks
  - CodeCenter and ObjectCenter from CenterLine Software
  - Energize Programming System, Lucid C++, and Lucid C from Lucid, Inc.
  - SynchroWorks from Oberon Software, Inc.
  - OpenBase from Prism Technologies, Ltd.
  - SPARCworks C++ Professional and ProWorks C++ from SunPro Marketing
- o Object-oriented GUIs
  - ViewCenter from CenterLine Software
  - zApp from Inmark Development Corp.
  - Devguide from SunSoft, Inc.
  - UIM/X from Visual Software, Ltd.
- o Analysis and Design Tools
  - Object Engineering Workbench for C++ from Innovative Software GmbH
  - HOOPS Graphics Development System from Ithaca Software
  - Paradigm Plus from ProtoSoft, Inc.

An evaluation of ObjectStore is provided in Section 5.4 of this report.

4.5 GemStone

GemStone is an object-oriented database product developed and marketed by Servio Corporation, 2085 Hamilton Ave., Suite 200, San Jose, CA 94125, (408) 879-6200. Evaluation information provided in this report was obtained from the technical documentation set for GemStone Version 3.2, from GeODE Version 2.0 (GeODE is a development environment for GemStone applications) and from discussions with representatives of Servio Corp.

GemStone may be hosted on the following platforms:

- o Sun4/SPARC under SunOS 4.1
- o IBM RISC System/6000 under AIX
- o DEC Station under Ultrix
- o Hewlett Packard 9000 under HP/UX
- o Sequent under DYNIX/ptx

GeODE is available on all of the platforms listed above. Servio provides Macintosh, PC/Windows 3.1, and)S/2 V2.0 Smalltalk application access to GemStone databases on the previously listed platforms

Applications running on any of the above platforms, connected via a local area network, may share access to a single database.

GemStone provides application interfaces for:

- GeODE
- C++
- C
- Smalltalk-80, Smalltalk/V
- Smalltalk DB, a multi-user extended dialect of Smalltalk, developed by Servio for building and executing GemStone applications.

Programs written in any of these languages can access GemStone objects simultaneously. C++ compilers from Sun, HP, Centerline, Sequent, and IBM are compatible with the C++ interface for GemStone. Smalltalk compilers from ParcPlace and Digitalk are compatible with the Smalltalk interface for GemStone.

Servio provides GeODE (GemStone Object Development Environment) for developing GemStone applications. This environment is a complete application development framework, providing both visual and textual construction of object-oriented database programs. GeODE supports construction of Motif based X Windows applications. GeODE includes tools for schema design, user interface construction, data browsing, reuse, visual software construction, and software debugging.

Servio provides the GemStone Databridge, a product providing access to SYBASE relational databases from GemStone applications.

An evaluation of GemStone is provided in Section 5.5 of this report.

4.6 ObjectStore PSE Pro

ObjectStore PSE Pro is a Java-based object-oriented database product developed and marketed by Object Design, Inc, 25 Mall Road Burlington, MA 01803, (781) 674-5000. Evaluation information provided in this report was obtained from the technical documentation set for ObjectStore PSE Pro Version 2.0.

ObjectStore PSE Pro may be hosted on the following platforms that support Java VMs:

- Windows 95
- Windows NT
- OS/2
- Macintosh
- Unix

ObjectStore PSE Pro supports JDK 1.1 or higher.

An evaluation of ObjectStore PSE Pro is provided in Section 5.6 of this document.

4.7 IBM San Francisco

IBM San Francisco is an object-oriented framework developed and marketed by IBM. Evaluation information provided in this report was obtained from the technical documentation set for IBM San Francisco Version 1.2, along with discussions with technical representatives of IBM, San Francisco division.

The IBM San Francisco framework may be used to develop client and server Java-based applications hosted in a heterogeneous network of:

- Windows 95 (client)
- Windows NT Workstation 4.0
- Windows NT Server 4.0
- AIX 3.1
- OS/400

Server applications can interface to ODBC-JDBC supported databases, or directly through object-relational mapping to:

- Microsoft SQL Server 6.5
- IBM DB2

An evaluation of IBM San Francisco is provided in section 5.77 of this report.

5. Evaluation Reports

This chapter provides a detailed evaluation of each OODBMS listed in Chapter 4, Evaluation Targets. Each OODBMS is evaluated in the areas of application development and on some of the more advanced object-oriented database issues. The specific evaluation topics are:

- Application Development Issues - which includes subsections:
    - Developer's View of Persistence
    - Application Development Process
    - Application Development Tools
        - Database Administration Tools
        - Database Design Tools
        - Source Processing Tools
        - Database Browsing Tools
        - Debugging Support
        - Performance Tuning Tools
    - Class Library
- Advanced Topics - which includes subsections:
    - Transaction and Concurrency Model
    - Relationships & Referential Integrity
    - Composite Objects
    - Location Transparency

- Object Versioning
- Work Group Support
- Schema Evolution
- Runtime Schema Access/Definition/Modification

The evaluations were performed by analyzing the technical documentation of the OODBMS products and through discussions with the vendors' technical representatives.

5.1 Evaluation of Objectivity/DB 2.0

Information presented in this section was derived from the technical documentation set for Objectivity/DB Version 2.0 and discussion with technical representatives of Objectivity, Inc. All evaluations are based on the C++ application interface to Objectivity/DB.

This section provides a detailed evaluation of application development issues and selected advanced topics.

5.1.1 Application Development Issues

5.1.1.1 Developer's View of Persistence

Objectivity/DB provides persistence on the basis of class. A number of system-defined classes provide behaviors that implement persistent storage of objects. Application classes whose objects are to be persistent must inherit from one of the supplied persistent classes. All instances of a class that inherit from a persistent class are persistent.

Objectivity/DB provides the concept of a Handle to access persistent objects. Handles are non-persistent objects that are created for purposes of providing access to persistent objects. Handles provide a level of indirect access to persistent objects which helps protect the integrity of the database (i.e., makes it more difficult but not impossible to inadvertently overwrite database pages that have been mapped into the application's address space). As applications access persistent objects via a Handle, the Handle is either:

- o storing the object identifier for the object, and upon access retrieves that object into the client application's memory space, storing the virtual memory pointer in the Handle, or
- o is storing a virtual memory pointer to the object.

In either case, when an application accesses a persistent object via a Handle a virtual memory pointer to that object is returned. This Handle-based approach helps to protect database integrity by ensuring all object accesses are to the desired object. The additional overhead of this indirect access is expected to be offset by other database and application operations (e.g., disk accesses).

Handle classes are automatically generated by the Objectivity/DB DDL processor for each persistent class defined by the user. Each Handle class inherits behaviors for operating upon the Handle. Among other things, these behaviors allow persistent objects:

- o to be referenced via the Handle,
- o to be moved from the database to the client process, and
- o to be marked as having been modified (which requires it to be written back to the database).

By default, all objects are retrieved via Handle in a read mode. Applications must explicitly request that an object be opened in update mode, indicating that the application intends to modify the object. In addition to the expected concurrency control effects, opening an object for update will result in the object being written back to the database.

5.1.1.2 Application Development Process

Objectivity/DB applications are developed using seven basic steps:

1. Develop a database bootstrap file which provides a unique database identifier, specifies upon which host machine a database lock server will reside, identifies a file to be used to store the database schema information, and defines the page size for the database. This is a one-time task that configures an Objectivity/DB database for use by any number of applications. All applications that will access this database will reference this bootstrap file when processing their DDL files. Note that the first DDL file processed using a given bootstrap file will create and initialize the database file used to store the schema information. This file is initialized with all predefined Objectivity/DB class definitions.
2. Design the database schema using Objectivity/DB DDL language. This is very similar to C++ class definition syntax and defines the classes that make up an application. The database schema can be organized in multiple DDL files as per developer preference. Objectivity/DB schemas may be designed by third party graphical object-oriented modeling tools.
3. Process the DDL schema files with the Objectivity/DB DDL processor. This generates standard C++ header and source files. Header files are generated for each class defined in the schema as well as for automatically generated classes which parallel the application defined classes (e.g., Handle classes, reference classes).
4. Develop the application by providing implementations for class member functions, free functions, and a main routine. This task is similar to any other C++ development effort with the exception that a number of automatically generated classes, and Objectivity/DB provided classes, must also be used in implementing the application on top of the Objectivity/DB database.
5. Compile the DDL-generated source files (from Step #3) and the application source files (developed in Step #4).
6. Link the compiled source using Objectivity/DB object libraries.
7. Test and debug the application as would be done for any normal C++ application. Once completed, performance tuning can be performed as needed to meet application performance requirements.

It does not seem that Objectivity/DB adds to the traditional difficulties of large scale, multi-person, application developments. Configuration control of DDL schema files is basically the same as that of standard C++ header files (in fact the schema files are used to generate header files). Refer to Section 5.1.2.7, Schema Evolution, to consider the issues relevant to maintenance of fielded applications.

5.1.1.3 Application Development Tools

5.1.1.3.1 Database Administration Tools

Objectivity/DB provides an extensive set of database administration capabilities. Database administration is performed through a set of utility programs. All database administration tasks may be performed via command line invocation of the utility programs. A subset of the database administration capabilities are available to applications through C++ and C interfaces. This allows applications to include database administration as part of their functionality and thus reduce the need for expert, off-line, database administration services. Objectivity/DB provides a graphical tool manager (requires X windows) which allows application developers to browse databases and perform some administration tasks (e.g., lock monitoring, backup, statistics gathering).

Objectivity/DB provides a hierarchical storage model in which a single logical database (called a federated database by Objectivity, Inc.) is comprised of multiple distributed databases. Each distributed database, located anywhere in the local area network upon which the database is stored, is composed of one or more containers. Each container stores objects in one or more disk pages.

Objectivity/DB provides utilities to perform the following database administration tasks:

- Database configuration - includes creating and deleting federated databases and (distributed) databases as components of federated databases. Distributed databases may be moved to a new host and/or physical file location while remaining in the same federated database. Control parameters for the federated database (e.g., location of the lock server) may be reviewed and/or changed. Copies of federated databases and distributed databases may be made. All files comprising a federated database may be listed. Database control information for a specific database file may be listed.
- Locking Support - provides a lock monitor which displays the current status of locks against the database. This can be used, for example, to identify transactions that have never been completed (which may occur, for example, when a debugging session is quit in the middle of a transaction). Start and stop of the lock server process. Set and release exclusive locks on an entire federated database.
- Transaction Support - identify transactions that are waiting, and upon which transaction they are waiting. Rollback transactions that were not properly terminated (e.g., due to debugger exit or system crash).
- Perform backup and recovery of federated databases. Backups can be performed on-line and incrementally.
- Reclaim unused storage in the federated database. Note that Objectivity/DB reuses space from deleted objects as new objects are created. A utility program to force space reclamation is also provided to the database administrator and through the database administration programming interface.

5.1.1.3.2 Database Design Tools

Objectivity, Inc. does not provide an interactive, graphical database design tool as part of the Objectivity/DB product. Section 4.1 defined a number of object-oriented analysis and modeling tools which can be used as a front end to developing applications with Objectivity/DB. For example, Objectivity/DB has been integrated with ADT PTech from Associative Design Technology. ADT PTech is a graphical modeling tool that can generate complete C++ applications from a user's object-oriented model. ADT PTech can be configured to generate Objectivity/DB DDL definitions for a given object-oriented model and to produce applications which use Objectivity/DB for persistent storage of objects.

Objectivity/DB is integrated with a number of software development environments as described in Sections 4.1 and 5.1.1.3.3. These tools typically provide graphical browsing capabilities that aid in understanding existing parts of an application and are essential during applications development.

Objectivity/DB provides a graphical type browser (see Section 5.1.1.3.4, Database Browsing Tools) which can also be used during schema definition as a means of viewing the definition of existing classes.

5.1.1.3.3 Source Processing Tools

Section 4.1 described Objectivity/DB's compatibility with any standard C or C++ compiler and identified a number of program development environments that may be used with Objectivity/DB. For example,

Objectivity/DB has been tightly integrated with SoftBench and C++ SoftBench from Hewlett-Packard. SoftBench can be used to invoke Objectivity/DB browsing and source processing tools. Objectivity/DB tools can request operations such as source code display from SoftBench. In addition, SoftBench graphical editors can be used to view and modify Objectivity/DB DDL files. SoftBench Make facilities can be used to build applications on top of Objectivity/DB. As has been demonstrated by their integration with SoftBench, Objectivity/DB tools have been designed to work with common communication protocols that allows Objectivity/DB to be tightly integrated with standard CASE environments.

Objectivity/DB provides a C++ source preprocessor that translates DDL files into standard C++ header files and an associated set of source files for database classes such as handles, iterators, etc. This preprocessor must be invoked on all DDL files as described in Section 5.1.1.2, Application Development Process. This preprocessor is also responsible for loading an Objectivity/DB database with object definitions for the persistent classes that make up an application. Any changes to the schema definitions of a class require re-processing of the DDL file by the preprocessor.

5.1.1.3.4 Database Browsing Tools

Objectivity/DB provides three browsing facilities:

- o the Type Browser,
- o the Data Browser, and
- o the Query Browser.

The browsers must be run in an X Windows environment.

The Type Browser provides a graphical representation of the database schema. The Type Browser displays class hierarchy and data members but not method specifications. Multiple Type Browsers may be operating concurrently.

The Type Browser is opened on a specific database. The browser allows the user to select a type, from all types defined in the database, and as a result of that selection, displays types derived from it and data members of the type. Traversing through the type hierarchy or across relationships defined within a type is supported by a single mouse click.

The Data Browser provides a graphical representation of the contents of a database. The Data Browser shows the storage hierarchy of a database, which is composed of multiple distributed databases, containers, and objects. Selecting a distributed database object displays all containers for that database. Selecting a container object shows all objects stored within the container. Selecting any object (database, container, or a basic object) shows the contents of that object. Multiple Data Browsers may be operating concurrently.

Within the Data Browser, object contents are displayed by showing the schema representation of the class of the selected object and, for each schema entity, the value stored within that object. Buttons, depicting an object's name or its identifier, appear where relationships exist between the selected object and other objects. Pressing on such a button causes the selected object to become the focus of the Data Browser. The Data Browser provides buttons for moving within the storage hierarchy and for scrolling through array typed data members. The Data Browser does not allow objects to be created or object values to be modified.

The Query Browser is a subcomponent of the Data Browser and provides a mechanism for performing ad hoc queries. Objects, selected from a user specified set of objects, and meeting a user specified condition, can be identified via the query browser. Once identified, the objects may be viewed using the Data

Browser. Output from the Query Browser may be directed to text files that can be used as input to report generation facilities. The output is in an Objectivity/DB specific format and requires processing by external tools.

Queries can be applied against specific types of objects and may be applied to the entire federated database, a single distributed database, or a single container. Queries may also be applied to the set of objects related to a given object across some relationship. Query conditions are specified in a subset of SQL and/or C++ syntax. Query expressions may refer to data members of a type but may not invoke behaviors.

5.1.1.3.5 Debugging Support

Objectivity/DB generates C++ header and source files that are compatible with any AT&T compatible C++ compiler. Any debugger for AT&T compatible C++ may be used to debug applications built on top of Objectivity/DB databases.

Objectivity/DB provides an interactive debugging environment that can be run as a standalone tool or as part of the Unix debugger dbx (not available on VAX and IBM RS6000 hosts). The debug facility provides read and update modes. Objects can be referenced by OID or by name.

Commands available in both read and update mode include:

- List contents of database section (shows OID and class).
- Print the class structure of an object.
- Print the contents of an object (also shows class structure).
- Iterate an association, printing contents of each associated object.
- Change debug mode.

Commands available in update mode only are:

- Create and delete objects.
- Associate and disassociate objects.
- Assign values to data members of objects.
- Control transactions which commit or rollback updates to objects.

Objectivity/DB provides two non-interactive debugging support facilities: verification and tracing. These facilities are accessible via a 'debugging' version of the Objectivity/DB object libraries. Use of these facilities slows performance but they are not intended to be delivered in a final product. Once an application is linked with the debugging versions of the object libraries, debugging is controlled via environment variables. Thus, applications need not be recompiled or relinked to turn debugging on or off.

The verification facility is provided at three levels:

- Object Verification - checks that objects accessed in read only mode are not modified by the application. Object verification is implemented by keeping a 'before image' of all objects accessed in read mode. At transaction commit, these images are used to ensure that the object has not been accidentally modified. This form of verification is useful for finding errors where applications modify objects that have been accessed for read. These checks may also catch erroneous modification of data due to misuse of C++ pointers.

- o Page Verification - checks the consistency of Objectivity/DB page structures when read from the database and prior to being written to the database. Objectivity/DB uses pages as the basic unit of object storage and disk transfer. Each page maintains information about all objects stored within it. Checks are made that all space within the page is accounted for (either used or marked as free), that all objects have valid types, offsets, and sizes, and that the page header is correct.
- o Container Verification - Objectivity/DB uses containers as the mechanism for grouping pages. Container verification checks the internal consistency of a container when it is opened or closed. This level of verification will normally only detect errors in Objectivity/DB since containers are transparent to the application.

The second form of debugging support is Event Tracing. Event Tracing writes an account of object/page/container/database operations to a log file. The event trace can be used for identifying the sequence of events prior to the occurrence of an error.

5.1.1.3.6 Performance Tuning Tools

Objectivity/DB provides both explicit and implicit techniques for improving the performance of an application. Also provided is a performance monitoring facility that reports on the frequency of database events.

Objectivity/DB provides the ooRunStatus function which reports frequency of operations on databases, objects (and versions), associations, transactions, buffer usage, page usage, hash overflows, and disk access. This information, along with operating system supplied statistics such as I/O overheads, can be used as input to a performance tuning phase of application development.

Objectivity/DB provides explicit means to control the page size for a database. Page size affects the amount of information transferred from disk on each access and may be set only once for a database. Objectivity/DB provides, on a per application basis, control over the initial number of cache pages, the maximum number of cache pages, and the maximum number of files that may be concurrently open. An Objectivity/DB is stored in multiple files, thus the number of concurrently open files affects file access performance. The application cache is the amount of application address space allocated for storage of Objectivity/DB disk pages. A larger cache requires fewer swaps of database disk pages (at the cost of a larger application memory image).

Objectivity/DB provides a number of implicit mechanisms for improving the performance of an application. The most important factor regarding database performance is clustering of objects that are typically used together. Clustering is specified when an object is created and may be adjusted by ‘moving’ an object.

In general, all forms of inter-object reference (see Section 5.1.2.2, Relationships & Referential Integrity) can be optimized to save space at the cost of location transparency. Uni-directional associations may be used instead of bi-directional associations, although the application must then ensure referential integrity of those associations. Judicious use of indexing can also improve application performance.

The Objectivity/DB documentation set provides approximately thirty-five pages devoted to performance tuning. In particular, information is provided on techniques for optimizing an application with respect to time, space, and concurrency.

5.1.1.4 Class Library

Objectivity/DB provides a C++ class hierarchy supporting the construction of persistent applications. The Objectivity/DB class library includes classes:

- ooObj, which defines persistent behavior for objects.
- ooContObj, ooDBObj, and ooFDObj, which define a hierarchical storage model for storage of objects in a distributed database.
- A set of classes supporting construction of dictionaries, allowing objects to be associated with a name.
- String and array classes.
- A set of classes for manipulating Handles, non-persistent objects used for application interface to persistent objects.
- Support classes for indexes, iterators, and inter-object references.

Objectivity/DB also provides a number of free functions for controlling database parameters and performing other miscellaneous operations.

5.1.2 Advanced Topics

5.1.2.1 Transaction and Concurrency Model

Objectivity/DB's transaction model facilitates consistent database access by multiple concurrent applications. Pessimistic and multiple readers/single writer concurrency control policies are available. Applications using the multiple readers/single writer concurrency policy are provided with behaviors for determining if their view of the data is stale.

Objectivity/DB supplies a non-persistent class definition ooTrans for transaction control. Applications create a transaction object and then invoke methods on that object to start, commit, and abort a transaction. An application can only have a single transaction active at a time. Nested transactions are not supported.

Transactions represent the basic unit of database activity. All access to the database should be from within a transaction in order to assure consistency. Transactions use locks to guarantee database consistency. Objectivity/DB provides read and update modes for locks. As objects are accessed from within a transaction, locks are implicitly acquired. An application can also explicitly acquire locks. All implicitly acquired locks are read mode locks. An application is responsible for specifying that an object is being modified, and thus should be written back to the database upon transaction commit. Objects that are being modified by an application are locked in update mode. Locks acquired in a transaction are held until the end of that transaction. Objectivity/DB makes use of a single lock server process for coordinating the lock requests of all applications accessing a single database.

Objectivity/DB provides a hierarchical storage model for organization of a distributed database. Containers are the storage entity in which pages of objects are stored. Although locks are requested on an object basis, they are held at the container level. Containers represent a tradeoff of processing overhead for concurrency. By locking on containers instead of objects, an application can expect lower locking overhead and potentially better response time. Locking on containers reduces concurrency since all objects stored in a container are affected by locks on that container. An application can control the placement of objects in a container through object clustering directives. A well designed application must control the placement of objects in containers in order to benefit from clustered access and reduced locking overhead while not restricting concurrency.

Objectivity/DB implicitly acquires update locks for all object creations. Object accesses, whether for read or write, result in an implicit read lock. Objectivity/DB does not use a source code pre-processor that identifies write access to objects. For this reason, applications must identify all object updates. This is typically done by invoking ooUpdate, an inherited method in all persistent classes. Objectivity/DB provides a mechanism to propagate locks across a related set of objects (see Section 5.1.2.3, Composite Objects). Objectivity/DB system behaviors which modify object contents (e.g., creating relations between objects, see Section 5.1.2.2, Relationships & Referential Integrity) always acquire locks of the appropriate mode.

Lock conflicts (e.g., requesting a write lock when an object is already locked for read or write under a pessimistic concurrency control policy) can result in one of three behaviors:

1. The application requesting the conflicting lock can be immediately informed of the conflict via an error return.
2. The request can be retried some number of times after some waiting period.
3. The request can be queued for some amount of time (including infinite).

Each Objectivity/DB application can specify the desired lock conflict behavior, and the selected behavior may be changed at any time. Waiting periods and number of retries are application controlled. Objectivity/DB executes a deadlock detection algorithm for any lock request that is made when the lock conflict resolution is to queue the request for infinite time. Identification of a deadlock results in a runtime error message and denial of the lock request.

Objectivity/DB provides two transaction commit options, each of which uses the standard two-phase commit protocol [Gra76]. ooTrans method Commit ends the transaction and copies modified objects back to disk; translates handles (see Section 5.1.1.1, Developer's View of Persistence) from a memory pointer state to an object identifier state; updates indexes to reflect objects that have been created, deleted, moved, or modified; releases all locks acquired during the transaction; and marks application memory buffers (which store database objects in application memory) as inactive. Subsequent transactions may reuse the same handles and memory buffers. ooTrans method CommitAndHold ends the current transaction, copies modified objects back to disk, updates indexes, and starts a new transaction. CommitAndHold does not invalidate handles or application buffers and does not release locks. CommitAndHold provides the ability to checkpoint a set of changes to disk and continue processing the same objects with minimal overhead. ooTrans method Abort provides the standard transaction abort capability. Abort is similar to Commit with the exception that object updates are not copied to disk and indexes are not modified. Abort ends the transaction, releases locks, and invalidates application buffers.

5.1.2.2 Relationships & Referential Integrity

Objectivity/DB provides three techniques for building relationships between objects: associations, object references, and non-typed object identifiers. Associations are the preferred technique and should be used for all new code development. Object references may be useful when modifying an existing C++ program to access Objectivity/DB. Non-typed object identifiers are simply a means of storing an object identifier (i.e., a reference to an object) in an object.

Associations provide a means of logically linking persistent objects together. Associations may be either uni-directional or bi-directional. Uni-directional associations are a link from one object to another without an associated inverse link. Objectivity/DB does not maintain referential integrity to objects that are referenced by uni-directional associations. Bi-directional associations provide two way linking of

objects and support the concept of referential integrity. If an object is deleted, all references to it along bi-directional associations are removed. Likewise, if an object is moved, all references to the moved object are updated for bi-directional associations but not are not updated for uni-directional associations (note that Objectivity/DB object identifiers are location specific, thus moving an object requires giving it a new object identifier and all references to it must be updated).

Associations may be one-to-one, one-to-many, many-to-one, and many-to-many. Associations are defined in the DDL. Processing the DDL file results in generation of member functions for the class containing the association, which allows an association to be set, queried, deleted, existence tested, and, for '-to-many' associations, iterated. '-to-many' associations are in a first-in-first-out (i.e., historical) ordering and allow duplicates (i.e., bag semantics). Associations support the concept of Composite Object as described below.

All associations for a given object are normally stored in a single storage structure external to the object itself. This allows new associations to be added to a class without requiring modification of the objects of that class. Associations may be specified as 'inline' in the DDL meaning the links to other objects are to be stored as part of the object itself, separate from the other associations for the object. Inline associations require less space and in some cases provide faster access to the referenced object.

Further optimizations are possible by specifying that associations are to be represented as 'short' associations. This saves space but requires that all associated objects be stored close to the object containing the association (thus eliminating location transparency).

Object references provide a type-safe access to persistent objects. Object references may be embedded in persistent classes (handles may not be embedded). Object references are supported by a set of class definitions generated by the DDL processor for each persistent class defined. The generated reference classes provide operations for accessing a persistent object at the other end of the reference. Referential integrity is not automatically maintained for object references. Access overhead for object references can be eliminated by using the reference to retrieve a virtual memory pointer to the persistent object. This pointer will be valid for the duration of the enclosing transaction.

Non-typed object identifiers (OIDs) are a means of storing object identifiers in persistent objects. OIDs, defined by either class ooId or ooShortId, can be operated upon to test for null and equality, and to perform assignment to and from object handles. OIDs are untyped so it is up to the application to ensure the proper typed handle is used in assignment operations. OIDs cannot be used to directly reference objects; an assignment to a handle is necessary. Referential integrity is not provided by the OID mechanism.

5.1.2.3 Composite Objects

Objectivity/DB supports class attributes (i.e., data members) whose type are another class. This nesting of objects provides a direct mechanism for building composite objects. Delete and lock operations applied to the outer objects are also applied to the nested object(s).

The concept of composite object can also be supported by Objectivity/DB associations. Associations have a propagation property which specify if delete and lock operations should be propagated to the associated objects. There is no mechanism to automatically propagate copy or equality operations. Objectivity/DB does provide the virtual function ooCopyInit, which is invoked when an object is copied. The application can implement this function for a particular class to provide a deep-copy operation, by copying objects across association links as desired.

Associations have a copy property that specifies the handling of associations when an object is copied. The associations of a copied object may be either dropped, moved, or copied:

- o Dropped - original object retains all associations, new object has no associated objects.
- o Moved - original object has its associations removed, new object receives the associations of the original object.
- o Copied - original object retains all associations, new object receives the same associations as retained for the original object. Note that the class linked by a copied association, if its a bi-directional association, must view that association as a '-to-many' association.

Each association has its propagation and copy property specified in the DDL where the association is defined.

5.1.2.4 Location Transparency

Objectivity/DB provides to the application developer the opportunity for complete location transparency. An object may reference any other object without regard to its location in the database. There is no syntactic difference when referencing an object that is located in remote segments of the database. However, Objectivity/DB provides many optimization features that reduce the location transparency of object references. Object references and object identifiers may be used in default mode that can access any object in the database. An optimized, or 'short' mode, is provided for space efficiency when the application knows the referenced object will be located in the same container as the referencing object.

Objectivity/DB provides a move member function for moving an object to a new location within the database. This may be useful in order to improve clustering, for example, which increases performance. Moving an object causes it to be assigned a new object identifier. Moving an object results in automatic update of any bi-directional associations (i.e., the associated objects are updated to reflect the new object identifier of the moved object). Uni-directional associations are not automatically updated. The application must ensure that these associations are correctly rebuilt. Objectivity/DB provides ooPreMoveInit and ooPostMoveInit virtual functions for application defined actions. These can be used to programmatically update uni-directional references to the moved object.

5.1.2.5 Object Versioning

Objectivity/DB provides a basic version control policy as well as low level facilities to implement application specific version control. Support for configurations is programmable using drop/move/copy semantics for the associations of a versioned object similar to that described for copied objects.

Traditional linear and branch version control is provided by Objectivity/DB. Versioning is controlled by the Handle mechanism, which maintains status as to the versioning state of the referenced object. A handle may be set to 'no version,' 'linear version,' or 'branch version.' When an 'update' is performed on a handle, signaling that the referenced object has been modified and should be stored to the database, either the original object is modified or a new version is created. Objectivity/DB maintains each version as a separate object, with implicit links to associated versions. A genealogy is the set of all object versions for a given object. A default version may be set for a genealogy.

Use of versioning is application specific. In general, an application adjusts the versioning state of a handle to control when new versions are created in response to update operations.

New versions are simple bit-wise copies of the original objects with a DDL specified adjustment of associations. Similar to association handling during object copying, a newly versioned object may:

- Drop associations, meaning the new version is not related to objects along the association,
- Move associations, meaning the original version has its associations removed and the new version is associated to those same objects,
- Copy associations, meaning both the original and new version are related to the same objects.

When a new version is created, the C++ copy constructor and assignment operator are not invoked. Objectivity/DB provides the virtual function ooNewVersInit as a means of performing application specific processing needed when a new version is created.

Member functions on handles provide a means to set the versioning status, to locate the next or prior version, and to set or retrieve the default version for a genealogy.

Objectivity/DB provides a set of Handle member functions for developing custom version control mechanisms. For each Handle, the DDL processor defines a set of associations that are provided to support versioning. Some of these associations are used to implement branch and linear versioning (e.g., associations to next and previous version). Additional associations are provided for referencing all versions derived from a specific version and all objects from which a version was derived. Additional associations relate versions to a genealogy object that tracks all versions for a given basic object.

These associations are available for the application to define custom version semantics. For example, the association relating a version to all the objects from which is was derived can be used to merge branch versions. The DDL processor generates member functions to operate upon these associations.

5.1.2.6 Work Group Support

Objectivity/DB provides a check-out facility that could be used as part of a work-group application. When an object is 'checked-out' a persistent lock is placed upon the object (recall it is actually placed upon the object's container). Checking out an object checks out the entire container. Objects may be checked-out for read or for update. Checked-out objects are available for normal processing by any application executing with the same user-id as that which checked the object out. All other processes can access the checked-out object only under normal concurrency control mechanisms. When an object is checked-in, any updates made to it are then visible to other database users. This facility could be used to support a level of work group support whereby a user can persistently check out a section of the database and operate on it autonomously - across many transactions and application executions. While checked-out, other users may read the information, knowing that it is in the process of being modified. When completed with some task, the information can be checked back into the database and made available to other users.

5.1.2.7 Schema Evolution

Objectivity/DB does not automatically update objects in response to schema modifications. Instead, facilities are provided to aid in application controlled schema evolution. Both lazy and aggressive evolution strategies may be used.

Objectivity/DB provides a pragma rename construct in the DDL language in order to simplify the schema evolution process. This construct is used to rename an existing set of class definitions so that old and new versions of a class (which is being changed) can be defined in the same C++ program. Using the rename pragma results in generation of a set of header and source files for the original form of the class with a different name. The rename pragma has an option that specifies all data members of the class are to be made public. This simplifies the process of reading data members from objects of this class. Once a renamed set of class definitions has been defined, the DDL file may be modified to represent the new

class definitions, and the DDL processor can be run to generate header and source files as appropriate. Notice that without this rename facility, either a manual or user-defined semi-automatic (i.e., an awk program) method would be needed to modify the original schema's header and source files so that they may be used in a program with the new schema's header and source files (in order to avoid name conflicts in the C++ compilation process).

Under an aggressive evolution strategy, the next step is to develop a special application which performs object evolution. This application would create objects of the new class definition based upon the contents of the already existing objects from the old class definition. This program makes use of Objectivity/DB iterator constructs to find all instances of a class and/or to iterate across associations. Application knowledge of the inter-connection of objects can also be used to find objects which require conversion. This application is responsible for finding all instances of the modified classes, and for each instance:

- create an instance of the new class,
- copy the data members of the original object to the new object,
- transfer all associations from the original object to the new object, and
- delete the original object.

Transferring associations from the original to the new object is a simple matter if bi-directional relationships are used. Each object being modified has access to each object referencing it. If uni-directional relationships exist to the modified object then it is more difficult to find all references to the objects being evolved. Application knowledge may make identification of these relationships straight-forward. In the worst case, a database sweep may be necessary visiting all objects of all classes which have associations to the modified class.

A lazy evolution strategy would use the same approach, but instead of a separate application program performing a one-time evolution, the logic would be embedded into the application programs so that objects can be modified on-demand. This is a highly complex task, especially if the schema is modified several times.

One critical concern is the correctness of the application code which performs the schema evolution. It must be guaranteed to be correct since it is effectively a one-time shot at upgrading the contents of a database (although backup copies would, of course, be made prior to the update). For this reason, this code must be carefully tested.

Considering the schema change operations identified in the Evaluation Criteria section of this report, the application controlled evolution process would be required to:

- add an attribute to a class,
- remove an attribute from a class,
- change the type of an attribute,
- alter the relationship properties for a relationship attribute,
- add a new superclass to a class,
- remove a superclass from a class,
- change superclass order for a class, and
- remove a class.

Non-inline associations may be added to a class without requiring evolution of the instances of that class. This is because associations, which are not specified as inline in the DDL, are normally stored in a separate storage structure from the object itself. Thus, adding an association has no effect on the memory representation and content of the existing instances of that class. Deleting an association, likewise, requires no adjustment to instance memory representations, but any existing links along the deleted association will have to be removed, thus a schema evolution is required.

Note that except for the automated changes described for non-inline associations, Objectivity/DB provides no support for automatic schema evolution (e.g., adding a scalar typed attribute and assigning a default value, deleting a scalar attribute).

Changes to attribute and method names and to attribute defaults require modification of DDL, application programs, and queries, but should not require migration of databases. Changing a method implementation requires no change to existing databases or DDL. Changing the name of a class can be handled by the rename facility or under application control as described above.

An alternative form of schema evolution is possible for well-defined changes on relatively small databases. Objectivity/DB provides a dump and load facility which produces an ASCII representation of a database via a dump, and produces a database from an ASCII representation via a load. Making a schema change can be accomplished by dumping the database using the original schema definitions, followed by a text based modification of the ASCII representation of the database to reflect the modifications being made to the schema. After actually modifying the schema (via DDL processing) and the applications, the database may be loaded from the textual representation.

5.1.2.8 Runtime Schema Access/Definition/Modification

Objectivity/DB maintains a persistent database representation of the class definitions that have been processed by the DDL. There is no public interface to this representation, thus the schema may not be queried or modified at runtime.

5.2 Evaluation of ONTOS DB 2.2

Information presented in this section was derived from the technical documentation set for ONTOS DB 2.2 and discussion with technical representatives of ONTOS, Inc. All evaluations are based on the C++ application interface to ONTOS DB.

This section provides a detailed evaluation of application development issues and selected advanced topics.

5.2.1 Application Development Issues

5.2.1.1 Developer's View of Persistence

ONTOS DB provides persistence on a class basis. ONTOS DB provides the class Object which defines the behaviors for persistent storage of objects. Any class whose instances are to be stored persistently must inherit from class Object. All instances of a class which inherit from Object are persistent under ONTOS DB.

Persistent objects under ONTOS DB are accessed via pointers in an application. ONTOS DB provides a number of functions which retrieve an object from the database and place it in an application's memory space. These functions take as input either an object's name or a reference to the object (see Section 5.2.2.2, Relationships & Referential Integrity). These functions return a pointer to the retrieved object. The returned pointer may be used just like a pointer to any non-persistent C++ object. Via these pointers, data members may be accessed and member functions may be invoked.

Under ONTOS DB, bringing an object into memory is termed activation. Writing objects from memory back to the database after a change has occurred, is called deactivation. All object accesses under ONTOS DB apply a default lock mode to the accessed object. The default lock mode is dependent upon the type of transaction that the application is currently executing. Each of the functions which activate objects (i.e., read objects from the database to memory) provide a parameter for the user to override the default locking mode.

Objects that are modified by the application must be explicitly deactivated (i.e., written back to the database). ONTOS DB provides interfaces for deallocating individual objects and aggregate objects. These interfaces include parameters for specifying whether the objects are to remain in the application's address space (or whether they are to be deallocated from that address space). When an object is explicitly deallocated, ONTOS DB implicitly acquires a write lock on that object. Notice that under ONTOS DB, objects that are acquired with a write lock are not automatically written back to the database.

5.2.1.2 Application Development Process

ONTOS DB applications are developed by the following steps:

1. Configure a logical database using the ONTOS DBA Tool (see Section 5.2.1.3.1, Database Administration Tools).
2. Design the database schema either directly in C++ header files or using the ONTOS DBDesigner Tool (see Section 5.2.1.3.2, Database Design Tools). Database schemata are defined in standard C++ without any ONTOS DB specific extensions. ONTOS DB provides the class Reference, which may be embedded in a persistent object specification, for describing relationships between objects (see Section 5.2.2.2, Relationships & Referential Integrity). ONTOS DB requires a few additional member functions to be defined in the C++ header files for activating objects, deactivating objects, identifying an object's class, etc.
3. Process the C++ header files which define the database schema using the ONTOS DB classify utility. This tool reads the header files and creates object representations of the persistent classes in a selected database. These object representations are required so that ONTOS DB can persistently store instances of these classes.
4. Develop the application by providing implementations for class member functions, free functions, and a main routine. This task is similar to any other C++ development with the exception of the need for explicit object activation and deactivation calls.
5. Process the application files using the ONTOS DB cplus utility. This utility is a pre-processor to a standard C++ compiler and is configured to invoke that compiler upon completion. The cplus utility modifies constructor and destructor member functions so that they properly read and write objects from the database.
6. Link the compiled source using the ONTOS DB object libraries.
7. Test and debug the application as would be done for any normal C++ application. Once completed, performance tuning can be performed as needed to meet application performance requirements.

5.2.1.3 Application Development Tools

5.2.1.3.1 Database Administration Tools

ONTOS DB provides the DBA Tool for performing all database administration tasks. DBA Tool commands may be invoked directly from the command line. Database administration tasks cannot be performed from within an application.

ONTOS DB provides the concept of physical and logical databases. Databases are composed of areas which may be located at sites distributed across a local area network. A database administrator creates areas, distributed throughout the network, as needed by database applications. Each physical database is composed of a single kernel area which contains definitions of ONTOS DB classes (i.e., schema information) and may contain instances of those classes. All other areas that comprise a physical database (i.e., non-kernel areas) store only instance data. A physical database includes all areas that are created and associated with a specific kernel area. Logical databases are composed of a subset of the areas in a physical database. Logical databases are named.

The DBA Tool is a command line oriented tool providing the database administrator the following capabilities:

- Database configuration - allows for the creation of kernel areas and non-kernel areas, as named files to be served by processes on specific host machines with specific cache sizes. Allows for identification of logical databases and for the associations of areas to one or more logical databases. Provides the ability to move areas to new locations within the networked file system. Various show commands are provided for understanding the current mapping of areas to databases and host machines. Areas may be removed from logical databases; logical databases may be removed. Areas which actually store data may not be removed.
- Database recovery - the DBA Tool provides the ability to recover areas and/or logical databases (i.e., multiple areas) that were corrupted or are inaccessible due to system failures.
- Database backup and restore - is performed on ONTOS DB databases by using operating system facilities for copying database area files, database journaling files, and the Registry file. Physical databases must be backed up in their entirety. The DBA Tool can be used to identify which areas comprise a physical database. Given a complete backup of a 5.1.2 Advanced Topics

5.2.1.3.2 Database Design Tools

ONTOS, Inc. provides a graphical database design tool called DB Designer. DB Designer provides the ability to rapidly construct class definitions in an interactive manner. DB Designer will automatically generate C++ header files for the classes so defined. These header files include constructor functions for each class and accessor functions for setting and getting the values of private data members. (DB Designer can also be used to browse the instances of a class that are stored in a database, see Section 5.2.1.3.4, Database Browsing Tools.)

DB Designer provides the ability to create new classes, to define or modify properties of a class, and to delete classes. Note that properties may be added or modified only if no instances of the class exist. Similarly, classes may be deleted only if no instance exist and the class is not referenced from other class definitions. DBDesigner has some limitations on the types that may be used for new data members. Most significantly, abstract references may not be defined in DB Designer (see Section 5.2.2.2, Relationships & Referential Integrity, for a definition of abstract references).

As described in Section 4.2, ADT PTech from Associative Design Technology can be used with ONTOS DB. ADT PTech is a graphical, object-oriented modeling tool. ADT PTech can generate C++ header and source files, compatible with ONTOS DB, from a user's object-oriented model. These generated files can be used as the basis for a persistent application built on top of ONTOS DB.

5.2.1.3.3 Source Processing Tools

ONTOS DB provides two source processing tools, classify and cplus. Classify parses C++ header files and generates a database representation of the classes whose instances are to be persistent. cplus is a C++ preprocessor which appends additional C++ code to client application code so that it can be used with an ONTOS DB database.

Classify analyzes C++ header files for class and structure declarations, and generates Type, Procedure, and PropertyType definitions for representation of C++ classes (and structures), member functions, and data members, respectively (see Section 5.2.2.8, Runtime Schema Access/Definition/Modification). Type, Procedure, and PropertyType definitions are required so that ONTOS DB can maintain instances of classes persistently and allow for programmed (i.e., through an ONTOS DB interface) invocation of member functions and access of data members. The classify utility uses a control file and/or command line options for:

- Specifying that classes have type extensions (i.e., an index across all instances of the class).
- Controlling what area of the database is to store the instances of Type, Procedure, and Property which represent the class definitions.
- Identifying data members which are to have indexes built upon them.
- Specifying attributes for data members as follows:
  - is_unique - all instances of the class must have a unique value for this data member,
  - is_required - all instances of the class must have a value for this data member, and
  - has_inverse - for reference data members (see 5.2.2.2, Relationships & Referential Integrity), specify that when a reference is created for a data member, an inverse reference is created for some other data member.

Data member attribute is_unique, is_required, and has_inverse can only be checked and maintained by ONTOS DB if instances are modified programmatically (see Section 5.2.2.8, Runtime Schema Access/ Definition/Modification). Direct modification of instances via C++ code is not visible to ONTOS DB and thus cannot be trapped in order to check that these attributes are enforced.

- Controlling whether or not member functions have Procedure representations (these are only needed if Procedures will be invoked via an ONTOS DB interface, see Section 5.2.2.8, Runtime Schema Access/Definition/Modification).

cplus is a C++ preprocessor which must be invoked as part of the source compilation process on all source files which define persistent classes. cplus generates additional C++ code in a temporary copy of the input source code. Compiler directives are used by cplus so that the original source file name and line numbers are visible to the compiler, debugger, etc. cplus generates construction wrappers for each persistent class. A construction wrapper is code which performs object activation (i.e., moves an object from the database into the client application's address space). Additionally, function bindings are generated to allow Procedures to be invoked programmatically and to provide access to the construction wrappers. Function bindings provide a mapping from Procedure objects and construction wrappers to the address of their associated code.

5.2.1.3.4 Database Browsing Tools

ONTOS DB Designer tool may be used for browsing the contents of an ONTOS DB database. Both the database schema and data instances may be browsed using DB Designer. DB Designer is an X windows application providing the user a multi-windowed environment for simultaneously viewing many different aspects of a database's contents.

DB Designer provides a mechanism to view the hierarchical relationship of all classes stored in the database. This includes user defined classes, application support classes provided with ONTOS DB (e.g., List, Set), and classes which implement ONTOS DB (e.g., Object, Type). The display of the class hierarchy can be expanded, pruned, and re-organized at the user's discretion.

Class definitions may be browsed in DB Designer. Information available while browsing a class definition includes:

- class name and list of direct superclasses,
- public, private, and protected data members defined in the class and in inherited classes, and
- interfaces of public, private, and protected member functions defined in the class and in inherited classes.

While using DB Designer, a user is provided with complete control over which of the public, private, and protected information is displayed and over the amount of inherited information displayed. DB Designer allows a user to easily open browsers on other classes by clicking on class names anywhere they appear (e.g., in a class hierarchy diagram, in a list of superclasses). Navigation to classes related by abstract references is not possible since these are implemented as embedded Reference objects in ONTOS DB (see Section 5.2.2.2, Relationships & Referential Integrity).

DB Designer provides an instance browsing capability. Instances are displayed showing data values for public, private, and protected data members defined in the instances class and superclasses. Again, the user has complete control over which of the public, private, and protected information is displayed and the amount of inherited information displayed. Navigation to other instances is provided by clicking on direct or abstract references in an instance browser.

DB Designer can be used to create new instances and to modify the values of instance data members. For classes defined in DB Designer, the automatically generated class constructor is used to create new instances. For classes defined by the classify utility, the DB Designer tool may be relinked with application code in order to provide access to custom constructor methods which are invoked when creating new instances.

DB Designer provides access to objects by name. ONTOS DB provides a directory-based naming capability. Naming directories may be defined in a hierarchical name space. Objects may be provided names in one or more of these naming directories. Applications may use this facility to provide named access to a few key objects of the application (from which all other objects are accessible). DB Designer provides access to the hierarchy of naming directories and to the names within each directory. New name directories may be defined. All names stored in a directory may be listed and, by selecting a specific name, the object associated with that name may be browsed.

5.2.1.3.5 Debugging Support

ONTOS DB provides no explicit facilities which support interactive application debugging. Standard C++ debuggers may be used with ONTOS DB. DB Designer may be used to browse and modify the

contents of instances, but not during a debugging session since both the debugged application and DB Designer would typically be in a transaction attempting to access the same object.

The ONTOS DB 2.2 Developer's Guide, one of the reference manuals used during this evaluation, provided debugging tips for unhandled exceptions, for segmentation faults and bus errors occurring in ONTOS DB routines, and for executing unexpected functions. ONTOS DB uses exception conditions as a mechanism for informing the application of errors. Exceptions that are not handled result in an error message being printed and program termination. The Developer's Guide provides information on where to set breakpoints and how to interpret the runtime stack to determine where the exception was raised.

Segmentation faults and bus errors in ONTOS DB routines are often the result of passing null or invalid pointers or References to the routine. The suggested debugging technique is to run the program to failure in the debugger, then inspect the call stack to determine where the first entry into an ONTOS DB routine was made. Inspect each of the parameters passed to this routine for validity. The documentation includes information on the expected format of memory pointers to objects and for References.

Execution of unexpected functions can result when an object's virtual table (i.e., the table used for dynamic binding of messages to methods) is corrupted or out of date. This may occur due to changes being made to class specifications and not all users of the class being recompiled. Additionally, casting an object to an incompatible type can cause similar problems.

5.2.1.3.6 Performance Tuning Tools

The ONTOS DB documentation set includes a chapter on designing for performance. The chapter identifies application design and implementation alternatives which affect performance and provides guidelines for selecting the best strategy. This section reviews some of the techniques suggested for improving performance.

Section 5.2.1.4, Class Library, defines the aggregate data structure classes supplied with ONTOS DB for application construction. ONTOS DB documentation describes the storage and access architecture for each of these components. This information can be used by application designers to select the component which will provide the best performance based on the expected usage pattern of that component.

ONTOS DB provides an automatic type extension facility (i.e., an index of all instances of a class). Type extents are necessary if an application wishes to:

- iterate all instances of a class using the ONTOS DB supplied InstanceIterator class,
- browse the instances of a class in DB Designer, or
- use the ONTOS Object SQL facilities for performing SELECT operations over the instances of a class.

Type extensions are expected to add considerable performance overhead. Significant application speed can be achieved by limiting the classes which have type extensions maintained for them.

ONTOS DB provides a hierarchical object name space. This facility is useful for providing named lookup of key application objects. Naming adds both space and performance overhead.

Many ONTOS DB operations rely on pointers to the Type instances which represent the database schema (i.e., the persistent representations of C++ classes). Access to the Type instances require a named lookup. An application can improve its performance by performing these lookups only once per transaction (instead of a lookup for each use). Pointers to the Type objects, after being retrieved once at the start of a

transaction, can be stored in some globally accessible structure or interface for use throughout the remainder of the transaction.

ONTOS DB provides facilities for clustering the storage of objects that will typically be accessed together. Clustering reduces the number of disk reads needed to access objects since a segment of disk is brought into a database server's memory when a requested object is not already available. Subsequent accesses of objects in a segment that has already been read will require considerably less overhead.

Associated with clustering of objects into segments is the number of segments that a database server can concurrently store in memory. ONTOS DB provides database administration operations which allow the cache size for database servers to be specified. By increasing server cache size, more database segments may be stored in memory, thus reducing swapping of segments. Cache size should not be increased to the point that process swapping occurs.

ONTOS DB provides a Group Storage Manager class which provides optimal storage for large numbers of small objects which are typically accessed together. ONTOS DB implicitly uses instances of other Storage Manager classes for creating, activating, locking, deactivating, and deleting objects. The implicitly used storage managers activate objects (i.e., move objects from server to client) one at a time, although they are read from disk in segments. The Group Storage Manager activates objects in groups, thus reducing the overhead for object access. For applications that will access groups of objects together, performance improvements can be achieved by associating these objects with a Group Storage Manager. One drawback to this approach is that extra overhead results when using references between objects in different storage managers. Applications should be designed to limit the number of references between objects controlled by different storage managers. Application designers must evaluate object activation overhead versus intra-storage manager reference overhead when considering the use of Group Storage Managers.

5.2.1.4 Class Library

ONTOS DB provides an extensive class library for building persistent C++ applications. ONTOS DB provides the class Object for defining persistent storage of its instances (see Section 5.2.1.1, Developer's View of Persistence). Class Reference is provided for building relationships between persistent objects (see Section 5.2.2.2, Relationships & Referential Integrity). ONTOS DB provides a hierarchy of classes implementing aggregate data structures, exception handling, and storage of the database schema.

Aggregate data structure classes include abstract classes which define general behaviors and specializations of these as follows:

- Array and ArrayIterator,
- Dictionary and DictionaryIterator,
- List and ListIterator, and
- Set and SetIterator.

Classes are provided for representation of the database schema. These classes are referred to as Metaschema classes. Instances of the Metaschema classes represent C++ classes, data members, member functions, argument lists, etc. Section 5.2.2.8, Runtime Schema Access/Definition/Modification, describes the use of the Metaschema classes in more detail.

Additional class definitions include:

- o CleanupObj, Failure, and ExceptionHandler classes for handling exceptions and aborts.
- o Directory and DirectoryIterator classes, for building a hierarchical object name space.
- o ChangedObjects class provides a mechanism where ONTOS DB can track changes to an application defined set of objects.
- o OC_ClientCallback class provides callbacks to an application when significant operations are performed on the application's cache (e.g., at transaction start, checkpoint, commit, abort). This facility is useful for applications which share a common cache.

ONTOS DB provides a large number of free functions which provide functionality such as:

- o controlling access to a database,
- o transaction control,
- o manipulating the object naming environment, and
- o activation and deactivation of objects via direct references (see Section 5.2.2.2, Relationships & Referential Integrity).

5.2.2 Advanced Topics

5.2.2.1 Transaction and Concurrency Model

ONTOS DB provides three concurrency control policies: Pessimistic, Optimistic, and Time Based. Selection of a specific policy is made when the application starts a transaction. Transactions are started and otherwise controlled by invoking free functions as follows:

- o StartTransaction - a function to begin a transaction with input parameters specifying the concurrency control policy,
- o CheckpointTransaction - make all object updates made by the transaction up to this point permanent in the database. The current transaction continues without releasing any locks.
- o CommitTransaction - make all object updates made during the transaction permanent in the database and release all locks held by the transaction.
- o AbortTransaction - discard all object updates made during the transaction, or since the last checkpoint and release all locks held by the transaction.

Transaction commit and abort functions allow the application to specify whether the client application cache is purged or not. Purging the cache removes all objects and releases memory. Leaving the cache intact causes objects to remain in the client memory. These objects are available to the next transaction without the overhead of accessing the database server. Unfortunately, if any of these objects have been updated by other transactions, the current application will continue to see its version of the object which is now out of date.

ONTOS DB provides read and write locks at the object level. When objects are accessed, a lock is acquired with one of these two modes. Each concurrency control policy defines a default locking mode for all object accesses. The application may override the default locking mode by explicitly specifying an alternative locking mode for the object access. In addition, a write mode lock is acquired for objects that are returned to the server after being modified.

The concurrency control protocols determine if the lock requests of two different transactions conflict. When lock requests conflict, the requesting application can either wait or be informed by an exception

return. When lock requests result in a deadlock or a violation of transaction serialization, the requesting transaction is implicitly aborted. ONTOS DB concurrency control policies are as follows:

- Pessimistic - standard multiple readers or single writer object access. All objects are accessed with default locking mode of ‘write’. Conflicts are detected as locks are acquired.
- Optimistic - allows multiple readers of an object to proceed in parallel with a single writer. Default locking mode is ‘read’. Conflicts are detected at transaction commit time. Object updates are returned to the server only at transaction commit (at which time write locks are acquired and conflicts are detected). The optimistic policy is useful when transactions are expected to rarely conflict and system generated aborts are acceptable.
- Time Based - provides a compromise between the pessimistic and optimistic approaches. Multiple readers and a single writer of an object are allowed to proceed in parallel. Default locking mode is ‘read.’ Conflicts are detected periodically by having updated objects periodically returned to the server (at which time write locks are acquired and conflicts are detected). The time based policy is an attempt to prevent transactions that cannot be serialized from continuing for long periods.

ONTOS DB provides transaction checkpointing as a means of committing work performed in a transaction to the database without terminating the transaction. Checkpoints make all database updates permanent, after checking for serializability, and then allow the current transaction to continue with the same set of object locks and cached objects. Checkpoints provide a mechanism to limit the amount of work lost due to system generated transaction aborts.

ONTOS DB provides a nested transaction facility. Nested transactions are useful for implementing rollback features in applications or for protecting data integrity in concurrent applications. Nested transactions are also useful when constructing applications from independent modules, each of which provides its own transaction control. Nested transactions are invoked with the same set of interfaces as normal transactions, and thus can be composed without source modification. Nested transactions do require that the same concurrency control policy is used as in the parent transaction. Aborting a nested transaction causes all object updates made during that transaction to be discarded from the database. Aborting a nested transaction does not result in a rollback of object state in the client application cache. Thus, objects in the client cache are inconsistent and should be refreshed by accessing them from the server.

5.2.2.2 Relationships & Referential Integrity

ONTOS DB provides direct and abstract references for building relationships between objects. References are always uni-directional and one-to-one.

Abstract references are the preferred means of building relationships between objects. An abstract reference can reference an object that is in memory or on disk. The interface for accessing an abstract reference returns a memory pointer to the referenced object and brings the referenced object into memory if not already present. Thus, users of abstract references need never be concerned with whether or not the referenced object is in memory. An interface is provided for assigning objects and null values to abstract references.

Abstract references are formed by embedding the class Reference into a class which is to contain an abstract reference (i.e., use class Reference as a data member’s type). Abstract references are untyped; any class of object may be related across the reference. When dereferencing the abstract reference, the application must cast the returned pointer to another pointer to the appropriate class of object.

Direct references are similar to standard C++ pointers. Direct references are declared by using pointers to classes as the type of a data member. ONTOS DB provides an interface for bringing the related object across a direct reference into memory. The application is responsible for ensuring that the referenced object is activated prior to dereferencing the pointer which comprises the direct reference. An additional interface is provided for storing relationships into direct references. Direct references provide speed advantages, since they can be dereferenced by a normal C++ pointer dereference (i.e., no method or procedure invocation is required to traverse the reference). However, this can only occur if the application is sure that the directly referenced object is already in memory. Dereferencing a direct reference which does not refer to an active object will result in a runtime error. When objects are activated (i.e., brought into memory) direct references are swizzled as follows:

- all direct references in the newly activated object which reference an active object are assigned a memory pointer to the referenced object, and
- all direct references in active objects which reference the newly activated object are assigned a memory pointer to the newly activated object.

Since abstract and direct references are uni-directional, ONTOS DB cannot provide referential integrity across relationships. When an object is deleted, references to it are not automatically retracted. Dereferencing an abstract reference to a deleted object raises an exception which can be trapped and handled by the application. Dereferencing a direct reference to a deleted object is unpredictable; the program might fail, or a reference to some other object might be returned.

Section 5.2.1.3.3, Source Processing Tools, describes the Has_Inverse property that may be specified for references. This property informs ONTOS DB that two abstract references are the inverses of one another. When a reference is set programmatically, meaning by a call ONTOS DB metaschema interface (see Section 5.2.2.8, Runtime Schema Access/Definition/Modification), the inverse reference will be automatically formed. This provides automatic maintenance of bi-directional relationships. Note, however, that when abstract references are manipulated via the Reference member functions or when direct references are manipulated via the free functions which operate upon direct references (as described above), inverse relationships are not automatically updated.

5.2.2.3 Composite Objects

ONTOS DB supports class attributes (i.e., data members) whose type are another class. This nesting of objects provides a direct mechanism for building composite objects - although the nested objects have no external identity. Delete and lock operations applied to the outer object are also applied to the nested object(s).

ONTOS DB provides abstract and direct references for building relationships between objects. ONTOS DB does not support operation propagation across either abstract or direct references. Thus, interrelated objects do not support composite object semantics.

5.2.2.4 Location Transparency

ONTOS DB supports the concept of location transparency across a distributed database. Object access is not dependent upon its location in the network of interconnected databases. There is no syntactic difference in the code to access an object based upon its location.

ONTOS DB databases are composed of areas that may be distributed throughout a network. ONTOS's DBA Tool allows areas to be moved to new locations in the network. ONTOS DB does not provide any facilities to move objects or groups of objects to new locations.

5.2.2.5 Object Versioning

ONTOS DB provides no support for object versioning.

5.2.2.6 Work Group Support

ONTOS DB provides no specific facilities which support work group type applications (e.g., versioning or object check out mechanism). An application can hold locks on objects for extended periods of time using the transaction checkpoint mechanism. This facility could be used to provide some low level of work group support since an application could maintain locks on objects for extended periods of time.

5.2.2.7 Schema Evolution

ONTOS DB does not automatically update instances in response to schema changes. The ONTOS DB classify utility, described in Section 5.2.1.3.3, Source Processing Tools, is responsible for identifying changes to the database schema (i.e., changes to the C++ class definitions). This tool can be run in one of three modes:

- o Report any changes to database schema and abort the update of the schema.
- o Allow schema changes which can be handled automatically by ONTOS DB. If any unsupported changes are detected, classify reports the change and aborts the schema update. Changes which can be handled automatically are those which do not require modification of instances. These include changing the parameter profile of a member function, adding member functions, deleting member functions, and changing data members which are direct references into abstract references.
- o Report on all class changes and update the schema regardless of the acceptability of those changes. In this case, all instances of classes whose definitions have changed must be deleted from the database.

Modifying the implementation of member functions has no effect on the database schema and does not require migration or deletion of instances.

5.2.2.8 Runtime Schema Access/Definition/Modification

ONTOS DB supports runtime access, definition, and modification of the database schema. ONTOS DB provides the following set of class definitions for representing the database schema:

- o Class Type - represents a C++ class, storing its name, and relationships to its superclasses, data members and member functions.
- o Class PropertyType - represents a data member of a C++ class, storing its name, its data type, whether it is public, private, or protected, and in which class it is defined.
- o Class Procedure - represents a member function of a C++ class or a free function, storing its name, the class to which it belongs, its return type, and a list of arguments which define its signature (as an ArgSpecList).
- o Class ArgSpecList - stores lists of argument specifications (as ArgSpecs) for Procedure instances.
- o Class ArgSpec - represents the formal argument of a Procedure, storing its name and type.

Instances of Type, PropertyType, etc., are created when the classify utility processes C++ header files. The created instances provide a representation of the classes which make up the database schema. This information is required for ONTOS DB to persistently store objects and to move objects from the database to client memory and vice versa.

Other classes are provided for building lists of arguments used to invoke procedures at runtime, and for binding Procedures to the address of their code in an executable. Classes are provided to iterate across:

- o all constructors of a Type,
- o all Procedures of a Type,
- o all Properties of a Type,
- o all subtypes of a Type, and
- o all instances of a Type (requires a type extension to be specified for the class when it is processed by the classify utility).

ONTOS DB allows the programmatic creation and modification of Types. This is useful for applications which are to be tailorable or extendible by the user. ONTOS DB provides interfaces for:

- o programmatic querying of information about a Type, PropertyType, etc.,
- o programmatic invocation of a Procedure,
- o programmatic creation of new Types, defining their superclass and data members,
- o programmatic creation of new instances of a Type,
- o programmatic modification of a Type, changing the properties of data members, its location in the class hierarchy, adding or deleting data members, and
- o programmatic modification of a Procedure, changing its signature or method dispatch table.

One limitation on programmatic modification of Types is, as described in Section 5.2.2.7, Schema Evolution, instances of modified Types are not automatically updated to reflect changes in inheritance or data members. Thus, many of the changes that can be made to Types require that all instances of that Type be deleted.

5.3 Evaluation of VERSANT Release 2

Information presented in this section was derived from the technical documentation set for VERSANT Release 2 and discussion with technical representatives of Versant Object Technology Corporation. All evaluations are based on the C++ application interface to VERSANT.

This section provides a detailed evaluation of application development issues and selected advanced topics.

5.3.1 Application Development Issues

5.3.1.1 Developer's View of Persistence

VERSANT provides persistence on an object basis. Schema definitions, for each class that might have persistent instances, are stored in the database. In C++, objects are created via the new operator. new <classname.> creates a transient (i.e., non-persistent) instance of the specified class. new persistent <classname> creates a persistent instance of the specified class. VERSANT applications execute in the scope of a session database. The session database provides overall control of the applications access to data and its cooperation in concurrency control algorithms. An application may connect to other databases in addition to the session database. At any given time, either the session database or one of the connected databases is designated as the default database. Persistent objects created by an application are stored in the current default database.

VERSANT provides no application level control over the placement of persistent objects except within a specific database. VERSANT provides schema level directives for clustering of objects of one or more classes within a database. These directives are specified when the schema is processed into the database

and can not be changed. The clustering directives define the set of classes whose instances are to be clustered together. This is a coarse grained clustering that may or may not result in clustering of instances that are interrelated (and will likely be accessed at the same time). VERSANT performs storage compaction as a background activity to reclaim space that is not is use as a result of object deletions. The storage reclamation activity consults the clustering directives when relocating objects within a database.

All persistent object access within VERSANT is pointer-based. Operations on C++ pointers are overloaded by VERSANT routines to ensure that referenced objects are located in memory when accessed. A typical database operation selects a set of objects, perhaps all objects of a class or all objects which meet some criteria. The result of this operation is a list of references to objects. This list can then be iterated by the application, with some action being performed on each object referenced in the list. VERSANT automatically brings the referenced object into memory when necessary. An application can programmatically request that an entire set of objects be brought into memory at once. This can improve the performance of accessing a set of objects. VERSANT caches objects in the application process so that repeated accesses do not require repeated interaction with database server processes.

Applications which update the state of an object are responsible for informing VERSANT that they have done so. VERSANT uses this information to determine which objects must be copied from the application cache back to the database server and onto disk.

5.3.1.2 Application Development Process

Application development under VERSANT is similar to that for the OODBMS already described. Utility programs are provided for creating and configuring VERSANT databases. Database schemas are described textually by class definitions in C++ header files. VERSANT makes use of the C++ template facility for defining typed inter-object relationships (as opposed to vendor specific syntax used by some other OODBMS). VERSANT provides a pre-processor which is invoked as part of the normal C++ compilation script. The preprocessor generates VERSANT schema files based on the contents of the header and source files compiled. The preprocessor may also generate intermediate source files, which include C definitions for the C++ templates used for describing typed object references and typed data structures such as lists, sets, and dictionaries. These intermediate files are then compiled and the resulting object files are linked with VERSANT libraries to generate an executable. Note that C++ compilers which implement C++ templates and exception handling can compile VERSANT header and source files directly; intermediate files are not necessary. VERSANT supplies both production and debugging versions of their object libraries.

Prior to application execution, a VERSANT database must be populated with class definitions. The schema files, generated during the compilation step, are read by the VERSANT sch2db utility which creates class definitions in the targeted database. This utility can be used to create new class definitions and to modify existing class definitions (see Section 5.3.2.7, Schema Evolution). VERSANT relies on each database having its own copy of class definitions (for any object that will be stored in that database). For this reason, application developers are advised to populate all databases which their application will access with the latest schema definitions. VERSANT will automatically copy the schema to databases as needed during runtime but inconsistencies can lead to error propagations. VERSANT provides a database method that can be invoked by an application to force the schema definitions from one database to be copied to another, thus ensuring consistency of the schemata and avoiding the error propagations. VERSANT will also dynamically copy the schema to any database which does not have the schema for new object creations or when objects are migrated from one database to another (see Section 5.3.2.4, Location Transparency).

Application testing, debugging, and performance tuning is performed in a fashion similar to any other C++ development. The following sections deal with specific VERSANT support for these activities.

5.3.1.3 Application Development Tools

5.3.1.3.1 Database Administration Tools

VERSANT Database administration can be done from the command line by invoking utility programs; from a graphical tool provided by Versant Object Technology; or from an application. VERSANT databases may be stored as files or raw disk partitions. Databases are stored in data volumes (which store database schema and application data) and logging volumes (which store logging data). Data volumes do not expand automatically (but may be expanded as part of an administration task). Logging volumes expand automatically if stored as files (but not if stored as raw disk partitions).

VERSANT databases may be personal or group databases. A personal database may be accessed only by its owner. A group database may be accessed by anyone, or by a specific set of users (controlled by VERSANT administration activities). A typical application will access multiple databases and object references may span databases.

Administration activities available by VERSANT utility programs include:

- Database configuration - creating and deleting individual databases, moving databases within the network. Database creations are controlled by a set of parameters which are described Section 5.3.1.3.6, Performance Tuning Tools. Space may be added to the data volume of a database.
- Start/Stop Databases - access to a database is via communication with a database server process. This process is started when an application attempts to access a database. The overhead of starting this process and preparing a database for access can be removed from an application by starting a database outside the application. Similarly, a database may be manually shutdown.
- Backup/Restore - an entire database may be backed up to a named file. The database must be inactive during this process. An entire database may be restored from a backup.
- Import/Export - instances of specified classes may be exported from a database. References to instances in other databases are not exported. VERSANT exports in the STEP export format. A set of instances may be imported into a database.
- Remove Class/Instances - a class, including its subclasses and all associated instances may be removed from a database. All instances of a class may be removed (leaving the schema intact).
- Display/Modify Database Users - provides a means to control the group of users that may access a group database.

VERSANT provides a graphical database utility tool which can be used to perform all of the database administration tasks listed above. Additional tasks for managing database schema may also be performed by this tool.

VERSANT provides a programmed interface to invoke all the database administration activities listed above. In addition, a programmatic interface is provided to:

- archive a specified set of instances to another database,

- o open and close connections to databases (see Work Group Support), and
- o query database parameters such as percentage of cache free, percentage of free space on the disk, etc.

5.3.1.3.2 Database Design Tools

Versant Object Technology does not provide an interactive, graphical database design tool as part of the VERSANT product. Section 4.3 defined a number of object-oriented analysis and modeling tools which can be used as a front end to developing applications with VERSANT. For example, VERSANT has been integrated with Paradigm Plus from ProtoSoft. Paradigm Plus supports many popular object-oriented methods (e.g., Rumbaugh, Booch, Coad-Yourdon). The integration of Paradigm Plus with VERSANT supports generation of C++ header and source files to implement the object models defined in Paradigm Plus in a VERSANT database application.

VERSANT is integrated with a number of software development environments as described in Sections 4.3 and 5.3.1.3.3. These tools typically provide graphical browsing capabilities that aid in understanding existing parts of an application and are essential during applications development.

VERSANT provides a graphical type browser (see Section 5.3.1.3.4, Database Browsing Tools) which can also be used during schema definition as a means of viewing the definition of existing classes.

5.3.1.3.3 Source Processing Tools

Section 4.3 identified software development environments and C++ compilers that may be used to develop applications which access VERSANT databases.

VERSANT provides a C++ source preprocessor which analyzes C++ header files and an associated set of source files and generates VERSANT schema files. These schema files are used by a utility program to load database schemata (i.e., runtime class definitions). The utility can be used to define new classes in the schema or to update existing schemata (see Section 5.3.2.7, Schema Evolution). VERSANT's source pre-processor also generates additional C code (in temporary files) to implement C++ template types and exception handling facilities if not supported by the installed C++ compilation system.

5.3.1.3.4 Database Browsing Tools

VERSANT View is a graphical database browsing tool that can be used to view class definitions and to create, view, and edit instances of those classes. Additionally, the browser may be used to perform ad hoc queries on the contents of a database. The browser is built on the Motif widget set of the X Windowing System.

VERSANT View allows a user to access multiple databases and to coordinate his activities with other clients of those databases through the standard concurrency mechanisms. VERSANT View allows a user to open multiple windows to view different segments of a database. VERSANT View operates on sets of objects contained in a database (e.g., the set of all instances of a class). Sets of objects are displayed as a list with an entry for each object in the list. A particular object can be viewed in detail by double clicking on an entry. Set Windows are used by VERSANT View for displaying objects.

VERSANT View uses a standard presentation style for both class and instance objects (recall that class objects define the database schema and are used to model C++ classes; instance objects are persistent occurrences of C++ classes storing application data). An object's attributes are depicted with either the value stored by the object or a box which can be clicked upon to traverse to or expand the attribute.

Individual class objects are shown by displaying the class's superclasses, subclasses, attributes which VERSANT uses to describe C++ classes (e.g., unique identifier), and the user defined attributes of the class (e.g., name for a class modeling a person). Methods for the class are not displayed. A user can traverse the class hierarchy by clicking on super or subclass links. Each attribute defined for a class is depicted by its name and type. Again, a user may traverse the class space by clicking on attributes which represent relationships between the classes.

Individual instance objects are depicted with links to the instance's class and containing database, and for each attribute, its name and current value. Array and embedded object attributes (see Section 5.3.2.2, Relationships & Referential Integrity, for a definition of embedded objects) are displayed with buttons that may be clicked on to expand the data stored for those attributes. Links between objects can be traversed by clicking buttons displayed alongside the attribute representing that link. Attribute values for individual instances or groups of selected instances may be modified from within VERSANT View.

VERSANT View provides an Operations Window which is used to control the objects displayed in Set Windows. VERSANT View makes it easy to view all the class objects in a database, specific class objects, or the superclasses or subclasses of a given class. (Recall that multiple objects are treated as sets by VERSANT View and depicted in scrolling lists.) The Operation Window also allows users to perform ad hoc queries on the database. Query predicates are composed of simple expression on a class's attributes. Query predicates cannot invoke class behaviors. Queries are applied to one or more selected objects and return the resulting set of objects. Queries provide an easy way to form sets of instance objects.

5.3.1.3.5 Debugging Support

VERSANT applications may be debugged in the software development environments listed in Section 4.3. VERSANT provides explicit support for application debugging through its Check Facility, which can be linked into any application. The Check Facility can verify internal VERSANT constraints and can be used to display information about user specified objects.

Objects may be identified to the Check Facility via an address, a logical object identifier (valid across all databases and applications), or a link address (a reference to an object specific to a given application).

Information displayed via the Check Facility for a specific object includes:

- general status information, such as its identifier, what database it is in, its lock status, etc.,
- information on the versioning status of the object, and
- current data value for each attribute of the object.

The VERSANT Check Facility provides a set of routines which may be invoke from the debugger to print information about objects. Information describing one or more objects, specified by address, logical identifier, or link address, may be output to a file or to the standard error device.

VERSANT provides the method spy, defined on all persistent objects and on references to persistent objects, for causing information about objects to be printed from a program. The spy method prints information in the same format as described for the Check Facility.

VERSANT provides an automatic tracing facility that outputs information about all significant database events. With tracing enabled, VERSANT produces a file which includes the following information:

- information on calls to VERSANT database methods,

- o use of new and delete operators for creating and deleting objects,
- o information regarding the movement of objects into and out of the application's object cache, and
- o information on VERSANT's use of exception handling for propagating and catching errors.

5.3.1.3.6 Performance Tuning Tools

VERSANT uses a number of performance tuning parameters which can be set by the user. These parameters may be adjusted manually by editing parameter files or interactively through a graphical tool manager. The parameters are not adjustable at runtime. VERSANT does not provide any profiling statistics that will aid in determining optimal values for the parameters. VERSANT technical manuals give some guidelines and suggestions for setting the parameters.

Performance parameters are provided for server processes and client processes. All parameters are provided with default values if not otherwise specified.

Server process parameters are stored in a system file associated with a specific database. The parameters are read when a database is started (a database is implicitly started when accessed by an application, or may be explicitly started by a utility command, see Section 5.3.1.3.5, Debugging Support). Categories of server process parameters are:

- o Database creation parameters - control the sizes of files used for storing system data and logging information.
- o Functional parameters - control whether locking and logging are on or off and whether the server buffers are flushed to disk with every transaction commit.
- o Tuning parameters, including:
  - Initial heap size - a preallocation hint for the server processes heap.
  - Lock wait time - waiting time for a lock request, after which an error is returned to the requesting client.
  - Maximum page buffers - maximum number of disk pages to be buffered by the server.
  - Logging buffer sizes - maximum memory that can be used for buffering logging information.
  - Estimated transactions - an estimate of the maximum number of transactions that will be concurrently active on this database.

Client process parameters are stored in a user data file, also associated with a specific database. The parameters are read when the application opens the database for access. Some of the client process parameters are:

- o Initial heap size - may be used by applications which require a lot of heap space in order to reduce the number of times the heap must be expanded.
- o Object cache size - specifies the amount of memory used for storing objects within an application. Access of more objects than will fit in the cache causes objects to be swapped out of application memory.
- o Signal blocking - provides the ability to block signals, such as Ctrl-C.
- o Connect - specifies group databases that are to be connected with automatically when the associated database is accessed as a session database.

5.3.1.4 Class Library

VERSANT includes an extensive class library for building persistent C++ applications. The VERSANT class library includes classes:

- o PObject, which provides the ability to store an object persistently.
- o PVirtual, which provides the ability to compare objects by value.
- o PDOM, for manipulating databases, such as connecting an application to multiple databases, performing administration tasks on databases, getting and modifying database control information. Methods on class PDOM are also provided to load selected objects from the database and to migrate objects between databases.
- o PClass, for defining the database schema, as per a set of C++ class definitions. PClass also supports the definition of class indexes.
- o Link, LinkVstr, BiLink, and BiLinkVstr, for managing typed uni-directional and bi-directional relationships between objects.
- o PPredicate, for describing query predicates which can be used to retrieve a specific set of objects from the database.
- o VString, PString, VDate, and VTime, for storage of strings, dates, and times.
- o Data abstractions classes which implement typed sets, lists, arrays, and dictionaries, and iterators across these objects.
- o Container, a class which can store an arbitrary group of objects.

Many of the classes supplied with VERSANT make use of the C++ template facility in order to provide typed references or data structures. Application developers are provided with significant capabilities through these predefined capabilities.

5.3.2 Advanced Topics

5.3.2.1 Transaction and Concurrency Model

VERSANT provides a standard pessimistic concurrency control policy. Applications access VERSANT databases from within a database session. A session coordinates the execution of short transactions. An application can access one or more databases from within a session. A session is always executing a short transaction. When a session begins, a short transaction is started. Each commit or rollback of a short transaction implicitly begins the next short transaction. When a session ends, the current short transaction is committed.

VERSANT supports various forms of sessions. A standard session allows a single process to execute a sequence of non-nested transactions. A nested session allows a single process to execute a sequence of nested transactions. A nested, shared session allows multiple processes, spawned from an application, to each execute a sequence of nested transactions. VERSANT also supports custom locking sessions, which may or may not be shared, and use application specific locking semantics.

VERSANT locking is performed at the object level. VERSANT provides short locks for standard concurrency control capabilities. Persistent locks are provided for supporting work group applications, see Section 5.3.2.6, Work Group Support. VERSANT short lock modes are:

- o Null - allows other processes to read and write the object.

- o Read - allows other locks to read the object, but guarantees to the requester that the object's state will not be changed while he holds the lock.
- o Update - allows other processes to read the object, but guarantees to the requester that the object's state will not be changed while he holds the lock. In addition, it guarantees to the requester that he will receive the next write lock on the object. While holding an update lock, a process has read access to the locked object.
- o Write - guarantees exclusive access of the object to the requester.

VERSANT also provides intention locks to lock the class objects (i.e., schema definitions) when data objects (i.e., the instances of those classes) are being accessed. This ensures that class definitions are not modified, by the schema building utility sch2db, for example, as data objects are being accessed. Likewise, these locks ensure that data objects are not accessed while class definitions are being modified.

Read and write locks provide the standard semantics. Null locking allows processes to read data without abiding by the concurrency control. A process holding a null lock is provided no guarantee as to the value of the read object while he holds that lock (i.e., some other application may update the object in his address space and cause that update to be written to disk. The process holding the null lock continually sees the state of the object as it was when the lock was acquired and has no knowledge of the modification of that object by the other application).

Update locking is provided for processes that know they may change the data at some point in the future, but currently will only read it and wish to allow other applications to read it as well. An update lock ensures that no other process will acquire a write (or update) lock on the object, so that the state of the object, as read by the holder of the update lock, can be used in determining what type of update might be performed. Update locking is provided to increase the level of concurrent access to objects. Update locking can be used by an application to provide a form of multiple readers/single writers concurrency control.

As applications access data, VERSANT implicitly acquires object locks. By default, VERSANT acquires READ locks for all object accesses. The application can change the default locking mode. There are no queuing alternatives for short locks. If the lock cannot be granted, the process will be blocked for a predefined amount of time. The blocking time is set as a database parameter and is not configurable for an individual application or lock request. VERSANT performs deadlock detection as locks are requested. Any lock request that cannot be granted prior to the waiting time will result in an error being propagated to the requesting process.

VERSANT provides low level routines for explicitly controlling object locking and caching. Typically, an application will query the database for a set of objects and then access the objects by iterating over the set of objects which result from the query. As each object is accessed, it is implicitly locked. The application may explicitly lock all the selected objects and even request that all objects be loaded into memory at once. As described in Section 5.3.1.1, Developer's View of Persistence, an application is responsible for informing VERSANT of its intent to modify an object. VERSANT provides the method dirty, defined for all persistent classes, which must be invoked prior to making any modifications to an object. Invocation of dirty results in a write lock being acquired for an object.

When short transactions commit or abort, all short locks are released. In addition, objects are flushed from the application's address space. VERSANT provides a CheckPointCommit, which commits changes to disk, but retains locks and objects in the application's address space.

Finally, VERSANT provides the concept of application savepoints. Savepoints allow an application to perform rollbacks to earlier states of a computation. When an application takes a savepoint, the current state of all persistent objects accessed by the application is written to a log file. The application may rollback to that state, causing all persistent objects to be returned to their prior state. Savepoints are an application facility and have no effect on the actual state of an object on disk. Objects on disk are updated only as a result of transaction commits.

5.3.2.2 Relationships & Referential Integrity

VERSANT provides typed inter-object relationships through the Link and BiLink template classes. (C++ templates allow class definitions to be parameterized). Link and BiLink classes are non-persistent and must be nested (or embedded) inside a persistent class (see Section 5.3.2.3, Composite Objects). The semantics provided by VERSANT Link and BiLink classes, assuming they were embedded in class X, are:

- Link<Y> - supports forming a one-to-one, uni-directional relationship from an object of class X to an object of class Y.
- LinkVstr<Y> - supports forming a one-to-many, uni-directional relationship from an object of class X to an object of class Y.
- BiLink<Y, attribute> - supports forming one-to-one, bi-directional relationships between objects of classes X and Y.
- BiLinkVstr<Y, attribute> - supports forming many-to-many, bi-directional relationships between objects of classes X and Y.

The attribute parameter defined in the BiLink class template identifies the corresponding BiLink attribute in the class identified by the Y parameter of the BiLink class template. A BiLink attribute may be paired with a BiLinkVstr attribute to form one-to-many relationships.

Relationships created and deleted in a BiLink attribute by an application are automatically created and deleted in the inverse direction by VERSANT.

When an object is deleted, VERSANT ensures referential integrity across BiLink relationships by removing all relationships to the deleted object. VERSANT cannot support referential integrity across Link relationships. Applications must ensure that references across Link attributes do not cause dangling reference problems when objects are deleted.

5.3.2.3 Composite Objects

VERSANT supports class attributes (i.e., data members) whose type is some other class. This nesting of objects provides a direct mechanism for building composite objects. Delete and lock operations applied to the outer object are also applied to the nested object(s).

VERSANT provides Link and BiLink types for forming inter-object relationships. VERSANT does not support operation propagation across Link and BiLink structures, thus interrelated objects do not support composite object semantics.

5.3.2.4 Location Transparency

VERSANT supports the concept of location transparency across a distributed set of databases. Object access is not dependent upon its location in the network of interconnected databases. There is no

syntactic difference in the code to access an object based upon its location. One caveat to VERSANT's concept of location transparency is that a process must be explicitly connected to all databases in which an object might be accessed. VERSANT provides facilities to make identification of databases to connect with easily specified by a user on the command line or in application startup files.

VERSANT supports the ability to move objects to different databases. VERSANT object identifiers are not location dependent, thus objects can be moved without concern of updating uni-directional or bi-directional references.

VERSANT database must contain the schema definitions for all classes whose instances will be stored within the database. Moving an object to a database tOne limitation on programmatic modification of Types is, as described in Section 5.2.2.7, Schema Evolution, instances of modified Types are not automatically updated to reflect changes in inheritance or data members. Thus, many of the changes that can be made to Types require that all instances of that Type be deleted.

5.3.2.5 Object Versioning

VERSANT supports object versioning as a means of tracking the history of changes to an object and also to increase concurrent access to objects. VERSANT supports both branch and linear versioning. VERSANT provides a number of facilities for creating and accessing versioned objects allowing applications to implement specific versioning policies.

All versions of an object are related in a version derivation graph. The first version of an object acts as the root of this graph. Each version in the graph is connected to the object from which it was versioned. Associated with each versioned object is a generic object. This object stores the derivation graph for a versioned object and maintains information about the most recent versions of the versioned object.

VERSANT provides version status levels for controlling access and update of versioned objects. Supported version status levels are:

- Transient - same as an ordinary object, the version may be modified or deleted.
- Working - used to denote stable versions. Working versions cannot be modified but can be deleted.
- Released - the version may not be modified or deleted.

An application can explicitly set version status levels (both upgrading and downgrading). Version status may also be set implicitly. For example, when a new version is derived from a transient version, the transient version is automatically upgraded to working status.

Versioned objects may be explicitly created by the application, or implicitly created as the result of checking in an object from a long transaction (see Section 5.3.2.6, Work Group Support).

New versions are managed as children of the object version from which they were derived. VERSANT allows an application to modify these parentage links so that a single version can depict the merging of multiple versions.

A few of the capabilities supported by VERSANT's versioning methods include:

- Specify that an object is to be versioned.
- Create a new version of an object. If the object has not been explicitly marked as a versioned object it is done so implicitly as a result of creating this first version.
- Delete a version.
- Access a specific version, the default version, or the latest transient, latest working, or latest released version.
- Retrieve the parents of a versioned object.
- Modify the parents of a versioned object.
- Query versioning status for an object or, from a generic object, about a version derivation graph.

5.3.2.6 Work Group Support

VERSANT provides long transactions and a checkout/checkin mechanism for supporting work group applications. As described in Section 5.3.1.3.1, Database Administration Tools, VERSANT provides the concept of personal and group databases. A personal database is accessible by only a single user. For non-concurrent applications, locking in personal databases may be turned off without sacrificing consistency, thereby improving performance. Group databases are provided for concurrent data access by multiple users. A typical work group application would be constructed so that it copies information from a group database into a personal database and then works on that data autonomously in the personal database. Checkout and checkin provide the ability to move information between group and personal databases for this purpose. Long transactions provide the mechanism to ensure the consistency of information that has been checked out of a group database.

Objects are moved from a group to a personal database by using one of several checkout methods. One form of checkout method is passed a list of objects to be moved to the personal database. This list can be formed explicitly by the application, as the result of a query, or by a getClosure method. The getClosure method finds all objects related to a given object, recursively traversing relationships to an application specified depth. A second form of checkout method can be passed a query predicate which it uses to identify the objects to be checked out.

Objects checked out of a database are locked (in the database they are checked out from) by a persistent lock. Persistent locks are implicitly acquired and released as part of the checkout and checkin process. A persistent lock survives short transactions, application executions, and system crashes. Persistent lock modes include null, read, and write. Null locks provide a snapshot of the checked out object and allow other users to continually access the object while it is checked out. Read and write locks have the typical meaning. For example, an object checked out with read access may be read in the group database by other users but not updated. Persistent locks are, of course, only granted if the group database has no conflicting locks, either short or persistent, on the objects being checked out.

Checkout operations allow the application to specify the lock mode and additional attributes of the lock activity. Locks may be either hard or soft. Soft locks can be broken by another application requesting a hard lock on the same object. Long transactions that have broken soft locks will result in errors being propagated to the application when the transaction is committed. A user can be notified by electronic mail if an application he is executing has had a soft lock broken.

Long locks can also be specified with a queuing parameter. Queuing options include:

- o Wait - suspend the process for an amount of time specified at startup time for the database server process.
- o No wait - inform process immediately if the lock cannot granted.
- o Reserve - allow process to continue executing and the inform the user via electronic mail when the requested lock becomes available.

Electronic mail can also be used to inform a user when an application he is using is holding a lock that has been requested by some other application.

Persistent lock mode, queuing option, and hard vs. soft can be configured on a per request basis by the application.

Objects can be checked back into a group database when a user has completed working with them. Objects checked out with a write lock cause the copy of the object to be updated in the group database when the objects are checked back in. Objects checked out with read locks cause no change to the object in the group database. Objects checked out with a null lock result in a new object being created in the group database. Checkin of a versioned object causes a new version to be created. Objects can be explicitly checked in by the application or can be implicitly checked in by committing a long transaction.

Long transactions are the mechanisms for structuring work group applications. Applications can start long transactions or join an existing long transaction when it begins its access to a database. By having long transactions span application executions, a work group application could be started and stopped several times while operating on the same set of information in a personal database. Assumably, after perhaps many days of design work, a user would complete some design task and check the results of his work back into the group database. This can be accomplished by completing the long transaction. When a long transaction commits, all objects checked out as part of that long transaction are implicitly checked back into their respective group databases. Abort of a long transaction causes no modification of objects that had been checked out. Both commit and abort of long transactions causes all persistent locks to be released.

5.3.2.7 Schema Evolution

Section 5.3.1.2, Application Development Process, introduced the use of VERSANT's sch2db utility. This utility populates VERSANT databases with schemas that have been derived from C++ class definitions. The schemas are object representations, based on instances of class PClass (we will call these class definitions in the remainder of this section). The sch2db utility can be used to create new class definitions or to modify existing class definitions. The utility can also be used to compare the database's view of class definitions with the latest compiled view of those class definitions.

The sch2db utility creates class definitions for any class in the input schema file not already in the database. The sch2db utility will (optionally) perform the following automatic schema evolutions:

- o Classes and attributes may be renamed.
- o Attributes may be added or removed. Attribute types may be any elementary type or class (i.e., embedded objects). Adding an attribute sets its initial value to 'null' or '0' for all instances of the class to which the attribute was added and to all instances of subclasses. Removed attributes have no effect on data values for other attributes.

- Leaf classes may be added or removed. When removing a class, the application developer must make sure that any classes which reference the removed class through Link and BiLink attributes have those attributes removed. Removing a class results in that class and all of its instances being deleted.
- Classes may have superclasses added or removed.

Automatic schema changes are applied by a lazy evaluation strategy. Instances are not updated as a result of invoking the sch2db utility, but instead the database schema is internally versioned and instances will be modified as they are accessed by applications.

Class behaviors can be modified by changing C++ source code. The automated schema evolution capabilities provided by VERSANT come close to matching the capabilities listed in the Evaluation Criteria section of this report. Unsupported capabilities such as changing the type of an attribute can be achieved by deleting an attribute and adding a new attribute in its place.

5.3.2.8 Runtime Schema Access/Definition/Modification

VERSANT supports runtime access and definition of schema. VERSANT provides the class PClass for representation of database schema. Instances of PClass are created and modified by the VERSANT utilities which load databases from schema files. VERSANT provides a number of methods which allow an application to access and/or modify the database schema. Methods are provided to:

- add classes,
- remove classes,
- rename classes,
- add attributes,
- remove attributes,
- rename attributes,
- remove all instances of a class,
- select a specific class definition,
- retrieve a list of all class definitions, and iterate across that list,
- extract a class definition (i.e., attribute and method information), and
- identify the superclasses and subclasses of a specific class.

Attribute information is passed to or from the above methods as a structured type. Information maintained for an attribute includes its name, type, and inheritance. Method information is currently not stored by VERSANT and returned as null when querying class definitions. When creating classes dynamically, an application may chose to store information about methods in an application specific representation. VERSANT databases will store application specific method information and return it when class definitions are extracted.

5.4 Evaluation of ObjectStore 2.0

Information presented in this section was derived from the technical documentation set for ObjectStore Version 2.0 and discussion with technical representatives of Object Design, Inc. All evaluations are based on the C++ application interface to ObjectStore.

This section provides a detailed evaluation of application development issues and selected advanced topics.

5.4.1 Application Development Issues

5.4.1.1 Developer's View of Persistence

ObjectStore provides persistence on an object basis. An object of any type can be stored persistently, including C types, C++ types, instances of classes, and pointers to any of these. When an object is created, it is created as either a persistent or a transient object. ObjectStore provides an overloaded new operator which allows applications to create persistent objects of a given type, in a given database or database segment. Initial values may be provided and an array of similarly typed objects may be created via the new operator.

All persistent object access is via virtual memory pointers. Thus, from the programmer's perspective, accessing a persistent object is no different than accessing a transient object. ObjectStore provides a unique persistent object access mechanism called Virtual Memory Mapping Architecture (VMMA). VMMA is built upon the virtual memory architecture of the host operating system. Accessing an object that is not in memory causes a virtual memory fault, which is handled by bringing the referenced object into the address space of the faulting process, and then continuing with the access that caused the fault. Objects are brought into memory in groups of one or more pages (application controllable). When objects are brought into memory, ObjectStore ensures that all virtual memory addresses stored within the objects are valid. In many cases, object pages are loaded to the same virtual memory address and thus all references are by default valid. In other cases, where a referenced object is currently not in memory, the virtual memory address where that object is expected to be loaded are reserved. Finally, in cases where referenced objects are already stored in different locations or their expected locations cannot be reserved, ObjectStore performs pointer swizzling as needed to ensure consistent object access. Of course, all of this processing is completely transparent to the application.

ObjectStore databases are accessed via entry points. Entry points can be defined by associating a name with a persistent object, or by explicitly declaring variables to be persistent using the ObjectStore DML facility. Typically, an application will access a small number of objects by name or persistent variable, and then traverse to all other persistent objects by following virtual memory pointers.

ObjectStore's VMMA automatically identifies modifications to persistent objects and causes these objects to be written back to the database upon transaction commit. This eliminates the need for application code to explicitly signal to the database that an object has been changed and should be written back to the database.

5.4.1.2 Application Development Process

Building applications on top of ObjectStore is similar to that described for the other OODBMS evaluated in prior sections of this report. Utilities are provided for creating and managing access to databases. Database schemata, described textually as C++ header files, can be developed manually or by a Schema Designer tool provided with ObjectStore (see Section 5.4.1.3.2, Database Design Tools). These headers can be composed of standard C++ source, C++ extended with parameterized types, and/or C++ extended with ObjectStore DML constructs. Source code for class implementations can be developed using standard techniques and C++ programming environments (see Section 4.4). Object Design offers a C++ compiler and an extension of the gdb debugger for building ObjectStore applications. Make files are typically used for compiling and linking programs. Standard debugging and testing techniques may be applied.

Like all other OODBMS products, some means of loading a database with the schema (i.e., class definitions) is required. ObjectStore uses the concept of an application schema database to support this process. An application schema database stores the schema definitions for all objects that might be allocated or accessed by an application. The application schema database is populated as part of the application link process. Schema definitions for all classes compiled as part of the application (including those from libraries) are loaded into the application schema database. At runtime, an application can access one or more application databases. An application database, referred to as a database from here on, stores data that is generated by execution of the application. All such databases must have schema definitions for the classes of all objects which are stored in that database. Databases are loaded with schema information in one of two ways:

- Batch mode - when an application opens a database, all the schema definitions from the application schema database are copied to the newly opened database.
- Incremental mode - when an application allocates an object, the schema definition for that object's class is copied from the application schema database to the opened database.

When an application opens an existing database, checks are performed that any schema definitions already in the opened database are consistent with the schema definitions in the application schema database.

Application schema databases are created as part of the application link process. An application schema database is actually built from one or more compilation schema databases. Compilation schema databases are loaded with schema definitions as individual source files are compiled. Construction of application schema databases and compilation schema databases is hidden from the user by the make files and tools provided with ObjectStore (see Section 5.4.1.3.3, Source Processing Tools).

5.4.1.3 Application Development Tools

5.4.1.3.1 Database Administration Tools

ObjectStore databases may be stored as operating system files or as ObjectStore Directory Manager files. The Directory Manager is an ObjectStore application which provides access to application databases. These databases may be stored as operating system partitions or raw disk partitions. The advantage of using the ObjectStore Directory Manager is that a logical hierarchy of databases is provided, thus providing an environment in which all databases can be tracked and managed. In addition, several special utility functions are provided.

ObjectStore provides database utilities which support both the database administrator and the application developer. (Some of these utilities may be used on databases stored as Directory Manager files only.) These utilities are available via the command line or through a programming interface. The following list briefly describes some these facilities.

Utilities provided for monitoring and administration of ObjectStore databases include:

- Start, stop, or verify that ObjectStore processes are running, These include server processes, cache manager processes, and the Directory Manager application.
- Force server checkpoints and to force flushing of application or server caches.
- Install schema definitions in a database.
- Set server and cache manager process configuration parameters.

Utilities provided for user-level manipulation of databases include:

- o Controlling file access such as owner, group, and permission.
- o Copy, move, and delete databases.
- o Export the contents of a database, import the contents of a database.
- o Identify the host which serves a named database.
- o List all databases in a directory.
- o Create and delete database directories.
- o Perform schema evolution (see Section 5.4.2.7, Schema Evolution).
- o Report the size of a database and the sizes of its segments.
- o Verify the pointers stored within a database, that none refer to transient or deleted objects and all refer to objects of the expected type.

Backup and recovery of ObjectStore databases must be performed off-line. Databases stored in operating system files are backed up using operating system commands. ObjectStore provide a similar set of utilities for backing up databases stored in Directory Manager files.

5.4.1.3.2 Database Design Tools

Object Design provides a graphical schema designer tool for interactively developing database schemas. The schema designer provides both a graphical and a textual view of the classes which make up a database schema. This tool operates on schema files, which are special data files used only by this tool. The tool has commands to generate C++ header files from the current representation of the schema as represented in this tool.

The schema designer provides graphic, text, and form based working areas. Edits made in one area are automatically reflected in the other areas. Using these three areas, the user is able to view, create, and modify classes, inheritance, relationships, data members, and member function specifications.

The schema designer’s graphic area allows the user to graphically create classes and specify the inheritance relationships between classes. The user has complete control over placement of the graphics that represent classes and routing of arrows which represent inheritance. Relationships between classes can also be graphically defined and their routing controlled. One-to-one, one-to-many, and many-to-many relationships can be formed by specifying type information in an associated form. The graphic depiction of the relationships is adjusted to represent their cardinality.

The schema designer provides a form based area, called the properties area, where additional information may be specified for classes, relationships, data members, and function members. For example, data member types may be specified, function member parameter profiles may be defined, and public, private, or protected attributes for data and function members may be specified.

The schema designer text area displays the textual representation of the schema which has been composed in the graphical and form areas. Relationships may be textually represented using parameterized C++ types or using ObjectStore DML syntax. The schema designer provides buttons for creating additional data members and member functions while viewing the textual representation of a class.

Section 5.4.1.3.4, Database Browsing Tools, describes the ObjectStore browser which also provides presentations of the database schema. This browser may be used to understand the existing database schema while designing additions to that schema.

5.4.1.3.3 Source Processing Tools

Section 4.4 identified software development environments and C++ compilers that may be used to develop applications on top of ObjectStore databases. In addition, Object Design supplies a compiler and debugger optimized for use in building ObjectStore applications. This compiler supports parameterized types, recognizes ObjectStore DML constructs and directly provides a C++ compiler supporting ObjectStore's DML for schema definition and associative queries. The Object Design compiler and debugger are used in the traditional manner for developing database applications, as described in Section 5.4.1.2, Application Development Process.

Object Design provides a schema generator tool that is used for populating ObjectStore database with schema definitions. The schema generator tool is normally invoked in a fashion transparent to the user through make file commands. The schema generator is required for applications built with the Object Design supplied compiler or with third party compilers. The schema generator is used during compilation to generate schema definitions for compiled C and C++ header files. Prior to linking, the schema generator is responsible for identifying and assembling all schema definitions needed for an application and loading these definitions into the application schema database, as defined in Section 5.4.1.2, Application Development Process.

5.4.1.3.4 Database Browsing Tools

Object Design provides a database browsing tool which allows a user to view the current schema and data contents of a database. The database browser provides three windows in which information is displayed. The main browser window lists the databases that are currently open for browsing and lists queries that have been formulated during this session. From this window, databases may be opened or closed for browsing, presentation format may be controlled, the second form of browser window, stack windows, may be opened, and queries may be edited and run (resulting in opening a stack window on the query result).

Stack windows are the primary means for viewing the contents of a database. When initially opened, a stack window can display either:

- the database schema,
- the entry points of the database (see Section 5.4.1.1, Developer's View of Persistence), or
- the result of a query applied to a database.

The stack window provides a hypertext like interaction style. Within any pane of a stack window, the user can select an object and the browser will display an expansion of that object in a subordinate pane. In addition, the user can formulate and execute queries from within the stack window.

The third window provided by the database browser is the version graph window. For databases which store versioned objects (see Section 5.4.2.5, Object Versioning), a version graph window graphically displays the relationships between each version of an object. Version graph windows are opened by a menu command from a stack window. This command is only available when an object which has versions is selected within the stack window.

The database browser allows queries to be formulated and executed against either the entire database or a selected collection of objects. Queries may be specified textually or with a query dialog. Queries are composed of one or more clauses, each of which is some expression on the data members of the class against which the query is being applied. Queries cannot invoke member functions.

5.4.1.3.5 Debugging Support

ObjectStore applications may be debugged in the software development environments listed in Section 4.4. ObjectStore is provided with an extended version of gdb, the GNU source level debugger distributed by the Free Software Foundation, Inc. The gdb debugger is a C source level debugger and thus many C++ facilities are not directly recognized. For example, expressions are evaluated using C syntax rules instead of C++ rules. Additionally, member functions must be invoked by prepending the class name and explicitly passing the target of the invocation as the first parameter.

Extensions to gdb implemented by Object Design aid in handling access to constructors, destructors, overloaded functions and methods of parameterized classes. This is necessary, for example, in order to set breakpoints on these routines.

ObjectStore provides a utility which can verify the pointers stored within a database. This can be a useful debugging aid to check that an application has left all the database pointers in a consistent state. The osverifydb utility checks that all pointers stored by objects in a database:

- do not reference transient objects,
- do not reference deleted objects, and
- do reference objects of the expected type, based on the database schema.

Any invalid pointers detected by the osverifydb utility result in a textual description of the location of the pointer (path to the object containing the invalid pointer), current value of the pointer, and expected type of the pointer.

Within an ObjectStore application, checking for illegal pointers can be performed at completion of each transaction. This checking is optional and controllable on a per application basis. The checking that is performed searches for pointers to transient memory and, if not explicitly allowed by the application, pointers to objects in other databases. Pointers to deleted objects or objects of incorrect types are not detected by this checking facility.

5.4.1.3.6 Performance Tuning Tools

Performance tuning of ObjectStore applications can be achieved through configuration parameters, command line utilities, and programmed control of server and client process options.

ObjectStore databases are accessed by application programs transparently communicating with database server processes and client cache manager processes. Server processes control all access to the database. Cache manager processes provide buffers of disk pages that are available to all the client applications on a single host. In addition, a Directory Manager process controls access to databases maintained in the ObjectStore file system. Performance monitoring and tuning capabilities are provided for each of these classes of processes.

ObjectStore provides the ability to cluster persistent objects on disk. Clustering is essential in order to reduce the number of disk reads necessary to access a related set of objects, and to increase the concurrency of multiple applications (by having non-related objects not clustered, and thus locked, together).

Object clustering is achieved by defining storage segments within a database, and object clusters within each segment. Segments are expandable storage containers within a database. Clusters are fixed size storage containers within a segment. Initially a database has two segments, one for schema data and one for application data (i.e., instances of the schema). All objects created by an application are stored in the

predefined application data segment. An application can create additional segments and clusters within a database in order to optimize the storage and access of objects within the database.

Objects are clustered when they are allocated by optionally specifying the cluster or segment the object is to be stored in. Overloaded versions of the new operator allow an object to be placed in a specific segment or cluster. (Functions are provided which return the segment or cluster of an object, thus eliminating the need to explicitly identify cluster and segment objects).

For applications which define segments and clusters, performance can be improved by specifying data fetch policies. The default data fetch policy attempts to transfer an entire segment to the client cache upon accessing an object (which is not already in the cache). If the entire segment will not fit in the cache, a user-specifiable number of bytes are retrieved. Alternative object fetch policies include a page mode, where each object access results in a user-specified number of pages being transferred, and a stream mode, whereby subsequent accesses of the same segment fetch an application configurable amount of data into the same client cache location. Stream mode is useful in applications that scan large data structures and do not wish to cause the entire client cache to be replaced as a result of the normal cache flushing algorithms. ObjectStore allows data fetch policies to be set for an entire database or for individual segments.

Utilities provided with ObjectStore for monitoring or affecting performance include:

- osdirmtr - prints process statistics, such as CPU time used, memory usage, page faults, and I/O operations for the ObjectStore Directory Manager process.
- ossvrmtr - prints process statistics, such as CPU time used, memory usage, page faults, and I/O operations for the ObjectStore Server process running on a specified host.
- ossvrsetcache - sets the data cache size for the server running on a particular host.
- ossvrsetchkp - controls time at which database logs (i.e., history of changes to the database) are flushed to database.

Parameters for controlling the behavior of the server process allow:

- specification of cache and buffer sizes,
- specification of block size for data transferred between the server and client,
- selection of a deadlock resolution strategy, and
- control over the logging of changes made to the database.

Cache manager process parameters provide for control over the directories where cache manager files are stored and hard and soft limits for disk space used in storing its data files. Client process parameters provide control over client cache size and location.

5.4.1.4 Class Library

ObjectStore includes an extensive class library for building persistent C++ applications. The ObjectStore class library includes:

- Class objectstore, which provides static member functions for controlling persistence, performance tuning, performance monitoring, and database administration tasks.
- Aggregate data structure classes collection, set, bag, and list, and iterators over these (called cursors). Both typed versions (using parameterized classes) and untyped versions of collections are provided.

- Classes for defining database schemas, see Section 5.4.2.8, Runtime Schema Access/ Definition/Modification.
- Classes for defining and executing queries.
- Classes for defining indexable data members of a class.
- Classes os_database_root and os_pvar, which define entry points into a database.
- Class os_reference, which can be used in place of memory pointers in order to form relationships between objects. Relationships formed with os_References are valid across transaction boundaries and between databases (whereas references formed with memory pointers are not always valid in these cases).
- Class os_schema_evolution, which provides an interface to the ObjectStore schema evolution facility (see Section 5.4.2.7, Schema Evolution).
- Class os_transaction, which defines transactions and provides member functions for querying their state and controlling their behavior. Transactions can also be controlled via macros or ObjectStore DML, but only when the transaction boundaries are contained within a single lexical construct. Instances of class os_transaction, allow transaction lifetimes to span lexical boundaries.
- Classes os_configuration and os_workspace which implement ObjectStore's version management facility.

Notice that ObjectStore does not include classes which provide persistent behavior. As stated in Section 5.4.1.1, Developer's View of Persistence, persistence of an object is independent of its class. Thus, no classes defining persistent behavior are required.

ObjectStore includes global functions and macros for some system level activities such as transaction control and persistent object allocation and deallocation (via new and delete operators).

5.4.2 Advanced Topics

5.4.2.1 Transaction and Concurrency Model

ObjectStore supplies a transaction which supports a traditional multiple readers/single writer concurrency control policy for access to individual objects. ObjectStore transactions are either read-mode, allowing read-only access to all objects accessed within the transaction, or update-mode, allowing read-write access to all objects accessed within the transactions. Applications must identify the access mode when the transaction begins. If an application attempts to update an object from within a read-mode transaction, ObjectStore raises an exception.

Sections 5.4.2.5, Object Versioning and 5.4.2.6, Work Group Support, describe alternative object access mechanisms that increase the concurrency available to multiple users of the same objects.

As described in Section 5.4.1.4, Class Library, ObjectStore transactions can be defined statically within a single lexical scope through the use of macros or DML statements or dynamically through the use of os_transaction objects. Regardless of how declared, transactions may be nested. Nesting is advantageous since each unit of code can be assured it is executing within a transaction by starting and ending that transaction. In addition, undoing portions of computations is easy to achieve with a nested transaction facility. Nested transactions must all be of the same mode, either read or update.

ObjectStore acquires locks implicitly as the result of an application's access to data. When an object is accessed, the page upon which it resides is locked. Locks are held until the outermost transaction

completes. Thus, as nested transactions complete, and their results become visible to the parent transactions, locks on the objects are retained.

When a transaction completes, all objects accessed during the transaction are released and the database is updated as appropriate. For transactions which span multiple servers, the two-phase commit protocol is used during commit. Upon transaction completion, the client cache is not purged. Although references to objects in the cache are no longer valid, subsequent accesses of these objects can be satisfied quickly since a disk access may not be necessary. Subsequent accesses do require a server access to ensure proper locking semantics, and will require an updated copy of the object if it has been updated on another host.

Transaction abort causes all changes, since start of inner-most transaction, to be undone. An application can abort a transaction through os_transaction methods, macros, or DML statements. ObjectStore will spontaneously abort transactions in response to system failures (e.g., network failure). ObjectStore will also abort transactions if a deadlock is detected. ObjectStore performs deadlock detection as locks are acquired. When a deadlock is detected, ObjectStore aborts one of the affected client processes. Which client process to abort is controlled by a server process parameter. Alternatives for selecting the process to abort include:

- the process whose lock request caused the deadlock,
- the process, from all of those deadlocked, that is the youngest,
- the process, from all of those deadlocked, that has done the least work within its current transaction (measured by server interactions), or
- a process, randomly chosen, from all of those deadlocked.

System aborted transactions that are statically declared (i.e., by macro or DML) are retried some configurable number of times. Functions are provided to determine that a transaction is being retried. Dynamically defined transactions (i.e., instances of os_transaction) must be retried under application control by catching the exception err_deadlock.

5.4.2.2 Relationships & Referential Integrity

ObjectStore allows objects to be inter-related simply by declaring data members which are pointers to other objects. Using this technique, building relationships between objects is no different than manipulating C++ pointers. The advantage of this approach is its simplicity. This pointer-based technique is suitable for modeling uni-directional relationships. Multi-valued uni-directional relationships are formed by using ObjectStore collection classes (i.e., a list of pointers to the related objects). Deletion semantics and referential integrity are not supported across this type of uni-directional relationship.

ObjectStore provides relationship macros to support bi-directional relationships with referential integrity. One-to-one, one-to-many, and many-to-many bi-directional relationships are supported. Forming or breaking a relationship in one direction implicitly causes the inverse relationship to be properly updated. Deleting an object results in all relationships to it being removed. The relationship macros support delete semantics as described in Section 5.4.2.3, Composite Objects. The relationship macros work on both typed and untyped relationships. Typed relationships require a compiler that supports parameterized types.

ObjectStore DML supports definition of bi-directional relationships. Application code which manipulates bi-directional relationships is the same whether those relationships were defined with macros or DML statements.

One consideration when using object pointers and relationships is the database in which a related object resides. ObjectStore databases are stored on a single host. Object Store applications may access multiple databases. Applications wishing to access data that is distributed throughout a network do so by accessing multiple databases. By default, object pointers and relationships may not span databases once the outermost transaction has completed. ObjectStore allows an application to override this restriction, resulting in a slight performance overhead. Note that the code which accesses objects across databases, either via pointer or relationship, is no different from code which accesses objects within a single database. The interface to specify that inter-database references are allowed is a simple one time call for a database or database segment. If a reference is made to an object in a database that is not open, ObjectStore implicitly opens the database for the application.

ObjectStore provides a third mechanism for building relationships between objects. Class os_reference and its specializations allow construction of relationships between objects that are always valid across databases and even across transactions. Both typed and untyped versions of os_reference are provided. Multi-valued relationships may be formed using os_references. Use of os_reference requires more storage than the pointer or relationship approaches already described. A number of os_reference specializations are provided for building relationships from transient memory or locally within a single database. A hidden advantage of using os_reference based relationships is that it reduces virtual memory overhead. Recall that when an object is loaded into memory, any objects that it points to, which are not already in memory, result in reserving the virtual memory locations that they would reside in if loaded. Use of os_reference eliminates the need to reserve virtual memory in this fashion.

5.4.2.3 Composite Objects

ObjectStore supports class attributes (i.e., data members) whose type are another class. This nesting of objects provides a direct mechanism for building composite objects - although the nested objects have no external identity. Delete and lock operations applied to the outer object are also applied 5.4.1 Application Development Issues

ObjectStore supports propagation of delete operations across relationships. The relationship macros described in Section 5.4.2.2, Relationships & Referential Integrity, include parameters for specifying whether related objects should be deleted along with a referencing object. For a pair of inverse relationships, the propagation property can be set for neither relationship, both relationships, or one relationship. There is no mechanism to automatically propagate lock, copy, or equality operations.

See Section 5.4.2.5, Object Versioning, for a discussion of ObjectStore configurations. Configurations provide composite object locking semantics for a group of interrelated objects.

5.4.2.4 Location Transparency

ObjectStore supports the concept of location transparency by allowing the same syntactic constructs to reference objects anywhere in a network of inter-connected databases. ObjectStore supports distributed data access by allowing an application to concurrently access multiple databases. In addition, databases are implicitly opened as objects within a database are referenced.

One consideration, as already described in Section 5.4.2.2, Relationships & Referential Integrity, is that an application must explicitly arrange for object references to be valid across databases. Pointers and relationships across databases are valid if a special flag is set on the database in which the reference is stored. Alternatively, relationships can be formed using ObjectStore os_reference classes, and these are always valid across databases.

5.4.2.5 Object Versioning

ObjectStore provides a versioning facility that is directly tied to the concept of composite objects and work group applications. ObjectStore versioning is based on the concept of configurations and workspaces.

A configuration is one or more objects that are to be treated as a unit for purposes of locking and versioning. ObjectStore provides a class os_configuration for building configurations of objects. An application can allocate a configuration object and associate all objects that are to be part of that unit with the os_configuration instance. A simpler approach is for some object which acts as the parent or root of the unit to inherit from the os_configuration class. This parent object then acts as the configuration, in addition to being an object in the configuration. This approach eliminates the need for explicitly allocated and managed os_configuration objects. Configurations can be nested, thus objects that are configurations can be contained in other configurations. For example, the design of a car might be stored in one configuration. An instance of the class automobile might act as the root of this design unit, and class automobile would inherit from os_configuration. This configuration might contain many subordinate parts, some of which are configurations of their own, such as the design of the engine and the transmission.

Workspaces are ObjectStore objects which control access to configurations. Workspaces form a hierarchy which is rooted at a single global workspace. Configurations stored in the global workspace are available for read access only. Applications can programmatically create workspaces in which configurations can be created and modified. An application can specify a current workspace. A configuration can be checked out of a parent workspace. This check-out places a lock on all objects contained in the configuration and makes a copy of the latest version of the configuration in the application's current workspace. Within the current workspace, objects may be added to the configuration, and objects within the configuration may be modified. Configurations may be checked back into the parent workspace resulting in a new version of the configuration, and all associated objects, to be formed in the parent workspace.

If a single application checks out a configuration, modifies it, and then checks it back in, a linear version of the configuration will be developed. If multiple applications concurrently checkout, modify, and checkin a configuration, branch versions of a configuration will result.

ObjectStore maintains version graphs to denote the linear and branch versioning of objects and configurations. ObjectStore provides mechanisms to control which version of a configuration is visible from a specific workspace. The versioning behaviors ensure that as new configurations are created, inter-object references are correctly updated so that the entire configuration represents a single consistent version.

ObjectStore supports merging of branch versions. Recall that checking out a configuration makes a copy of that configuration in the current workspace. The checkout can also select a second configuration from the parent workspace which is to be associated with the checkout. Upon checkin, the two configuration branches will be merged in the version graph.

ObjectStore's configuration and workspace concepts can be used to implement alternative concurrency control policies. Recall from Section 5.4.2.1, Transaction and Concurrency Model, that ObjectStore's transaction policy is a standard pessimistic single writer/multiple reader policy (concurrent read and write is not allowed). An optimistic policy, in which multiple readers are allowed to proceed concurrently with a single writer can be implemented by having readers access the latest version of a configuration from the

global workspace, and allowing a single writer application to update the configuration in a subordinate workspace. Work group applications are similarly supported since the configuration and workspace concepts allow multiple applications to concurrently update the same configuration, albeit branch versions of that configuration. As already mentioned, ObjectStore provides facilities for programmed merging of the resulting branch versions.

5.4.2.6 Work Group Support

ObjectStore supports work group applications through their versioning mechanism which is implemented with configurations and workspaces. See Section 5.4.2.5, Versioning, for a discussion of these concepts and how they support workgroup applications.

5.4.2.7 Schema Evolution

ObjectStore provides extensive support for performing automatic schema evolution. ObjectStore provides the static member functions evolve (of the class os_schema_evolution) for initiating schema evolution. Schema evolution is performed by invoking the evolve function from within an application whose application schema contains the new schema definitions. Passed to this function is one or more databases, whose schemata are to be evolved to the currently executing application's schema.

ObjectStore's schema evolution is performed in an eager fashion (i.e. all instances are updated upon invocation of schema evolution). ObjectStore's schema evolution capability supports:

- adding/deleting classes,
- changing inheritance relationships,
- adding/deleting data members,
- modifying the type of a data member, and
- rearranging the order of data members.

In addition, instances may be reclassified to be of a subclass of their current class. This is useful as a schema evolves by adding new specialized classes. Schema evolution is not needed for most changes to member functions, the exception being when the only virtual function is added or removed from a class. This requires migration of the class's instances since these instances either now support dynamic method selection or no longer are required to support dynamic method selection and a change of instance storage is required.

ObjectStore's schema evolution facility will initialize new data members to null values. Data members whose types are changed will receive the value of the originally stored data if an appropriate conversion is possible.

ObjectStore supports application-controlled schema evolution by allowing transformation functions to be associated with each class being migrated. These functions have access to both the new and original instance so that new data values can be calculated from original. These transformation functions are free functions, not new member functions of the modified class. As such, direct access to private data members is not possible. ObjectStore schema definition facility (see Section 5.4.2.8, Runtime Schema Access/Definition/Modification) supports access of data member definitions and retrieving/setting the instance data for these definitions.

Many forms of instance migration are performed by creating new instances for the modified class, copying unchanged data members, initializing new data members, and finally deleting the original instances of the class. ObjectStore's inter-object relationship constructs, memory pointers and references,

are all automatically updated in response to these changes. Pointers and references from other databases (that were specified when invoking schema evolution) are also updated as part of the schema evolution process. ObjectStore does not remove relationships, either pointers or references, to (the deleted) instances of deleted classes. Following these relationships at runtime results in ObjectStore generating the exception err_se_deleted_object. (However, removing references to deleted classes is not necessarily required. When a class is deleted from the schema, any relations to that class should also be removed from the schema. Removing a relation from the schema results in all instances which store relationships across that relation being updated. Thus, relationships to instances of deleted classes are removed as a result of making a consistent modification of the schema. By consistent we mean if a class is removed from the schema, all relations to that class should likewise be removed.).

The ObjectStore schema evolution facility identifies queries and indexes that are no longer valid due to data members whose type has changed or data members that have been deleted. Application specific functions can be provided which are invoked upon identification of these invalid queries and indexes. Subsequent use of the queries or indexes results in an exception being raised.

ObjectStore provides the ossevol database utility for performing schema evolutions which do not require application programmed transformation functions or handlers. This is useful for most simply schema evolutions.

5.4.2.8 Runtime Schema Access/Definition/Modification

ObjectStore defines an extensive set of classes which are used to define the runtime description of a database's schema. Instances of these classes are generated during source processing in order to define the class definitions for objects that may be stored within a database. An overview of the classes provided for this purpose is:

- os_schema, os_comp_schema, os_database_schema, os_app_schema - an abstract schema representation class, and specilizations for representing compilation, application, and database schemas. Class os_schema_evolution provides an interface to the schema evolution capabilities of ObjectStore.
- os_class_type - represents a C++ class whose instances are to be stored in a database.
- os_base_class - represents the superclass of some class, and the form of inheritance (e.g., private, public, etc.)
- os_member, os_member_type, os_member_variable, os_member_function - classes for representing the data members and their types, and the member functions of a class.
- os_type, os_enum_type, os_enumerator_literal, os_integral_type, os_pointer_type, os_real_type, os_void_type, os_function_type - classes which represent the standard C++ types.
- os_template, os_template_instantiation, os_template_formal_arg, os_template_actual_arg - classes for defining type or function templates and their instantiations.

The classes which model a database's schema are available to an application. Using these classes, a custom object browser could be built.

5.5 Evaluation of GemStone Version 3.2

Information presented in this section was derived from the technical documentation set for GemStone Version 3.2 and GeODE (the GemStone Object Development Environment) Version 2.0. This evaluation is based on both the C++ application interface to GemStone and on GeODE. GeODE is a multi-user application development and execution environment for GemStone OODBMS applications.

A major discriminator between GemStone and the other OODBMS products described in this report is that GemStone is an active database while the others are passive databases (see Section 3.1.2.9, Active vs. Passive Data Model). ITASCA, an OODBMS developed by Itasca Systems, Inc., is another OODBMS that supports an active data model. The ITASCA OODBMS was not evaluated as part of this critical review and technology assessment of OODBMS.

This section provides a detailed evaluation of application development issues and selected advanced topics.

5.5.1 Application Development Issues

Servio Corp. provides two very different avenues for building GemStone applications:

- o standard programming language development in C, C++, or Smalltalk, and
- o visual programming, supported by textual program descriptions where necessary, in GeODE.

Standard programming language development under GemStone is similar to that for the other OODBMS products described in this report. For example, GemStone is provided with a C++ interface which allows passive OODBMS applications to be developed under the following model:

- o C++ class definitions are used to describe the database schema. GemStone provides a number of predefined classes which can be incorporated into an application (e.g., classes defining standard data structures and the ability to form relationships between objects).
- o Classes are identified as persistent by inheriting, either directly or indirectly, from class GS_Object which defines persistent behavior. User defined classes may have instance variables whose types are standard C++ data types, other user or GemStone defined classes, or one dimensional arrays of these. GemStone currently does not support the concept of multiple inheritance.
- o A GemStone utility, the *registrar*, parses user defined classes and builds a database schema for these classes. The *registrar* also generates additional methods for these classes which allow instances to be mapped from the database to the application's address space and vice versa.
- o In addition to standard class specifications, the *registrar* recognizes special keywords, placed in comments, for describing additional properties of a class. Keywords, for example, can be used to describe the class of object that may be stored in a new set or array class, or may specify a query, whose implementation is to be automatically generated by GemStone.
- o Within an application, persistent objects are referenced through special pointers, each of which is a non-persistent object. Definitions of the classes which represent these pointers are generated automatically by the registrar and results in a class hierarchy for object pointers paralleling the application's class hierarchy.
- o As objects are accessed by the application, they are moved from the database to the application's address space. Any modification to the state of an object is recognized by GemStone and causes the object to be updated in the database (assuming the current transaction eventually commits). Alternatively, for efficiency reasons, an application can manually control when object updates are reflected in the database.

- Standard C++ compilers and debuggers (see Section 4.5) can be used to develop and debug the methods for new classes and any additional functions needed in the application.

Application development in GeODE is very different from the standard text based approach. GeODE provides a comprehensive development environment in which much of an application's functionality can be specified graphically. An application developer may use text-based programming for highly complex or unique activities or for performance critical sections of code. GeODE provides a complete application design, implementation, and debugging environment especially tailored to building object-oriented database applications on top of GemStone. This section, on Application Development under GemStone, will focus on building applications using GeODE.

Constructing active GemStone database applications is supported through Servio Corp.'s Smalltalk DB programming language and development environment. Smalltalk DB is a Smalltalk-like language that acts as both a DDL and DML for GemStone database applications. Behaviors implemented in Smalltalk DB can be executed in the server processes (i.e. those which access the database files) instead of the client process. Applications constructed with GeODE use Smalltalk DB for implementation of object behaviors, thus they can be characterized as active database applications. An application, created in any of the programming languages supported by Servio, may access the active data objects stored in a GemStone database. These applications need not link to Smalltalk DB, the language used for defining methods which represent the active object behaviors. These applications simply interface to a GemStone database in their native language to invoke an object behavior. This initial invocation has the potential to access the active behaviors of any object in the database.

5.5.1.1 Developer's View of Persistence

Application development with GeODE largely hides the details of object persistence from the developers. Using GeODE programming tools, an application is developed by designing and implementing classes, visual programs, forms, and methods. Behavioral programming, done either with the visual programming tools or the Smalltalk DB programming environment, is purely based on sending messages to objects. When sending a message to an object, there is no need for the developer to consider where it is loaded in memory or even whether it is loaded in memory. GemStone is an active database, meaning that objects truly contain their behaviors and can execute those behaviors. Thus, methods can be executed in database server process without explicitly copying all accessed objects to a client process. This is significant in applications which access large numbers of objects (e.g., queries over all instances of a class). With GemStone, there is no need to copy each accessed object from the server, over the network, to the client, prior to invoking methods on those objects.

Persistence under GemStone is independent of type. New persistent objects are created by sending messages to objects representing a class, requesting that a new object be created. Objects that are created by an application are automatically made persistent, once the transaction they were created in is committed. (Smalltalk DB programming supports the concept of transient, or non-persistent, objects. These transient objects can be made persistent by creating a reference to the transient object from a persistent object.) An interconnected network of objects is formed by creating new objects and assigning them to the instance variables of the current object (i.e., the object which received a message and is currently executing one of its methods).

Complex objects are formed by defining classes whose instance variables are combinations of:

- named variables - refer to other objects (e.g., an employee object might store its salary and its department). These variables can be thought of as pointers to other objects,

- index variables - arrays containing objects that are referenced by a number (i.e., an index), and
- anonymous variables - used in classes such as bags and sets, and are accessible by association instead of direct reference.

Instance variables may be typed or untyped. An instance variable is typed by assigning a constraint to it which identifies the class of object which may be assigned to that instance variable.

A unique aspect of GemStone is that an object will persist as long as there is some other object holding a reference to it. Objects cannot be explicitly deleted by an application. This helps to assure the referential integrity of the database (see Section 5.5.2.2, Relationships & Referential Integrity).

GemStone supports clustering of objects on disk. An object is clustered by sending it a message with an indication of where it is to be clustered. Thus, an object may be clustered at any point in its lifetime, not only when it is created. A method is also provided to cluster an object and all of its instance variables (i.e., other interconnected objects), and to cluster a heterogeneous collection of objects.

GemStone does not implicitly acquire object locks. All locking must be done explicitly by the application (see Section 5.5.2.1, Transaction and Concurrency Model).

5.5.1.2 Application Development Process

GeODE, the GemStone Object Development Environment, provides a multi-developer environment in which all class definitions (i.e., schema), behaviors (i.e., code), and instances (i.e., application data) are represented as objects and managed by the GemStone OODBMS. A GeODE application consists of:

- a database schema, representing the information manipulated by the application,
- the data being manipulated by the application, and
- forms allowing the user to interact with the application in order to create, view, and modify the data.

GeODE provides the following tools for developing applications:

- Schema designer - a graphical definition of database schemas (i.e., class definitions).
- Application designer - controls contents and visibility of an application, such as what schema it uses, what data objects are accessible to the application, what users can access the application.
- Forms designer - a graphical user interface building tool that allows complete Motif-based X Windows applications to be built without any textual programming.
- Visual program designer - a graphical tool allowing application behaviors to be described visually, eliminating the need for textual programming for most application tasks.
- System programming tools - a complete development environment for textual programming of applications. These tools include class, hierarchy, and method browsers, database administration tools, file access tools, etc. Textual programming in GeODE is done in Smalltalk DB, a Smalltalk dialect developed by Servio Corp. for building and executing GemStone applications.

All tools are interactive, graphical tools especially designed for building software based on reusable object structures. GeODE provides a complete user authorization mechanism to prevent accidental access of data by unauthorized users. The GeODE tools support debugging of applications as they are incrementally built, such as individual forms or behaviors implemented by visual program fragments.

Application development under GeODE proceeds by designing the database schema using the Schema Designer, creating one or more application development folders to be used by one or more application developers, designing and implementing custom forms to allow users to display and interact with the data, and finally implementing program behaviors either visually or through the system programming tools. Once an application is complete it is released to users by making the application globally visible and providing access authorization to the application for the users.

Additional information on each of GeODE's tools is provided in Section 5.5.1.3, Application Development Tools.

5.5.1.3 Application Development Tools

5.5.1.3.1 Database Administration Tools

Operating characteristics of the GemStone database can be controlled through system level configuration files and user configuration files (for a specific application). Configuration parameters are available for:

- concurrency control policy,
- what file extents make up the database, are they replicated, and how they grow,
- control over cache page sizes, and
- control over the process performing storage compaction.

GeODE provides DBATool, an interactive, graphical database administration tool for easily performing most common database administration tasks. Topaz, a GemStone interface, provides access to all GemStone administration tasks. Topaz users must be familiar with Smalltalk DB (a Smalltalk dialect developed by Servio Corp. for building and executing GemStone applications.) Database administration tasks may also be performed programmatically. GemStone represents the entire database system as objects, each with specific authorization access. Users with access to system level objects can invoke methods which perform database administration tasks on-line.

GemStone database administration capabilities include:

- User Administration - all GemStone users are represented in the database with specific information regarding their access to the GemStone database (user id and password). Database administration activities include setting up new user accounts, specifying user privileges, and placing users in authorization groups (which control group access to GemStone objects).
- Segment Administration - GemStone defines the concept of segments to specify users' access to groups of objects. Every object is associated with a segment which identifies access privileges to the objects. Access rights can be individually controlled for the owner of the segment, to a fixed number of specific user groups, and to the entire user population.
- Database File Storage Administration - GemStone databases are stored in one or more files (called *extents* in GemStone). Extents grow as objects are allocated and only shrink as the result of specific database administration tasks which shrink extents. Extents can be added at system startup or dynamically as the GemStone database grows. Database administration tasks control the location, number, size and usage of extents. Extents may also be replicated to recover from media failures (keeps on-line shadow image of an extent).
- Network administration tasks control where the multiple processes making up a GemStone database and a set of applications execute and how they communicate over a local area network.

- o Database backup recovery tasks - both full and incremental backup utilities are provided with GemStone and may be run on-line. Recovery can be performed from backups or, if available, from replicates of the database extents.

5.5.1.3.2 Database Design Tools

The GeODE Schema Designer is an interactive graphical schema browsing and specification tool. The Schema Designer allows application designers to define database schema (i.e., classes) in a visual manner. The schema is defined as a set of interrelated classes, where the relationships are displayed as arrows between boxes representing the classes. Each class has relationships defined to its superclass, subclasses, and instance variables. All of these relationships may be shown visually, or may be hidden to reduce clutter on a diagram. The entire schema of a database is represented by one or more pages of class definitions - called class graphs. Classes defined in detail on one class graph are referenced, and shown in an abstract form on other class graphs.

In addition to the graphic representation of the database schema, the Schema Designer provides Class Description Dialogs for specifying additional information about a class (e.g., varying part of a class, class storage formats, allowing/disallowing subclasses, etc.). The Class Description Dialog can also be used to quickly define instance attributes and their properties (i.e., define the class of instance that can be stored in an instance variable and whether it represents a multi-valued reference, which is really a reference to a set or an array of objects).

The Schema Designer makes it easy to see all the classes defined within the Database (at least those accessible to the application being developed). Classes predefined in GemStone and available for use in the application are automatically included in all Schema Designer sessions.

Classes may also be defined from within the Systems Programming Tools. Existing class definitions may be copied as the starting point for defining a new class, or a class may be defined from scratch. Class definitions may be copied from a class browser, and once copied, modifications of the class may be made (e.g., add an instance variable). Classes may be defined from scratch by invoking a special dialog which allows the application designer to specify:

- o the name of the class,
- o the class's superclass,
- o instance variables of the class, and the type of object referenced by the instance variable, and
- o other parameters of the class, such as whether this class can have subclasses, whether the class is modifiable, whether default collection classes should be defined to hold instances of this class, etc.

5.5.1.3.3 Source Processing Tools

GeODE provides many tools for translating visual program descriptions into a format suitable for application execution. For example, the Schema Designer provides menu commands to apply a schema description to the database. The result of this operation is that the database is loaded with a description of the classes and they are available for use in an application. Many of the system level browsing tools allow class behaviors to be edited and menu options provide a means to compile the new version of the method. These and many other less visible capabilities of GeODE are what makes it a complete program development environment. The remainder of this section briefly describes two of the main tools for building visual applications, the Forms Designer and the Visual Program Designer.

The Forms Designer is a graphical tool for defining forms in which a user interacts with a GemStone application. The Forms Designer is similar to GUI builder tools presently on the market. It provides a menu of widgets which an application designer places on the screen to provide a particular presentation of the information. A GeODE application developer defines forms interactively by laying out their format and connecting objects, components, or methods to individual form fields, defining responses to user actions on the forms. Forms have behaviors associated with them that specify how a user interacts with the form and what actions are to occur in response to those interactions. Form application designers specify the actions that occur in response to events occurring within the form.

Forms are combinations of buttons, labels, menus, lists, tables, text, scroll bars, and other nested forms. Forms can include regions for display of graphic information and for recording and playback of sound. Forms can display groups of related information in tree formats. Form definition allows users to control placement, color, shading, and visibility of an entity. Form designers can specify that fields must have data entered in them by the user or that they may not be modified.

Actions that may be invoked from forms include:

- execution of visual programs, which are user defined program sequences,
- accessor/updater methods to retrieve information from the database into a form or to store information to the database from a form,
- method invocation, meaning messages may be sent to objects related to the form to perform some action on that object, and
- validation and display formatting, meaning behaviors can be triggered to check the validity of user inputs or to transform information in the database into a format most suitable for display within the form.

The Visual Program Designer provides to GeODE application developers a graphical specification tool for describing the behaviors of forms. Visual programs are used for loading information into forms, performing computations on that information, and for storing information from a form to the database. The Visual Program Designer provides a graphic palette and display surface for constructing visual programs. The palette contains visual programming blocks that may be combined to form a visual program. GeODE provides numerous predefined visual blocks for performing operations including:

- arithmetic operations,
- communication between forms,
- execution control flow,
- invocation of methods in objects, of responses from other forms or parts of forms, or even operating system functions,
- string handling,
- transaction control,
- user interaction, and
- form operations (e.g., open, close, etc.).

Application developers create visual programs by linking visual blocks. Each visual block has input and output pins associated with it that control how it may be linked with other visual blocks. A visual block fires, calculating values for it's output pins based on its input pins whenever data is available on those

input pins. Visual blocks can also request values from its input pin, causing blocks preceding it to either fire or request input for itself.

Visual programs are normally triggered as the result of events occurring within a form. These may be user actions or system caused actions such as form initialization.

Visual programs can be saved, copied and reused. Segments of visual programs can be packaged and treated as a single visual block, thus providing a technique for extending the capabilities of the visual programming language.

Both forms and visual programs are supported by GeODE debugging facilities, see Section 5.5.1.3.5, Debugging Support.

5.5.1.3.4 Database Browsing Tools

GeODE provides many tools for browsing information in GemStone databases. (The Schema Designer, Forms Designer, and Visual Program Designer all provide mechanisms to browse program structures designed by the respective tools.) Specific browsers supplied in the Systems Programming Tools with GeODE are described in this section.

The System Browser provides a view into all the classes accessible to an application developer. The System Browser is a multi-paned browser allowing the user to scroll through classes and methods. Selecting a particular class results in its methods being displayed. Selecting a particular method results in the source for that method being display in a text pane. (Note: actually GeODE organizes all classes into class categories and methods into method categories a la Smalltalk.) The System Browser allows a user to easily refine his search by selecting a class category, then a class, then a method category, then a method. This is expected to be quite helpful for large application development efforts.

The Class Browser is similar to the System Browser, except that it is opened for a particular class. Method categories and specific methods can be selected and the resulting code is displayed in a text pane. The class browser can display instance variables.

The text panes of the System and Class Browsers allow Smalltalk DB code to be edited and compiled (as methods of the currently selected class). Debugging information, such as listing current breakpoints may be viewed. Breakpoints may be set or removed within these panes. Finally, selected source code may be executed as an Smalltalk DB expression.

The Method Browser is spawned from one of the other browsers to provide detailed information on a particular method. The method browser displays the list of implementing class and method pairs for methods which meet some user specified criteria (e.g., show all the implementations of some selected method). The method browser's text pane provides the edit, compile, and execute capabilities already described for the System and Class Browsers. The method browser provides search and query facilities which supports the user in understanding which methods are inherited or reimplemented and from where messages are sent.

The Hierarchy Browser provides a graphical description of the class hierarchy currently available in the GemStone database. A user can expand and collapse segments of the hierarchy as desired. Browsers may be opened on classes selected in the Hierarchy Browser.

The File Browser provides an interface to the host operating system's file system. The multi-paned browser allows the user to select a directory whose contents are to be viewed in an associated pane, and

then to select a specific file whose contents are to be displayed in a third pane. Again, displayed code can be edited, compiled, and debugged.

An Inspector Tool is provided to easily traverse the objects in the database. This tool allows the user to view the contents of an object and traverse to view the contents of related objects.

Both the System and the Class Browser can be used to show and set lock status of objects. This is useful to determine if any lock conflicts will result when attempting to modify class definitions, forms, or behaviors that might be shared with other developers. An auto locking feature can be invoked to automatically lock all objects accessed by the browsers to ensure that any changes will not be lost due to system generated transaction aborts caused by lock conflicts detected at transaction commit time.

5.5.1.3.5 Debugging Support

GeODE is a complete programming environment and thus includes debugging capabilities. As described in the previous section, many of the GeODE browsers allow source expressions to be evaluated in place, and support the display and manipulation of debugging controls (e.g., breakpoints). Three types of explicit debugging are supported by GeODE:

- visual program debugging,
- form debugging, and
- Smalltalk DB code debugging.

The Visual Program Designer supports interactive debugging of visual programs. Visual programs can be executed in a mode that graphically highlights the movement of data through the program. The program can be run in single step mode and the user can inspect data values at any point in the program. Special windows may be attached to any point in the visual program to display the current value of the data passing through the block.

The Forms Designer allow forms to be run for purposes of testing and debugging. In order to run a form, the user first associates some data with it, normally one or more objects that are to be displayed or manipulated by the form. Running a form causes the visual programs associated with that form to be executed. Errors encountered in these programs result in the Visual Program Designer being opened with the error being highlighted.

Smalltalk DB debugging is supported by a typical screen-oriented debugger providing a view of the current call stack and source code. The debugger supports inspection and modification of variables and objects. Execution can be controlled by single stepping or through the use of breakpoints. The System and Class Browsers also support debugging of Smalltalk DB code (see Section 5.5.1.3.4, Database Browsing Tools).

5.5.1.3.6 Performance Tuning Tools

Operating characteristics of the GemStone database can be controlled through system level configuration files and user configuration files, see Section 5.5.1.3.1, Database Administration Tools.

The Database Administration Tools provide access to process statistics for the processes which access the GemStone database. Statistics available include:

- elapsed real time,
- elapsed CPU time, and
- the number of I/O operations performed for each process.

The database administrator is also provided with guidance on how to potentially shrink the size of a database.

Advice on when not to use visual programming is provided in the documentation set. Visual programming is not suggested for:

- functions which require highly optimized performance,
- large or complex behaviors that might be easier to understand textually due to the large amount of space needed to represent the visual program, or
- behaviors with many unique processing requirements which would require specification of many new visual programming blocks.

5.5.1.4 Class Library

GeODE provides an extensive class library for building X Windows / Motif applications on top of the GemStone OODBMS. All of GeODE is in fact implemented as GemStone data objects and tools provided in the GeODE environment can be incorporated into user applications if desired by making use of the provided class definitions. Classes provided with GeODE include:

- Form and field classes - provide the ability to create custom user dialogs based on X and Motif.
- Dialog classes - provide for user interaction with an application.
- Visual block classes - providing building blocks for GeODE visual programs.
- Collection classes - providing common data structures such as array, set, and dictionary, and means for iterating over a collection.
- Folder classes - provide high-level application support such as an application's folder, icon, execution context, etc.
- Tool classes - define the tools provided in the GeODE programming environment (e.g., Forms Designer, File Browser, Debugger).
- Schema classes - supports the implementation of, and information represented by, the GeODE Schema Designer.
- Low level kernel classes - provide access to the GemStone database and user interface systems.
- X Windows classes - provide an interface to X Windows, the Xt Intrinsics layer and the Motif widgets.

5.5.2 Advanced Topics

Except where noted, topics described in this section are common to the GemStone database, regardless of the interface through which an application accesses the database (e.g., GeODE, C++).

5.5.2.1 Transaction and Concurrency Model

GemStone supports both optimistic and pessimistic concurrency control policies. An optimistic policy is enforced by default. Applications can implement a pessimistic policy by explicitly acquiring and releasing object locks. GemStone applications usually execute from within a transaction. A transaction-less state is also provided for data browsing purposes. When a user logs into a database, a local workspace is created and a transaction is begun. Each transaction commit or abort implicitly begins another transaction. After each transaction commit or abort, the latest state of objects in the database are available to the application.

Under GemStone's optimistic concurrency control policy, each client application freely accesses objects in its local workspace. GemStone maintains lists of objects read and written during the transaction, called read and write sets respectively. At transaction commit time, conflict detection algorithms compare this transaction's operations with that of all other transactions that have committed during the lifetime of this transaction. Conflict detection protocols are applied as follows:

- write/write conflicts - checks for objects in write sets of both this and another transaction, and
- read/write conflicts - checks for cases where an object in this transaction's write set is in another transaction's read set, and that the other transaction has an object in its write set that is in this transaction's read set.

Notice that the read/write conflict checking is truly an optimistic policy. A transaction may read objects that have been concurrently updated by another transaction, and use that information in the process of updating other objects in the database. Configuration parameters are available to eliminate the read/write checking in cases where the application uses locks to prevent out of date data from being stored in the database.

Pessimistic concurrency control is implemented by an application explicitly acquiring locks on objects that it intends to update or base updates upon. GemStone provides three types of locks:

- Read lock - prevents other applications from acquiring write or exclusive locks; prevents other transactions from committing if they have written the object.
- Write lock - prevents other applications from acquiring any form of lock; prevents other transactions from committing if they have written the object; allows the object to be read optimistically.
- Exclusive lock - prevents other applications from acquiring any form of lock; prevents other transactions from committing if they have read or written the object.

Read locks are the typical shared access lock, allowing multiple applications to read the data. Write locks ensure that only the holding transaction can update the data, but does not prevent others from reading the object and using it in an optimistic transaction which makes changes to other, potentially related, objects. Exclusive locks prevent other transactions from performing this sort of optimistic update.

Locks are held across transactions. When a transaction commits or aborts, all locks are retained. Locks must be explicitly released by the application. (Methods are provided to drop all locks on transaction boundaries). Lock requests that cannot be granted result in error conditions being raised in the requesting application. GemStone does not provide any mechanism to wait for a lock to become available. Such behaviors must be explicitly programmed by the application. Since GemStone does not allow processes to wait for locks, deadlocks cannot occur, and thus deadlock detection algorithms are not required.

When a transaction successfully commits, all changes made by the transaction are updated in the database and are available to other applications. Similarly, all updates made concurrently by other applications are now visible to this application. If a transaction aborts, no changes are made to the database and the application's local workspace remains intact. By keeping the failed transaction's workspace intact, modifications that cannot be committed may be saved in some other fashion (e.g., create new objects, save external to GemStone, etc.). Subsequent activities can then attempt to redo the aborted work. Transaction aborts caused due to optimistic locking can be avoided by acquiring explicit locks on objects that could be potential causes for conflicts.

Somewhat related to concurrency control is the idea of security and authorization. GemStone defines the concept of segment for grouping all objects which have a common authorization profile. Segments are independent of the concurrency control mechanisms and the location of an object in the database. Each object is assigned to a segment. A segment is like a tag that defines what users have access to the object. A segment defines read and write privileges for the owner of the segment, for groups of users, and for all users. A user may access an object only if he is authorized through the segment to which the object is assigned. Segments are essential in a multi-user development environment. For example, only one or a small number of users should have write access to database schema objects. Many applications can also benefit from segments, where different end users are restricted from seeing some portions of the database.

5.5.2.2 Relationships & Referential Integrity

GemStone supports uni-directional, one-to-one relationships. Relationships in GemStone are specified by including a class as the type definition for an object's data member (i.e., instance variable). Relationships are formed and removed by assigning values to the data members. Objects related across a relationship are accessed by sending messages to it. GemStone ensures that the target of the method send is loaded into memory as part of this operation.

Bi-directional relationships are not automatically supported in GemStone. Bi-directional relationships must be explicitly maintained as two separate uni-directional relationships. GemStone provides data abstraction classes such as Sets and Arrays which can be used to form one-to-many relationships.

GemStone supports the concept of referential integrity by not allowing objects to be deleted. In fact, there is no mechanism to explicitly delete an object in GemStone. An object is removed from a GemStone database when there are no longer any references to it. Under this model, it is impossible to traverse a relationship that references an object which has been deleted. For this reason, referential integrity is always guaranteed.

5.5.2.3 Composite Objects

GemStone does not support the concept of composite objects whereby a set of interrelated objects are locked, copied, or manipulated as a whole. (A collection of heterogeneous objects may be locked, copied, or clustered as a unit however).

GemStone supports locking a collection of objects. Collections (e.g., bags, lists) are used to group objects in some application specific manner. A lock may be acquired on each object that is present in a collection. Locking a large group of objects can be performed by first forming a collection of the objects and then locking the collection.

5.5.2.4 Location Transparency

GemStone supports the concept of location transparency across a distributed set of extents comprising a single database. Object access is not dependent upon its location in the network of interconnected databases. There is no syntactic difference in the code to access an object based upon its location.

GemStone databases are composed of multiple files called extents. GemStone supports the ability to move objects to different extents within a database. GemStone supports replicated extents for purposes of recovery or improved access performance. A replicate is an on-line copy of a database extent. Updates made to the objects in the extent are automatically propagated in the replicates of the extent. Thus, to improve retrieval performance, replicates could be placed local to machines which typically query information stored in the associated extent.

5.5.2.5 Object Versioning

GemStone Release 3.2 provides no support for object versioning. GemStone does support versioning of schemas (see Section 5.5.2.7, Schema Evolution).

5.5.2.6 Work Group Support

Work Group applications are partially supported under GemStone through the explicit locking mechanism described in Section 5.5.2.1, Transaction and Concurrency Model. Since GemStone lacks an object versioning capability, work group applications which allow concurrent access and update of a common set of objects is not possible. Using the GemStone locking facility allows an application to assure itself prolonged exclusive access to a set of objects (since locks are not released at the end of each transaction).

GeODE is an example of a work group application built on GemStone. GeODE is a multi-user development environment for building GemStone applications. GeODE users explicitly lock objects (e.g., class definitions, source code, etc.) to prevent other users from concurrently modifying those same objects. The security and authorization mechanisms supported by GemStone aid the ability to develop multi-user database applications.

5.5.2.7 Schema Evolution

GemStone supports various levels of automatic schema evolution depending upon the programming interface being used for application development and the type and form of changes made to the classes. GemStone always allows changes to the number of methods, their names, and parameter profiles. Changes of these forms do not require adjustment to the memory representation of any instance objects.

Within the C++ development environment, the registrar utility (which is responsible for parsing C++ header files into a database schema) can be configured to detect changes to existing classes and either allow or disallow them. Changes to existing classes result in any existing instances of those classes no longer being accessible. Programmatic migration is necessary in order to recover data in existing instances when changes are made to the C++ header definitions.

Within the GeODE Schema Designer, classes are interactively designed as an application is constructed. All classes are tagged as being modifiable or invariant. Classes that are modifiable may have instance variables added, deleted, or modified, but may not have instances created. Prior to creating instances for a class, it must be changed from modifiable to invariant. Invariant classes can have instances created for them but, for the most part, they can not have changes made to their instance variables. An exception to this rule is that if an invariant class has no subclasses and is not indexable (i.e., stores a varying number of non-named instance variables which are accessed via an index, similar to an array), then instance variables may still be added. Under this scenario, existing instances are updated in a lazy evaluation mode, where the instance's storage is updated to reflect the new class definition only when the new instance variable for the instance is accessed. GemStone can support this process because it provides the concept of schema versioning. (All Smalltalk DB classes are supported by this form of schema evolution).

Schema versioning is essential in a multi-user application development environment. Schema versioning allows multiple versions of the schema to exist simultaneously. Multiple schema versions allow different application designers to proceed in parallel with different versions of the schema, updating to the new schema when most convenient for their work. Additionally, an application may be released to a group of users with a public version of the schema while application developers make enhancements based on new versions of the schema.

Schema versioning provides the most complete form of automatic schema evolution. Within the Schema Designer and Application Designer tools, commands are provided for updating all or a subset of a class's instances to reflect a later version of the schema. This capability modifies instance storage as needed to reflect new or deleted instances variables. All forms of schema modification are supported under this model (e.g., adding and deleting instance variables, changing the type of an instance variable, modifying superclass relationships). The schema migration activity proceeds as follows:

- for each existing instance, creates a new instance based on the latest version of the schema,
- copy current values for any instance variables that have not been changed, and finally,
- switch the identity of the new and original instance, so that references to the old instance are now to the new instance.

When this automatic migration is complete, all of the original instances will no longer be referenced by any object in the database. As a result of no longer being referenced, they will be deleted as part of GemStone's garbage collection process.

5.5.2.8 Runtime Schema Access/Definition/Modification

GemStone can be characterized as a pure object-oriented database. As such, all of GemStone is represented as objects and can be manipulated by sending messages. GemStone provides access to and modification of the database schema at runtime.

GeODE is an example of an application which allows users to perform dynamic schema definition. Recall GeODE is a program development environment, and a major part of that is defining database schemas. The GeODE programming environment is built as a set of GemStone classes, and those classes are available to others building GemStone applications. Applications wishing to perform dynamic schema definition need not write tools to present and modify the schema. Segments of GeODE may be incorporated into an application to provide the ability to view, define, and modify the schema. Since GemStone supports complete instance migration through the use of schema versions, applications need not dispose of all existing instances when changes are made to the database schema.

GemStone allows applications to modify the behavior of the classes which implement the database, but suggest this not be done. Instead, GemStone classes should be subclassed (i.e., specialized) in order to extend or modify their behavior. Thus, for example, an application which requires class behaviors to include constraints (in addition to standard methods) can be provided through subclassing and adding constraint representations and firings to the GemStone definition of a class.

GemStone provides a complete set of classes which define and implement the database schema (called metaclasses). The classes named Class and Behavior represent the ability to create and modify classes. GemStone applications (e.g., GeODE) create and modify class definitions by sending messages to instances of Class and Behavior. For example, a new class is defined by sending a subclass creation message to the superclass of the new class. The class named Behavior defines the ability to query for a class's superclasses and instance variables. Behavior provides methods for defining new methods to access the instance variables of a class or even to define and compile Smalltalk DB methods (the source for the method is provided as a string). These behaviors are useful for advanced applications that allow users to define new classes and behaviors dynamically.

5.6 Evaluation of Object Design ObjectStore PSE Pro

Information presented in this section was derived from the technical documentation set for ObjectStore PSE Pro Release 2.0 for Java. All evaluations are based on the Java application interface to ObjectStore. This section provides a detailed evaluation of application development issues and selected advanced topics.

5.6.1 Application Development Issues

5.6.1.1 Developer's View of Persistence

ObjectStore provides persistence on an object basis, through an enhanced form of Java object serialization. Serialization provides a simple yet extensible mechanism for storing objects persistently. The Java object type and safety properties are maintained in the serialized form and serialization only requires per class implementation for special customization. Objects are created as transient objects, and are promoted to persistent objects if they are added to an existing persistent object (a root object can be created). The Java Transient keyword can override persistence.

Root objects provide entry points for ObjectStore databases. Generally, only a small number of root objects exist, with other persistent objects accessed by reference.

5.6.1.2 Application Development Process

Developing applications utilizing ObjectStore persistence involves building class files from Java source through standard Java compilers, followed by post-processing to generate stubs for ObjectStore serialization. Java source code is created through standard OO Design techniques that decorate persistent classes. The classes generated from compiling the Java source code are used by a post processor, which may replace specific class files with a version modified to implement the ObjectStore persistence mechanism. If multiple persistent classes exist, the post processor can be run in a batch mode, which specifies all the persistent related classes. Zip and jar files can also be used as input to aid in the batch processing.

If a class references a persistent class but is not persistent itself, it can be post processed as a persistent-aware class, which requires less overhead. The post processor 'annotates' all persistent classes and their super-classes, providing implementations of an ObjectStore interface class responsible for persistence. Initialization code is also added for transient fields within persistent classes.

5.6.1.3 Application Development Tools

5.6.1.3.1 Database Administration Tools

Because the persistence mechanism is based on serialization, there are no dedicated processes that require administration or monitoring. A 'Session' within a Java virtual machine has lock access to a database, and an API with many management functions is available to that session. While there are no specific tools, there are provisions for schema evolution which involve dumping and reloading serialized data.

5.6.1.3.2 Database Design Tools

Because Java classes are defined as persistent through the post processing tool, any design tool which will generate Java source is compatible. There are no specific database design tools included with ObjectStore PSE Pro for Java.

5.6.1.3.3 Source Processing Tools

The process of annotating a Java class as persistent is executed through a post-processing tool. This tool provides implementations of a persistence interface for classes and their super-classes.

5.6.1.3.4 Database Browsing Tools

Object Design provides a database-browsing tool, which allows the user to view the schema and contents of one or more databases. This tool will provide the following information:

- Name of the database
- Name and number of each type of object in the database
- Total size in bytes occupied on the disk by each type of object
- Number of destroyed objects

Additional information can be retrieved from the command line:

- Oid, which is an internal representation of its location in the database
- Type
- Number of bytes it occupies on the disk
- If it is an array, the number of elements in the array

This information can also be retrieved through the ObjectStore PSE API.

5.6.1.3.5 Debugging Support

Object Design does not supply a source level Debugger; moreover, third-party debuggers will not correctly provide source level debugging of persistent classes, due to the ObjectStore annotations. A utility is provided to validate references in a database and provide various consistency checks. A scan of the database is made to determine if references to deleted objects exist. The tool provides the ability to search for dangling references and change invalid references to null values.

5.6.1.3.6 Performance Tuning Tools

A Persistence Garbage Collection tool is included to collect unreferenced Java objects within the ObjectStore PSE database. This tool uses a two-phase approach; first marking unreferenced objects, then removing these marked objects. There is no provision for forming contiguous blocks from the freed up space. This functionality is available through the ObjectStore PSE API.

5.6.1.4 Class Library

ObjectStore PSE includes an extensive API built upon a static class library. The API supports:

- Session management
- Thread management
- Synchronization of threads within sessions.
- Database management for creating, opening, closing, copying, garbage collecting, schema evolving, destroying and querying a database.
- Transaction management, such as initiating, promoting or demoting and exiting.
- Root object manipulation
- Collections such as bag, map, tree and list.
- Reference checking and versioning.
- ODMG-compliant query interface

5.6.2 Advanced Topics

5.6.2.1 Transaction and Concurrency Model

ObjectStore provides a transaction implementation that supports a traditional multiple readers/single writer concurrency control policy for the access of individual objects. ObjectStore PSE supports four types of transactions: update, read-only, update non-blocking and read-only non-blocking.

If the update mode is selected, the application is permitted to modify data within the current transaction's open database. A write lock is requested, and if not used by another session under the same Java VM, granted. The transaction request blocks until a write lock is granted. The read-only transaction allows the application to read but not modify data in the open database for the current transaction. If there are no sessions within the same Java VM owning or waiting for an update lock, a shared read-lock is granted; otherwise, the application blocks until a read-lock is available. An update non-blocking transaction is similar to the update transaction, without the blocking. If a write lock is not immediately available the transaction operation exceptions out. The read-only non-blocking transaction is similar to the read-only transaction, without the blocking. If a read lock is not immediately available the transaction operation exceptions out.

ObjectStore PSE Pro can support multiple sessions within the same Java VM process. Each session can access the same database through a read-only transaction, and simultaneously engage separate update transactions against different databases. Only one session at a time can open an update transaction against a particular database.

5.6.2.2 Relationships and Referential Integrity

ObjectStore allows objects to be inter-related through the use of references to other objects. While one-directional relationships can be formed this way, multi-valued unidirectional relationships can utilize the ObjectStore collection classes. One-to-one, one-to-many and many-to-many bi-directional relationships are supported under Java, and references can be checked after deletion through the referential check tool.

5.6.2.3 Composite Objects

ObjectStore supports class attributes whose type are another class, thus supporting the building of composite objects. Objects referred to by a persistent object are themselves persistent, unless the reference was keyworded as transient. Delete and lock operations apply to these persisted through reference objects.

5.6.2.4 Location Transparency

Location Transparency, or the lack of a visible object location, can be achieved through ObjectStore's External References. An External Reference represents a reference to a persistent object. The contents of an External Reference can be encoded into a text string for transport within applications across a network, then decoded back to an External Reference. The persistent Java object can be extracted from these External References, providing persistent object access across transaction boundaries.

5.6.2.5 Object Versioning

ObjectStore PSE Pro utilizes Java serialization for versioning, which includes support for utilities that generate a unique ID value for a class based on its signatures. Incorporating a static final integer set to a value for a specific version of the class ensures that future versions of the class can load that serialized version.

5.6.2.6 Work Group Support

ObjectStore PSE Pro has no provisions for workgroup support.

5.6.2.7 Schema Evolution

ObjectStore has no direct support for schema evolution; however, sample code utilizing Java's serialization for versioning can be used. A tool is supplied with the Java JDK which generates a unique integer ID for a class, called the version ID. This version ID is stored within a public constant inherited from the Java serializable classes. The schema evolution involves calculating the version ID and dumping the data for the original class, modifying the class persistent fields, then update the public constant with the calculated previous version ID.

5.6.2.8 Runtime Schema Access/Definition/Modification

The database contents and object types can be retrieved during runtime either through an inspection utility, or through API method calls. Information available includes:

- Name of the database
- Name and number of each type of object in the database
- Total size in bytes occupied on the disk by each type of object
- Number of destroyed objects

An application can not modify the application schema during runtime.

5.7 IBM San Francisco

Information presented in this section was derived from the technical documentation set IBM San Francisco. This section provides a detailed evaluation of application development issues and selected advanced topics.

5.7.1 Application Development Issues

5.7.1.1 Developer's View of Persistence

San Francisco provides the framework for class persistence, along with an abstract Entity class that all persistent objects must inherit from. Persistent objects are only accessible within the scope of a transaction; however, copies of persistent objects can exist across transactional boundaries. San Francisco transactional persistence includes implementations for flat file serialized persistence and drivers and mapping tools for RDBMS storage. A persistent object is associated with a specific server storage mechanism, which actually contains the object. These containers can either be defined as a relational database (RDB) container, or the portable operating system interface for computer environments (POSIX) container. The POSIX container is generally used in a testing environment, or when a small number of objects are to be used. Objects stored within a POSIX container can not benefit from the recoverability and data integrity offered by RDB containers. The persistence of an object within the San Francisco framework, along with the location of the object and its container, is transparent to the application developer. The persistent object is created through an object factory, which resides on the same server as the storage mechanism that contains the object. An object factory can create both persistent and transient objects, as well as delete persistent objects from the storage mechanism. San Francisco provides methods for creating transient copies of persistent objects, along with methods for promoting transient copies to persistent objects.

Referential persistence is supported through the concept of ownership. An object owned by a persistent object is by default persisted; however, objects referred to by a persistent object are not persisted when the referring object is persisted.

5.7.1.2 Application Development Process

IBM San Francisco provides a complete roadmap for application development including requirements mapping, analysis and design. Initial requirements collection results in application scenarios, which are mapped to existing San Francisco framework scenarios to determine the requirement coverage by the framework, including the extensions of the framework and the changes to the framework. The analysis phase of the application roadmap includes defining any required user interfaces or business logic. Application scenarios or use cases are detailed, possibly uncovering additional scenarios. Related scenarios are used to develop class categories (high level groups of classes). These class categories are compared to the existing San Francisco framework, identifying required extensions to the San Francisco class model. Software patterns, used extensively within San Francisco, are then identified.

A design model is created from a combination of the analysis model and knowledge of the San Francisco technical architecture. This model is built using the Rational Rose modeling tool, which includes helper wizards for annotating specific San Francisco classes or business domain logic. A key step in the design model is mapping legacy systems such as databases to the model. San Francisco provides a schema-mapping tool to aid in the process of mapping RDB tables and attributes to application objects. Once a design model is complete and annotated with San Francisco-specific class information, a code generator is used to create the persistent application business objects. The application user interface and additional domain specific logic is added to the generated code.

5.7.1.3 Application Development Tools

5.7.1.3.1 Database Administration Tools

San Francisco provides limited support for data base evolution through dynamically extensible classes, along with a schema-mapping tool. Most of the administrative tools provided by San Francisco involve configuring the object containers and their San Francisco interface, rather then the administration of a specific database that may implement a particular container. The Server Management Configuration Controller configures servers within the San Francisco network. This tool allows an administrator to define the location and configurations of server objects, such as Factory objects, Transaction Service objects and Problem Service objects. The San Francisco Configuration Utility allows an administrator to change the persistence type of a specific container, create new classes residing within a container, move classes from one container to another, generate new containers, or control transaction processing within a container. The Conflict Control Administration Utility allows the administrator to define conflicts and default resolutions within the distributed environment. The Save/Restore Manager is a backup utility for all objects associated from the SF root object that are based on the San Francisco programming model.

5.7.1.3.2 Database Design Tools

The Default Schema Mapper (DSM) tool uses Java Reflection to generate a new relational database table from application class files. The DSM maps single classes' attributes to default columns within a single table. An automatically generated ID column becomes the primary key column for the new table, and a partition key column, which is used to identify which San Francisco objects map to this specific 'EntityOwningExtent' San Francisco object. Th Extended Schema Mapper (ESM) provides object mapping into new as well as existing relational database tables. With ESM, advanced functions, such as sub-classing, joins, specific data-type mapping can be applied.

5.7.1.3.3 Source Processing Tools

San Francisco provides source code generation through the Rational Rose modeling tool and the SF Code Generator. The SF Code Generator tool uses specifications in the Rose model to generate the Java code according to the programming model. The San Francisco Code Generator implements a large portion of the San Francisco server-programming model; the developer adds the client-programming model and business logic. San Francisco provides tools for developing Java Bean Components or SF classes. The SF Wizards for Borland JBuilder can be used to construct a new application solution, common business object, or industry-domain framework that builds on the San Francisco Entity, Dependent, or Command classes from within the Borland JBuilder IDE. The SF Java Bean Wizard can be used to create Java Beans that work within the San Francisco framework.

5.7.1.3.4 Database Browsing Tools

The Extended Schema Mapper provides the user with database meta (schema) information; however, there is no provision for examining the content of an RDB container. The physical location of all containers and the classes within the containers can be examined and modified through the SF Configuration tool.

5.7.1.3.5 Debugging Support

Distributed applications which place restrictions on which actions can be executed concurrently can utilize the Conflict Control Manager, which can monitor and prevent defined conflicts from occurring. Problems encountered by San Francisco applications can be logged through the Problem Manager.

5.7.1.3.6 Performance Tuning Tools

The SF Configuration Manager tool provides the user with the location, type and content of all SF containers. Grouping on the same server SF commands with the objects they utilize reduces remote activity and improves overall performance. The Configuration Manager can be used to relocate classes to different containers, relocate containers to different servers, and change the underlying persistence mechanism of a container.

Because an object query (performing a query through method calls on an object) can incur considerable performance degradation, San Francisco provides a query pushdown mechanism on certain query implementations. The query pushdown avoids the hydration of each object in the queried collection by mapping the query directly into the underlying container. For relational databases, a select SQL statement is generated to match the query.

5.7.1.4 Class Library

IBM San Francisco provides an application framework for core business processes, providing greater reuse within a specific application domain then a traditional class library. This application framework provides the developer with an abstraction of their business processes, with concrete classes for common business objects and extension points where a developer could assert competitive advantages. The framework is separated into three layers, the Core Business Processes Layer, the Common Business Objects Layer and the Foundation Layer.

The Core Business Process Layer provides default business logic for selected vertical domains in the form of frameworks. These frameworks include extension points for competitive discriminator business logic. This layer currently includes ‘towers’ for General Ledger, Warehouse Management and Order Management.

The Common Business Objects Layer is intended to provide cross-application support for recognized standard business objects. These CBOs are extensible allowing for the inclusion of enterprise wide domain logic; however, concrete representations of these objects are available. An example of some of these business objects include:

Business Partner

- Address
- Calendar
- Currencies

Providing the infrastructure for distributed multi-user multi-platform applications is the Foundation Layer, serving as a consistent interface for application programmers and framework developers. The Foundation layer can be categorized into four functional areas, Base object model classes, Concrete business object classes, Factories and Utilities. The Base object model classes define the basic structure for San Francisco objects and business application objects, providing services such as persistence. Examples of these include:

- Command
- Entity and subclasses
- Dependent and subclasses

The Concrete business object classes are instantiable implementations of basic business object components such as:

- Collection
- Iterator
- Dtime
- DDecimal.

Factory classes, based on the Abstract Factory pattern and Factory Class Replacement pattern, are used to create all persistent, distributed San Francisco objects. Services provided by the factory objects include:

- Naming
- Concurrency
- Transactions
- Distribution
- Persistence
- Object life cycle services (creation, deletion, and copying)

The Utility classes, generally intended to be used as concrete classes rather then extended, include:

- Security administration
- Conflict control
- Session management
- Batch management

5.7.2 Advanced Topics

5.7.2.1 Transaction and Concurrency Model

The San Francisco transaction model supports two-phase commit protocol and mixed-phase commit protocol. The transaction model is based on a presumed abort protocol that allows the changes to be aborted when the system failure occurs before ending the transaction. The transaction model also supports a runtime combination of objects from both one-phase resource containers and the two-phase resource containers in the same transaction. The mixed-phase commit protocol allows the application to have recoverable resources and non-recoverable resources participate in the same transaction.

Persistent objects can only be accessed within the scope of a transaction, unless a NO_LOCK copy of the object is requested. Only NO_LOCK copies of objects provide non-transactional access; however, changes to NO_LOCK copies are not reflected to the original object. Transactions are managed through a set of global factory methods. These methods include:

- begin: Used to initiate a transaction; required before accessing any persistent object.
- commit: Terminates current transaction. Traditional commit protocol is executed to either update all the objects and their persistent storage or to invoke recovery procedures. Nested begin() calls are not supported.
- rollback: Terminate the current transaction, resetting the state of all objects accessed within the transaction and removing their locks. This is used for transaction errors from where there is no recovery. Objects created within the transaction are not persisted, and released through Java garbage collection.
- rollbackOnly: Mark the current transaction as requiring a rollback, but allows the transaction to continue until the commit. When San Francisco detects a transaction failure, an exception is thrown and this method is called, allowing the programmer to attempt to perform some state saving recovery.
- suspend: Suspend the current transaction thread, allowing the processing of another thread. San Francisco assigns a work area to each San Francisco derived thread. It is this work area which is saved off.
- resume: Resume control of a suspended San Francisco thread by restoring the saved work area. A resume operation cannot occur within a running transaction.
- getTransactionStatus: Returns the status of the current transaction.

San Francisco defines three policies to the modification of business objects, the Pessimistic, the NO_LOCK and the Optimistic approach. Using the Pessimistic approach, a read lock is used to retrieve object attributes via accessor methods. Typically, a server located command object retrieves the necessary attributes. The application then works with the attributes indirectly, modifying transient or state variables. When the application wishes to write the attributes, a write lock is requested, and a validation of the modified data takes place before the new values are written and the write lock removed. This validation must check for any modifications to the object between the initial read lock and the current write lock. The modified state values are compared with the original values to determine what has changed. The disadvantage of not working with the object directly is overcome with the NO_LOCK approach. Retrieving an object with an NO_LOCK access mode returns a copy of the object, which can be used directly by the application. If the original object undergoes modification, the NO_LOCK copy can refresh

these changes at the container level (each reference to the NO_LOCK object must also be refreshed). The write operation proceeds in a manner similar to the Pessimistic access, by merging changes in the NO_LOCK copy with the write copy. While this access method provides the application with the actual object, it incurs the overhead of the object copy. The Optimistic approach also provides the application with a copy of the actual object to work on; however, it is expensive if the object access is collision prone. Similar to the NO_LOCK, the Optimistic approach retrieves a container-refreshable copy of the object. While the NO_LOCK copy is used across transactional boundaries, the Optimistic copy is only accessed within the read transaction. When access is complete, the transaction is committed and the transaction server determines if there were any other updates to the object. Any outside changes to the object will cause the transaction commit to fail, usually resulting in the application retrying the operation using the Pessimistic approach. If there were no conflicts, the commit updates the object persistently.

5.7.2.2 Relationships and Referential Integrity

San Francisco business objects support two different types of containment relationships, containment by ownership and containment by reference. Objects contained by ownership will be deleted when their container parent is deleted (maintaining referential integrity), while objects contained by reference will not be deleted in this case. San Francisco supports transient primitive representation called a Dependant that may only be owned by other objects. Multi-valued unidirectional relationships are supported through the use of containers. Objects, which are mapped to relational databases normally, would not update their state to the database until the commit operation, possibly permitting integrity conflicts. Referential integrity can be maintained through a sync method within the SF base object, forcing the object's state to be written to the underlying store before commit.

5.7.2.3 Composite Objects

The IBM San Francisco framework relies heavily on software design patterns and software re-use techniques. Generalization through object composition is a powerful reuse technique, and is supported by several design patterns within San Francisco. The Extensible Item pattern allows dynamic modification of an entity's interface, adding or removing methods driven by some application level policy change. The "Ables/Ings" pattern is a generic builder pattern that allows for the dynamic composition of complex objects through steps, where the next step is de-coupled from the result of the previous step. The SF Foundation layer provides a concrete implementation of a composite object, which allows the addition of other objects. San Francisco also supports dynamic extensible objects, which provide for runtime evolution of the object model.

5.7.2.4 Location Transparency

All San Francisco business objects can have application location transparency, as the underlying distributed framework of the Foundation layer manages the copying or remote access of objects. San Francisco objects may reside strictly on the server maintaining their container and accessed transparently through the use of a proxy stub on the client, or be serialized and transported to the client and accessed directly.

5.7.2.5 Object Versioning

San Francisco currently does not support versioning of classes, but provides wrappers for future versioning support. This does permit future versions of San Francisco to update the data streams written with the current version if the data change follows certain upward compatibility rules. The following changes to a class are considered upward compatible:

- The addition of a class or interface
- The addition of an attribute to a class
- The addition of a method on a class or interface
- The addition of a class or interface into an existing class hierarchy
- The removal of a private attribute from a class
- The removal of a private method from a class or interface

To aid in the recognition of previous versions of an object, each instance of a BusinessObject (Entity, Dependent, or Command) must correspond to a version number. There are three options for scoping this version number, depending on the distribution of the product or application. The version number can be scoped to an application, to a package, or to a class. Using San Francisco streaming methods requires versioning per class.

5.7.2.6 Work Group Support

San Francisco only supports workgroups through transaction locks on objects, ensuring that only one application within a sequence may modify the object at a time. The Common Business Objects and the framework itself are mechanisms that support the development of workgroup applications.

5.7.2.7 Schema Evolution

An enterprise business model can by dynamically modified through the use of San Francisco's extensible entity patterns and composite object patterns. Because the framework applications are built to an interface and not an inherited object, changes in the data model can be shielded from quiescent applications using the model. San Francisco supports a limited form of relational database schema evolution for situations where an object's underlying datastore is a relational database. The Extended Schema Mapper can be used to re-map application business object attributes to table attributes.

5.7.2.8 Runtime Schema Access/Definition/Modification

The extensible entity provides a runtime ability to modify the enterprise object model. San Francisco provides a Policy pattern (similar to GOFs Strategy pattern) which is based on the concept of encapsulated algorithms separated from the objects which use them. Policies contained and referenced by name within a policy container, where they can be added or removed by controller objects. Applications designed to utilize this mechanism could support runtime algorithmic replacement without changing their base object model.

6. Conclusions

Modus Operandi has evaluated seven commercial OODBMS products as part of this State of the Art Report for the Data and Analysis Center for Software (DACS). Due to the short duration of this contracted effort, we have based our evaluation on a detailed review of the technical documentation set provided by the OODBMS vendors. An evaluation based on technical documentation will be more substantive than that available from marketing literature, but not as comprehensive as one based upon actual use of the OODBMS products.

In addition to being an evaluation of the current state of OODBMS products, this report was written with two specific purposes. The first was to enumerate and describe the many features that are expected of an object-oriented database. This is accomplished in Section 3, Evaluation Criteria. This information should be useful to those readers who are new to object-oriented databases, and even to those new to object-oriented technologies and/or database technologies. A second objective was to provide readers a starting point for performing more in-depth OODBMS product evaluations. Appendix A, Evaluation Survey Template, provides a detailed listing of questions and issues that a thorough evaluation should address. It is hoped that these two sections prove useful to others in future OODBMS product evaluations. Chapter 4 provides summary information about each product, focusing on target platforms, interface languages, development tools and environments compatible with the OODBMS. Chapter 5 provides an in-depth analysis of application development issues and advanced object-oriented issues for each OODBMS evaluated as part of this effort. The remainder of this section draws some overall conclusions from this evaluation effort and gives guidance to those who will be performing OODBMS product evaluations.

Overall conclusions of this evaluation effort are:

- The evaluated products tend to be second generation systems. New versions of the products typically appear in less than the one year time frame, each fixing problems and adding functionality.
- The products can be considered true databases. They provide support for distributed access to data, multiple users, transaction support, recovery and backup facilities. Actual experimentation with the products is needed to understand the performance impacts of these database features and how well the databases scale when storing large amounts of data distributed across a network.
- The products typically have complex application interfaces, often offering several alternatives for performing the same task. System and application parameters make the range of implementation choices even greater. Building complex applications that perform and scale well will require significant expertise and understanding in the operational capability of the database. This expertise will take time to acquire and early application development efforts must plan accordingly. Vendor support, in the areas of training, phone assistance, and direct consultation may be essential during initial development efforts.
- The products differ in their support for some of the more advanced object-oriented capabilities. All OODBMS support the typical object-oriented features such as classes, attributes, methods, dynamic binding, encapsulation, etc. Products differ in areas such as the ability to model relationships between objects, provision of object versioning, work group support, etc.

- Schema evolution is an area of concern when developing applications on top of an OODBMS. Schema evolution is the ability to bring objects up to date after changes have been made to the class definition of the objects. Schema evolution is helpful during development, when the schema definitions are changing rapidly but the only existing objects represent test data. Schema evolution is essential when fielded applications are upgraded, either to fix errors or enhance functionality, and user databases must be maintained. The evaluated products differ widely in their support for schema migration. Some products offer no support, some support the application developers in doing programmed schema modifications, others support limited or complete automatic schema migration.
- OODBMS products can be distinguished as being active or passive databases. Passive databases provide object storage and sharing capabilities leaving all application processing to be performed in client applications. Passive databases may store interface descriptions of the methods defined for the classes in the schema, but they do not store the implementation of those methods and can not execute queries or object updates in the database processes. Thus in a passive database, all objects that need to be operated on must be moved from the database to the application process. Active databases store the implementation of a class's methods in the database and have the ability to execute those behaviors in database processes. Active databases can thus perform significant computations and associative retrievals without moving the accessed data to the application's process. In addition, an active database will typically provide one or more procedural interfaces allowing non object-oriented programs access to objects and behaviors stored.
- Each vendor has a unique application interface and set of classes upon which applications are built. Porting an application from one vendor's OODBMS to another will require significant rework. All of the major OODBMS vendors have formed a working group whose charter is to define a single application interface to an OODBMS, thus promoting portability of applications. A draft of this interface is due in the fall of 1993. All participating vendors have agreed to implement the interface once defined. Applications built to a vendor's custom interface will most likely require significant modification when the standard interface is defined.
- The set of application tools available for use with the evaluated OODBMS vary greatly. All are suitable with one or more third party software development environments. Some provide vendor specific compilers and development environments. Using vendor specific tools has the advantage of a better integration between database and application, and thus improved performance is to be expected. On the other hand, using vendor supplied tools requires a change from a user's current development environment.
- The set of database administration facilities provided by each of the evaluated OODBMS also vary widely. For example, the ability to relocate entire databases and/or specific objects is an important facility not supported by all vendors.

Some general guidelines for selecting an OODBMS for a particular application are as follows:

- Identify preferred development language, expected target platforms, network requirements, and the need for heterogeneous operation. Products differ widely here, and there are no workarounds.

- Identify high level application requirements in the areas of versioning, work group support, and schema evolution. As already stated, products vary greatly in these areas. Programmed workarounds for missing functionality are possible but will add significant complexity to the application development effort.
- Evaluate the candidate products based on the technical documentation, not the marketing literature. Appendix A of this report provides a template that can be used to structure this evaluation.
- Build a benchmark which closely mirrors the expected application. Model the number of concurrent users, the expected distribution of data through the network, the typical data access patterns, etc., as closely as possible.
- In addition to benchmarking, experiment with the database to understand the architecture of the system. Information sought is an understanding of how data is moved from database to server processes and to applications process, how process memory requirements grow, how disk space requirements grow, the effect of object delete on disk and memory requirements, buffering of object updates, the overhead associated with locking, transaction control, and journaling, etc.
- Consider the quality of the vendor's documentation, support staff, and training classes. The evaluation, benchmarking, and experimentation tasks that should be performed as part of the product selection effort will afford product evaluators ample opportunity to consider these issues.

About the Authors

Gregory McFarland is Director of Enterprise Application Solutions at Modus Operandi. While at MO, Mr. McFarland's work has focused on object management technologies. Mr. McFarland is currently involved in the integration of an OODBMS with commercial and custom tools as part of Modus Operandi's Advanced Systems Engineering Automation (ASEA) contract funded through U.S. Air Force Rome Laboratory. Mr. McFarland was responsible for the design and implementation of an object representation and constraint system as part of Modus Operandi's InSight project, a meta-modeling toolkit used to automate engineering methods. Mr. McFarland was also responsible for Modus Operandi's Classic-Ada product, an object-oriented pre-processor for Ada. Mr. McFarland was the lead engineer in building the persistent object management system for Classic-Ada.

Prior to joining Modus Operandi in 1989, Mr. McFarland was a Systems Designer at Grumman, Data Systems Division. Mr. McFarland was a lead engineer on Grumman's Fault Tolerant Project and, as part of this project, participated in the development of a distributed programming testbed to be used in the study of software fault tolerance and distributed systems construction. Mr. McFarland was responsible for developing an Ada code generator, a distributed Ada runtime, and tools for distributing Ada programs.

Mr. McFarland received B.S. and M.S. degrees in Computer Science from the State University of New York at Stony Brook, in 1982 and 1983 respectively.

Dr. Andy Rudmik is Executive Vice President and Chief Technology Officer at Modus Operandi. He has been involved with object-oriented database technologies since 1983. In 1983, Dr. Rudmik was the lead engineer on the Distributed Software Engineering Control Process, which, to our knowledge, is the first environment framework using an object-oriented data management system. He has also been a reviewer and consultant on the European ESPRIT project interface specifications for the Portable Common Tool Environment (PCTE). Dr. Rudmik was a member of the KIT/KITIA, a Government and Industry team tasked to define a standard interface set for the software engineering environments.

More recently, Dr. Rudmik was the Chairman of the CASE Integration Services committee, which he led to become ANSI standards body X3H6, with a charter of defining object-oriented interfaces for software engineering environments. He is an active participant in the Object-Oriented Database standards committee X3H7.

Dr. Rudmik has developed several object-oriented repository technologies that have been used in compilation systems, software engineering environments, and programming systems.

Dr. Rudmik received B.A.Sc., M.A.Sc, and Ph.D. from the University of Toronto in 1970, 1972, 1976 respectively.

David Lange is a Lead Engineer at Modus Operandi. Mr. Lange has leveraged his 12 years of software and OO experience in developing persistent object based solutions for such projects as the Advanced Engineering Requirements Workstation, which aids the developer in performing the software engineering process, and Scenario Generation, which provided tools and methodologies for the elicitation of requirements through scenarios. He has investigated integrating large legacy databases with object-oriented databases to support the CLASS project, a robust future-proof repository of information supporting Distance Learning and Electronic Classrooms. Mr. Lange has developed multi-tier Java applications requiring a variety of persistence schemes, including object-oriented, direct relational and relational mapped databases.

Mr. Lange received a B.S. degree in Computer Engineering and a M.S. degree in Microelectronics from the University of Central Florida in 1988 and 1990 respectively.

References

[Atk92] Atkinson, Malcolm, et al., "The Object-Oriented Database System Manifesto." *Building An Object-Oriented Database System*, California: Morgan Kaufmann, 1992.

[Ban87] Banerjee, Jay, et al., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases." *Proceedings of the ACM SIGMOD Conference* , (1987).

[Ban92] Bancilhorn, Francois, et al., *Building An Object-Oriented Database System.* California: Morgan Kaufmann, 1992.

[Bar93] Barry, Douglas K., "ODBMS Feature Checklist." *Object Magazine*, (January-February 1993).

[Bor81] Borning, A., "The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory." *ACM Transactions on Programming Languages and Systems*, Vol 3, (October 1981) pg. 353.

[Bro84] Brodie M.L., et al., *On Conceptual Modelling*. New York: Springer-Verlag, 1984.

[Bro91] Brown, Alan W., *Object-Oriented Databases - Applications in Software Engineering*. London: McGraw-Hill, 1991.

[Buc86] Buchman, A.P., et al., "A Generalized Constraint and Exception Handler for an Object-Oriented CAD-DBMS." *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, (September 1986) pg. 38.

[Bun79] Buneman, O.P. and E.K. Clemons, "Efficiently Monitoring Relational Databases." *Transactions on Database Systems,* Vol. 4, (September 1979) pg. 368.

[But92] Butterworth, Paul, "ODBMSS As Database Managers." *Journal of Object-Oriented Programming Focus On ODBMS*, (1992) pg. 3-12.

[Car93] Carey, Michael J., et al., *The 007 Benchmark*. Computer Sciences Department, University of Wisconsin-Madison, 1993.

[Cat91] Cattell, R.G.G., *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Massachusetts: Addison-Wesley, 1991.

[Cel92] Cellary, Wojciech and Genevieve Jomier, "Consistency of Versions in Object-Oriented Databases." *Building An Object-Oriented Database System*, California: Morgan Kaufmann 1992.

[Coa90] Coad, Peter and Edward Yourdon, *Object-Oriented Analysis*. New Jersey: Prentice-Hall 1990.

[Col92] Colley, Grant., "Versioning." *Object Magazine,* (November-December, 1992) pg. 32.

[Dat86] Date, C. J., *An Introduction to Database Systems*. Massachusetts: Addison-Wesley Vol. I, 4th ed. 1986.

[DeW92] DeWitt, David J., et al., "Three Alternative Workstation-Server Architectures." *Building An Object-Oriented Database System,* Morgan Kaufmann, San Mateo, CA, 1992.

[Gra76] Gray, J.N., et al., "Granularity of Locks and Degrees of Consistency in a Shared Data Base." *Proceedings IFIP TC-2 Working Conference on Modelling in Data Base Management Systems*, (January 1976).

[Hud86] Hudson, Scott E. and Roger King, "CACTIS: A Database System for Specifying Functionally-Defined Data." *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, (September 1986) pg. 26.

[Kim90] Kim, Won, *Introduction to Object-Oriented Databases.* Massachusetts: The MIT Press, 1990.

[Kim92] Kim, Won, "Architectural Issues in Object-Oriented Databases." *Journal of Object-Oriented Programming Focus On ODBMS*, (1992) pg. 13.

[Kun81] Kung, H.T. and J.T. Robinson, "On Optimistic Methods for Concurrency Control." *ACM TODS,* Vol 6, No. 2, (June 1981).

[Lai91] Lai, K.W. Larry and Leon Guzenda, "How To Benchmark An OODBMS." *Journal of Object-Oriented Programming,* Vol 4, No. 4, (July-August 1991).

[Lec92] Lecluse, C., et al., "O2, an Object-Oriented Data Model." *Building An Object-Oriented Database System*, California: Morgan Kaufmann, 1992.

[Loc79] Lockman P., et al., "Data Abstraction for Database Systems." *ACM Transactions on Database Systems*, Vol 4, No. 1, (March 1979).

[Loo92a] Loomis Mary E.S., "Object Versioning." *Journal of Object-Oriented Programming*, (January 1992) pg. 40.

[Loo92b] Loomis, Mary E.S., "ODBMS vs. Relational." *Journal of Object-Oriented Programming Focus On ODBMS,* (1992) pg. 35.

[Loo92c] Loomis, Mary E.S., "Integrating Objects with Relational Technology." *Journal of Object-Oriented Programming Focus On ODBMS*, (1992) pg. 39.

[Loo92d] Loomis, Mary E.S., "Objects and SQL: Accessing Relational Databases." *Journal of Object-Oriented Programming Focus On ODBMS*, (1992) pg. 51.

[Loo92e] Loomis Mary E.S., "Object Database - Integrator For PCTE." *Journal of Object-Oriented Programming*, (May 1992) pg. 53.

[Loo93a] Loomis Mary E.S., "Object And Relational Technologies: Can They Cooperate?" *Object Magazine*, (January-February 1993) pg. 35.

[Loo93b] Loomis Mary E.S., "Object Programming and Database Management." *Journal of Object-Oriented Programming*, (May 1993) pg. 31.

[Mai86b] Maier, David and Jacob Stein, "Indexing in an Object-Oriented DBMS." *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems,* (September 1986) pg. 171.

[McF93] McFarland, Greg and Rudmik, Andres, "Object-Oriented Database Management Systems," *DACS Technical Report,* September, 1993.

[Mey88] Meyer, Bertrand, *Object-Oriented Software Construction.* New York: Prentice Hall, 1988.

[Mos85] Moss, J. Eliot B., *Nested Transactions*. Massachusetts: The MIT Press, 1985.

[Par89] Parsaye, Kamran, et al., *Intelligent Databases.* New York: Wiley, 1989.

[Ras92] Rasmus, Daniel W., "Relating To Objects." *Byte*, (December 1992) pg. 161.

[Rot92] Rotzell, Katie and Mary E.S. Loomis, "Benchmarking an ODBMS." *Journal of Object-Oriented Programming Focus On ODBMS*, (1992) pg. 99.

[Rud93] Rudmik, Andres and Sharon Rohde, "InSight - An Object-Oriented Modeling Tool." *Proceedings of the Third Annual Symposium of the National Council on Systems Engineering,* (July 1993) pg. 285.

[Sar92] Sarkar, Manojit and Steven P. Reiss, *A Data Model for Object-Oriented Databases*. Department of Computer Science, Brown Univeristy, Providence RI, CS-92-56, December 1992.

[Ska86] Skarra, Andrea H., et al., "An Object Server for an Object-Oriented Database System." *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, (September 1986) pg. 196.

[Sol90] Soley, Richard Mark (ed.) *Object Management Architecture Guide.* Object Management Group, Vol. 2.0, September 1992.

[Ste92] Stein, Jacob, "Evaluating Object Database Management Systems." *Journal of Object-Oriented Programming,* (October 1992) pg. 71.

[Sto90] Stomebraker, M., et al., "Third-Generation Database System Manifesto." *ACM SIGMOD Record*, (September 1990) pg. 31.

[Tha86] Thatte, Satish, "PERSISTENT MEMORY: A Storage Architecture for Object-Oriented Database Systems." *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, (September 1986) pg. 148.

[Thu92a] Thuraisingham, M.B., "Security in Object-Oriented Database Systems." *Journal of Object-Oriented Programming Focus On ODBMS*, (1992) pg. 81.

[Thu92b] Thuraisingham, M.B., "Multilevel Secure Object-Oriented Data Model - Issues On Noncomposite Objects, Composite Objects, And Versioning." *Journal of Object-Oriented Programming Focus On ODBMS*, (1992) pg. 121.

[Vas93] Vasan, Robin, "Relational Databases And Objects: A Hybrid Solution." *Object Magazine,* (January-February 1993) pg. 41.

[Ver92] Versant Object Technology. *How to Evaluate Object Database Management Systems,* 1992.

[Wir90] Wirfs-Brock, Rebecca, et al., *Designing Object-Oriented Software*. New Jersey: Prentice-Hall, 1990.

[Zdo86] Zdonik, Stanley B. and Peter Wegner, "Language and Methodology for Object-Oriented Database Environments." *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences,* (1986) pg. 378.

Bibliography

Ahmed, Shamim, et al., *A Comparison of Object-Oriented Database Management Systems for Engineering Applications.* Massachusetts Institute of Technology, Research Report R91-12, 1991.

Atwood, Tom, "An Introduction to Object-Oriented Database Management Systems." *Journal of Object-Oriented Programming Focus On ODBMS,* (1992) pg. 1.

Blakely, Adrian, "So, What Makes Object Databases Different?" *Hotline on Object-Oriented Technology*, Vol. 2, No. 4, (February 1991) pg. 6.

Blakely, Adrian, "Data Definition Languages." *Hotline on Object-Oriented Technology,* Vol. 2, No. 5, (March 1991) pg. 6.

Blakely, Adrian, "So, What Makes Object Databases Different? (Part 3)" *Hotline on Object-Oriented Technology,* Vol. 2, No. 6, (April 1991) pg. 5.

Bonte, Eugene A., "Object Databases And Object Request Brokers: Foundation For Distributed Object Management." *Journal of Object-Oriented Programming Focus On ODBMS,* (1992) pg. 61.

Crosse, Glenn, "Selecting An Object Database System." *Object Magazine,* (July-August 1991) pg. 19.

Kanellakis, Paris, et al., "Introduction to the Data Model." *Building An Object-Oriented Database System,* California: Morgan Kaufmann, 1992.

Kim, Won, "Object-Oriented Database Systems: Strengths And Weaknesses." *Journal of Object-Oriented Programming Focus On ODBMS*, (1992) pg. 27.

Loomis, Mary E.S., "Database Transactions." *Journal of Object-Oriented Programming Focus On ODBMS,* (1992) pg. 115.

Maier, David, "Why Object-Oriented Databases Can Succeed Where Others Have Failed." *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems,* (September 1986) pg. 227.

Marrs, Kieth, *Object-Oriented Database Management Systems: The State Of The Art*. McDonnell Douglas Corporation, Report Number MDC B1659, June 1989.

Maryanski, Fred et al., "The Data Model Compiler: A Tool for Generating Object-Oriented Database Systems." *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems,* (September 1986) pg. 73.

Schwarz, P. et al., "Extensibility in the Starburst Database System." *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems,* (September 1986) pg. 85.

Vasan, Robin, "Integrating Objects And Relational Databases." *Object Magazine*, (July-August 1992) pg. 59.

Zaniolo, Carlo, "Object-Oriented Database Systems and Knowledge Systems." *Proceedings from the First International Workshop On Expert Database Systems,* (1986) pg. 50-65.

Zicaro, Roberto, "A Framework for Schema Updates in an Object-Oriented Database System." *Building An Object-Oriented Database System,* California: Morgan Kaufmann, 1992.

Appendix A – Evaluation Survey Template

This appendix provides a template that may be used for evaluating an OODBMS. The contents of this template were derived from the Evaluation Criteria presented in Chapter 3 of this report. The template consists of two types of entries:

- o statements regarding functional capabilities of an OODBMS, presented in normal font, and,
- o *implementation approaches, application development issues, and vendor issues, presented in italics.*

The statements regarding functional capabilities are typically easily evaluated: either an OODBMS provides the listed functional capability or it does not. Notice that some of the listed capabilities are variants, each providing a more sophisticated level of functionality. An OODBMS may support one or more of these alternatives.

The implementation approaches, application development issues, and vendor issues are subjective evaluation criteria. Each requires substantial investigation to understand the characteristics of the OODBMS in regard to the specific evaluation topic. Implementation issues will, of course, be considered proprietary information by the vendor. These issues deal with how certain database functionality is implemented, which can be used to estimate performance characteristics. Application development issues deal with the process by which applications are designed, developed, tested, debugged, and maintained. Tool support and integration with other tools is considered under this category. Vendor issues evaluate vendor maturity, support capabilities, etc.

The following evaluation survey template is numbered to match the evaluation criteria section numbering of Chapter 3.

A.1 Functionality

A.1.1 Basic Object-Oriented Modeling

A.1.1.1Complex Objects

Supports definition of complex objects which can be used to describe real-world entities.

A.1.1.2Object Identity

Provides system generated object identifiers which are independent of object contents and transparent to the user.

Object identifiers are not reused.

Object identifiers are not based upon the object's location within the database.

Describe the size and format and object identifiers.

Describe how object identifiers are used to identify a specific object.

A.1.1.3 Classes

Classes are used to define the database schema.

Classes define the external view of an object.

Class extents are supported.

A.1.1.4 Attributes

Provides instance attributes.

Provides class attributes.

Supports basic attribute types.

Supports fixed length arrays as attribute types.

Supports variable length arrays as attribute types.

Supports structures as attribute types.

Supports non-persistent classes as attribute types.

Supports binary large objects (BLOBS) as attribute types.

Supports derived attributes.

A.1.1.5 Behaviors

Supports behavior definition.

Supports behavior invocation.

A.1.1.6 Encapsulation

Provides a mechanism to encapsulate attributes.

A.1.1.7 Inheritance

Supports single inheritance.

Supports multiple inheritance.

Provides a means to control name conflict resolution.

A.1.1.8 Overriding Behaviors and Late Binding

Allows a subclass to override the behavior of a superclass.

Supports late binding of behavior invocation.

A.1.1.9 Persistence

Supports persistence of objects.

Persistence is a characteristic of a class (i.e., all objects of a class persist).

Persistent is a characteristic of an object (i.e., persistence is independent of class).

Persistent is a characteristic of being related to some other persistent object.

Objects may be explicitly deleted.

Objects are implicitly deleted when no more references to the object exist.

A.1.1.10 Naming

Supports object access by name.

An object may be known by multiple names.

A single database wide name scope is provided.

Application defined name scopes are supported.

A.1.2 Advanced Object-Oriented Database Topics

A.1.2.1 Relationships & Referential Integrity

Supports uni-directional relationships.

Supports bi-directional relationships.

Supports one-to-one cardinality.

Supports one-to-many cardinality.

Supports many-to-many cardinality.

Supports list semantics (ordered, duplicates allowed).

Supports set semantics (no order, no duplicates).

Supports bag semantics (no order, duplicates allowed).

Bi-directional relationships ensure referential integrity.

Referential integrity is assured for all relationship forms via tombstones.

Referential integrity is assured by only deleting objects when all references to that object have been removed.

Describe the implementation of relationships (e.g.,as attributes, objects, or hidden data structures).

Describe the process by which relationships are resolved by the OODBMS, for example, are relationships swizzled and, if so, when.

Describe mechanisms used for ensuring referential integrity (e.g., tombstones).

A.1.2.2 Composite Objects

Supports equality operations across composite objects.

Supports copy operations across composite objects.

Supports delete operations across composite objects.

Supports lock operations across composite objects.

A.1.2.3 Location Transparency

An object anywhere in the database may be accessed.

The syntactic structure used to reference an object is not dependent upon the object's location.

An object can be programmatically moved to a new location within the database.

A.1.2.4 Object Versioning

Supports a linear versioning policy.

Supports a branch versioning policy.

Supports the maintenance of consistent configurations (i.e., interconnections of versioned objects).

Supports the process of merging divergent versions.

Allows for application specific versioning policies.

Supports versioning of entire databases.

A.1.2.5 Work Group Support

Supports checkin and checkout of information for long periods of time.

Supports detached operation for a segment of a database.

A.1.2.6 Schema Evolution

Identify all schema changes, from the list provided in Section 3.1.2.6, Schema Evolution, that can be performed automatically.

Schema evolution must be performed off-line through a dump and load of the database (with suitable off-line data transformation).

Aggressive schema evolution is supported.

Schema evolution may be performed as a background activity.

Lazy schema evolution is supported.

Describe OODBMS facilities for supporting application controlled schema changes.

Provide examples of how all schema changes can be achieved if not performed automatically by the OODBMS.

A.1.2.7 Runtime Schema Access/Definition/Modification

Database stores schema definitions as objects.

Database provides an interface to access the schema definitions, and from that information, access objects.

Database provides an interface to create and modify the schema definitions.

A.1.2.8 Integration with Existing DBs and Applications

Describe the approach to access legacy databases. This may include capabilities provided by the vendor or by third party tool developers.

A.1.3 Database Architecture

A.1.3.1 Distributed Client - Server Approach

Supports distributed access to data.

Uses a client-server architecture.

Transfers data in units larger than an object.

Describe the overall architecture of OODBMS including all interacting processes (e.g., client, server, lock, etc.) which are required for concurrent, multi- application, distributed database access.

Describe mechanisms used to manage secondary storage (e.g., as files, as disk partitions, etc.)

A.1.3.2 Data Access Mechanism

Describe how the OODBMS transfers data between servers and clients, and how and when inter-object references are resolved will not be provided. (OODBMS performance will be closely tied to the data access mechanism, thus an evaluation of this feature is best supported by benchmarking.)

A.1.3.3 Object Clustering

Supports application controlled clustering of objects.

A.1.3.4 Heterogeneous Operation

Supports heterogeneous operation.

Describe the process by which data is translated between processor formats and the times when this transformation occurs.

A.1.4 Database Functionality

A.1.4.1Access to Unlimited Data

Provides access to a large data space.

Supports storage of large transient objects.

A.1.4.2 Integrity

An OODBMS cannot guarantee absolute database integrity since data is directly mapped into the application's address space. Techniques for achieving database integrity, and the section which covers the identified technique, are:

- integrity of inter-object relationships, see Section 3.1.2.1, Relationships & Referential Integrity,
- consistency of instances and schema, see Section 3.1.2.6, Schema Evolution,
- data access integrity, see Section 3.1.1.6, Encapsulation,
- application level integrity checks, see Section 3.1.4.10, Constraints,
- consistency in the presence of concurrent updates, see Section 3.1.4.3, Concurrency, and Section 3.1.4.5, Transactions.

A.1.4.3 Concurrency

Provides standard pessimistic concurrency control.

Provides multiple writers of data resulting in versioned data sets.

Provides a multiple reader, single writer concurrency control mechanism.

Describe the concurrency control policies provided by the OODBMS and how an application's access to data is affected by these policies.

A.1.4.4 Recovery

Provides recovery from application failures.

Provides recovery from system failures.

Provides recovery from media failures.

Describe the use of and overhead associated with journal files, checkpoints, etc., required to ensure database recovery.

Describe the processing required to achieve recovery from each of the failure types listed above.

A.1.4.5 Transactions

Transactions can access data located in any portion of the distributed database.

Objects accessed in a transaction can remain in the client cache after transaction commit.

Supports long transactions.

Supports nested transactions.

Describe the transaction processing and associated overheads. Describe the use of locks for transaction implementation, especially with regard to how and when objects are locked and when locks are released.

A.1.4.6 Deadlock detection

Deadlocks are detected.

Deadlocks are resolved by aborting and restarting a transaction.

A.1.4.7 Locking

Supports object level locking.

Supports page level locking.

Implicitly acquires and releases locks.

Requires no application support for determination of lock modes.

Provides interface for manually requesting and releasing locks.

A.1.4.8 Backup and Restore

Supports backup.

Backup can be performed while the database is being accessed.

Backups can be performed on specific segments of the database.

Supports incremental backup.

A.1.4.9 Dump and Load

Supports dump of a database.

Supports dumping of specific segments of the database.

Supports recreating a database by loading from a dump file.

A.1.4.10 Constraints

Supports specification of logical constraints.

Defines model for application controlled constraint execution.

Automatically invokes constraint execution.

A.1.4.11 Notification Model

Supports passive notification of object modifications.

Supports passive notification of general database events.

Supports active notification of object modifications.

Supports active notification of general database events.

A.1.4.12 Indexing

Supports indexing across all objects of a class based on data members of that class.

Supports indexing of a class and its subclass.

Indexing is supported across the entire database.

Indexing can be restricted to segments of the database.

An application may select from alternative implementation technologies for an index.

A.1.4.13 Storage Reclamation

Unused storage space is reclaimed.

Storage reclamation is performed incrementally or as a background activity.

A.1.4.14 Security

Provides multi-level mandatory access controls.

Provides discretionary access controls.

To what security level (e.g., C2, B2, ...) is the database certified.

A.1.5 Application Programming Interface

A.1.5.1 DDL/DML Language

Language(s) used for data definition (DDL).

Language(s) used for data manipulation (DML).

A.1.5.2 Computational Completeness

The DML is computationally complete (i.e., suitable for large scale applications development).

A.1.5.3 Language Integration Style

Provides a library-style interface to the database (loose integration).

Provides an inheritance-based interface to the database (tight integration).

A.1.5.4 Data Independence

Language supports data independence.

Language supports derived attributes.

A.1.5.5 Standards

Vendor is a member of the ODMG working group.

Product uses standard version of a programming language.

Product uses third-party supplied compilers.

A.1.6 Querying an OODBMS

A.1.6.1 Associative Query Capability

Provides a query capability.

Query capability is based on SQL.

Queries can operate on selected portions of the database.

Queries can operate on the entire database.

Queries can return a selected set of objects.

Queries can return an arbitrary type.

Queries can return text suitable for report generation.

A.1.6.2 Data Independence

Queries are optimized to make effective use of schema associations and indexes.

A.1.6.3 Impedance Mismatch

Reduces impedance mismatch by providing a tight integration of query mechanism with the OODBMS application programming language.

A.1.6.4 Query Invocation

Queries can be invoked from within the application programming language.

Ad hoc queries (i.e., not from within an application) are supported.

A.1.6.5 Invocation of Programmed Behaviors

Programmed query predicates can invoke object behaviors.

Ad hoc query predicates can invoke object behaviors.

A.2 Application Development Issues

A.2.1 Developer's View of Persistence

Describe how persistent objects are accessed from within an application. In particular, are they referenced via pointer or by some other mechanism? Also, are they implicitly loaded and stored from the database or must data access be explicitly programmed? Finally, must the application explicitly identify updates to objects (so that the OODBMS knows what objects need be write locked and copied back to the database)?

A.2.2 Application Development Process

Describe the steps in developing an application including schema design, application development, coding, testing, and debugging.

A.2.3 Application Development Tools

A.2.3.1 Database Administration Tools

Database administration utility programs are provided.

Database administration tasks can be programmed.

Describe the database administration capabilities.

A.2.3.2 Database Design Tools

Describe vendor supplied and third party supplied object-oriented modeling tools that can be used for schema definitions.

A.2.3.3 Source Processing Tools

Describe vendor supplied and third party supplied tools needed for source processing. In particular, describe any integrations of the OODBMS and its tools into application development environments.

A.2.3.4 Database Browsing Tools

Describe tools provided to interactively view and modify the contents of a database.

A.2.3.5 Debugging Support

Debugging facilities can be used from third party debugging environments.

Describe specific techniques, facilities, or tools supplied by the vendor for debugging support.

A.2.3.6 Performance Tuning Tools

Describe specific facilities or tools supplied by the vendor for performance tuning.

A.2.4 Class Library

Provides a set of data abstraction classes such as lists, sets, dictionaries, etc.

Provides source code for the data abstraction classes.

Describe the class library provided with the OODBMS.

A.3 Miscellaneous Criteria

A.3.1 Product Maturity

Describe the product maturity based on criteria from Section 3.3.1, Product Maturity.

A.3.2 Product Documentation

Evaluators should obtain a copy of the vendor's documentation set as part of the evaluation process.

A.3.3 Vendor Maturity

Describe the company's maturity in terms of size, age, staff, and financial stability.

A.3.4 Vendor Training

Describe the training classes offered for application developers and database administrators.

Evaluators should attend vendor training classes as part of the evaluation process.

A.3.5 Vendor Support & Consultation

Describe the support staff provided by the vendor and the availability of hands on consulting.

A.3.6 Vendor Participation in Standards Activities

Describe the standards activities the vendor participates in, especially those listed in Section 3.3.6, Vendor Participation in Standards Activities

Appendix B – Schema-Mapping Tools

The integration of object-based applications and legacy relational databases is a critical issue for n-tier applications. Integration options include application inclusion of SQL, which results in "impedance mismatch" due to the differences between the application object data model and the relational database schema, or stored procedural processing on the database server, which eliminates server offloading techniques such as caching. Object-relational mapping is a technique, which preserves the application's data representation while supplying client-side query mechanisms. Some of the more complex issues of mapping objects to schemas arise because a relational database contains data, while objects also include identity, state and behavior.

B.1 Mapping Techniques

There are several techniques available for schema-mapping tools. Table mapping involves matching all attributes of a specific class to columns within a specific table, made difficult due to the incongruent nature of normalized tables and objects modeled from a business process. Subset mapping uses only a subset of a table's columns within a 'projection class', which contains enough information for a full table row access. Conversely, Superset mapping associates class attributes to columns spanning multiple tables, such as found in a join-query.

B.2 Mapping Inheritance Trees

The problem of mapping a class inheritance tree is generally solved through three methods, vertical mapping, horizontal mapping and filtered mapping. Vertical mapping associates each class within the inheritance tree with a table. Tables matching the sub-classes contain foreign keys to the table matching the super-class, resulting in a join-query using all the tables within an inheritance lineage. Horizontal mapping designates tables for each concrete sub-class, including within the table all super-class attributes. This form of mapping improves query responses, but is sensitive to changes in the super-class. The third type of mapping, Filtered mapping, is the inverse of Horizontal mapping: a single table representing the super-class is built, with columns for each attribute in the super-class and all sub-classes. A special filter column is built to associate a row with a sub-class. This results in fair query performance, but invalidates table-normalization rules.

B.3 Mapping Object Relationships

A one-to-one relationship is implemented in a relational database through the use of foreign keys. Objects are given attributes to match these keys. A join-query can also implement this relationship, with the result being a single object instead of a collection of objects. Objects can contain two types of one-to-one relationships, owned and referenced. Owned relationships contain creation/deletion semantics to the object on the other side of the relationship. These may result in database updates.

The one-to-many relationship comes in two flavors, an aggregation or association. An aggregate one-to-many relationship implies ownership of the many side by the one, and is usually implemented within a relational database through the use of a foreign key column for tables on the many side of the relationship which hold a primary key of the table on the one side. The ownership results in database updates that propagate through the aggregation, a technique known as update via reachability. Associative one-to-many relationships have no ownership (update semantics). Relational database implementations include the foreign key column, along with join-queries.

The many-to-many relationship is commonly implemented within a relational database through the use of a join-table, which is a table containing foreign keys to the tables representing each side of the relationship. Generally, this involves creating a new persistent class in the object model to represent this join-table.