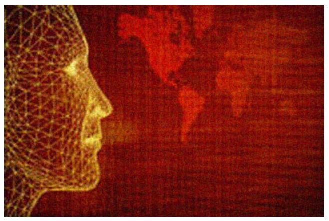


– Okeanos^{TM} –

A Self-Reliant, Distributed, Object Store

D.McClain

SpectroDynamics, LLC

Objects are Self-Describing

- Sam in Tucson adds a new class and some instances to the database:

```
(defclass address-entry ()
  ((name     :accessor address-entry-name
             :initarg  :name
             :indexed  :unique)
   (location :accessor address-entry-location
             :initarg  :location
             :indexed  t))
  (:metaclass persistent-class)
  (:oid       #$(OID {390DE170-854D-11DE-9B87-00254BAF81A0})))
;; ------------------------------------------------------------

(make-instance ‘address-entry
               :name      "Dave”
               :location  "Tucson”)

(make-instance 'address-entry
               :name      "Panos”
               :location  "Boston”)

(commit)
```

Self-Reliant Retrieval

- Sam tells Mary, in San Diego, to look up "*Panos, in Boston*", but Mary has never heard of an "*Address-Entry*"…
  - that's okay…
- Sam tells Mary to retrieve all instances of class '*Address-Entry* and see for herself…

```
(mapcar 'deref (find-instances 'address-entry))

  (#<ADDRESS-ENTRY :LOCATION Tucson :NAME Dave 200BF96B>
   #<ADDRESS-ENTRY :LOCATION Boston :NAME Panos 200DC963>)


;; hmm… interesting… what is an 'Address-Entry?
(describe-persistence 'address-entry)

  Class: ADDRESS-ENTRY
    Inherits Persistence: PERSISTENT-OBJECT
    Indexed Slots: (NAME :UNIQUE) (LOCATION :INDEXED)
    Slots:
      NAME            Initargs: (:NAME)
      LOCATION        Initargs: (:LOCATION)

;; Let's add me!
(make-instance 'address-entry :name "Mary" :location "San Diego")
(commit)
```

Object Queries

- Finding Instances of a Class
  - FIND-INSTANCES, FIND-INSTANCES*
  - FIND-INSTANCES-FOR-SLOT
  - FETCH-INSTANCES-FOR-SLOT
  - FIRST-INSTANCES-FOR-SLOT, LAST-INSTANCES-FOR-SLOT
- Mapping Instances of a Class
  - MAP-INSTANCES, MAP-INSTANCES*
  - MAP-INSTANCES-FOR-SLOT
- Smart QUERY compiler
  - Queries are performed on the server next to the data
  - Mathematically complete Set composition language:
    - Operators: AND, OR, :XOR, :SET-, NOT, :NOR, :NAND, :NXOR, :NSET- *(also :AND, :OR, :NOT)*
    - Terms: (:ALL), (= *slot-name value*), (:MEMBER *slot-name values-list*), (:RANGE *slot-name from [limit]*)
  - Compiles query clauses to most efficient form
  - Eliminates redundant database access
  - Works on indexed slots (*fastest*) and non-indexed slots

Set Composition Language

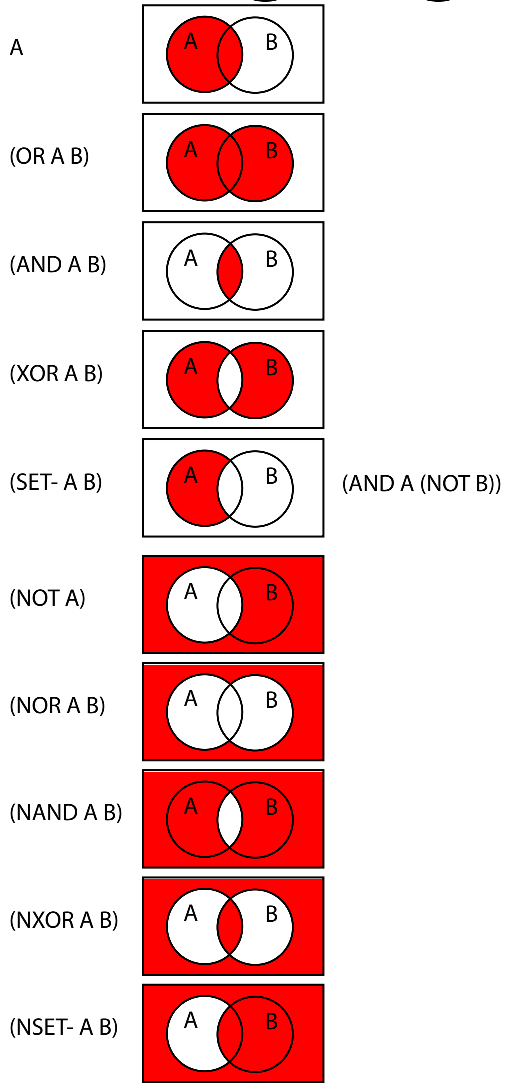
Some operations could potentially involve large amounts of data.

E.g., (NOT A) where A is a small set in a large universe.

But as illustrated in the *SET-* case *(5th from the top)*, the use of a conjunctive mask, e.g., *(AND A …)*, can make the overall operation very efficient.

The use of a conjunctive mask alleviates the need to read the entire universe.

However, non-indexed slots do require scanning the entire Class collection to find candidate instances.



Example Object Queries

```
(mapcar ‘deref (find-instances-for-slot ‘address-entry ‘location :from “A” :to “M”))

  (#<ADDRESS-ENTRY :TELEPHONE NOT-AVAILABLE :LOCATION Boston :NAME Panos 23E947A7>)


(mapcar ‘deref (fetch-instances-for-slot ‘address-entry ‘name “Dave”))

  (#<ADDRESS-ENTRY :TELEPHONE NOT-AVAILABLE :LOCATION Tucson :NAME Dave 23D78ECB>)




(mapcar ‘deref
  (query ‘address-entry ‘(AND (= name “Dave”)
                              (NOT (= location “Boston”)))))

  (#<ADDRESS-ENTRY :TELEPHONE NOT-AVAILABLE :LOCATION Tucson :NAME Dave 23D78ECB>)




;; Illustrate QUERY expression rewriting
(compile-query '(OR (AND (= name "Dave")
                         (NOT (= location "Boston")))
                    (AND (NOT (= name "Dave"))
                         (= location "Boston"))))

  (_XOR (= NAME "Dave") (= LOCATION "Boston"))
```

Classes Can Evolve Smoothly

- Sam decides to add a telephone number to ‘Address-Entry…
- Existing objects automatically evolve to the new definition upon retrieval

```
(defclass address-entry ()
  ((name      :accessor address-entry-name
              :initarg  :name
              :indexed  :unique)
   (location  :accessor address-entry-location
              :initarg  :location
              :indexed  t)
   (telephone :accessor address-entry-telephone ;; added
              :initform :not-available
              :initarg  :telephone))
  (:metaclass persistent-class)
  (:oid       #$(OID {390DE170-854D-11DE-9B87-00254BAF81A0}))) ;; OID unchanged
;; -----------------------------------------------------------

(mapcar ‘deref (find-instances ‘address-entry))

  (#<ADDRESS-ENTRY :TELEPHONE NOT-AVAILABLE :LOCATION Tucson :NAME Dave 2185DEBF>
   #<ADDRESS-ENTRY :TELEPHONE NOT-AVAILABLE :LOCATION Boston :NAME Panos 2185D8DF>
   #<ADDRESS-ENTRY :TELEPHONE NOT-AVAILABLE :LOCATION San Diego :Name Mary 2185D2FF>)
```

Self Reliant Data

- When new classes are defined, or existing classes are redefined, and instances of them are saved to a database, Okeanos^{TM} detects whether or not the database has ever seen the class definition, or if it needs to be updated for evolutionary changes.

- If the database is up-to-date, then just the instances are saved

- Otherwise, enough information is also retained about the class, and all of its superclasses, to reconstruct them when needed.

Object Identifiers – OID's

- Every object in the database is assigned an OID.
- OID's are universally unique, and carry the time of creation to the nearest 100 ns, and the MAC address of object's creator.
- Class definitions must have manually assigned OID's in order for them to be universally recognized across all databases.
- All other objects automatically receive their OID from the Okeanos™ system when they are first created.
- Objects copied from one database to another retain their originally assigned OID.

```
;; an OID from one of the Address-Entry objects
(when-created #$(OID {D97F6D9C-88A7-11DE-8CEA-00254BAF81A0}))

  "2009-08-14 07:55:35.7616540 UTC 00:25:4B:AF:81:A0"
```

Retrieval of Objects

- When objects are retrieved, their OID is returned
- This avoids potentially large amounts of data on the first look.
- Collections of OID's satisfying various criteria can be selected without needing to see the physical data in the object.
- Union, Intersection, Set-difference, operations can be applied to these OID collections
- Once the OID's of the objects of interest are decided, the physical data are retrieved by DEREF.

Persistent Objects

- Any kind of data item can be saved to the persistent store
- Persistent-Object instances, and any instances of its subclasses, automatically register with the database upon creation
- Basic data items, e.g., numbers, strings, lists, vectors, arrays, etc., need to be manually PERSIST'ed.
- You can obtain the OID assigned to any persisted object with REF.

```
(persist pi)
  3.141592653589793D0

(ref *)
  #$(OID {07075AB2-88B3-11DE-94C3-00254BAF81A0})

(deref *)
  3.141592653589793D0
```

Commit

- Objects persisted to the database do not get written to the physical store until COMMIT.
- Okeanos^{TM} uses a distributed-system 2-phase commit.
  – All databases involved in a transaction are asked to verify the integrity of the proposed changes
  – If anything is wrong, a ROLLBACK is issued to all databases, and the update is cancelled
  – If all goes well, then all databases are asked to commit their individual changes, and each of them logs their individual changes to their transaction log file.

Rollback

- In general, we have a distributed environment with multiple open databases.
- Some databases may reside on the user's host machine, others may be accessed through remote connections.
- During ROLLBACK a new running-transaction timestamp is negotiated with all participating databases. Every remote client-proxy process is assigned the same, new, transaction timestamp.
- The timestamp is used to uniquely identify the transaction start-time and MAC address of a commit originator.
- With a variety of machines involved in a multi-database transaction, the timestamp represents the supremum of time values reported by each machine. So it might not reflect local estimates of clock time.
- All databases receive the same transaction timestamp during commit so that multi-database updates can be coordinated and identified later in the transaction logs.

Transaction Time-Order (TO)

- The database system ensures proper Transaction Time-Order to provide data consistency
- Rollback-Exceptions
  - Occur during database retrieval if an item being retrieved has a write-timestamp later than the requestor's transaction timestamp. Attempting to read from the future…
  - Occur during phase-1 of the 2-phase commit if some reader with a later transaction timestamp has already read an item about to be updated by the commit. We can't change the past on them…

Transactions

- Most of the time, coordinated changes are performed inside of atomic, restart-able, transactions.
- Atomic transactions either succeed fully with a commit, or they fail with a rollback.
- If a Rollback-Exception is signaled at any point, then the entire transaction is rolled-back, caches are emptied, a fresh transaction timestamp is obtained, and the entire transaction is retried, up to some limit of retries.
- If any other kind of exception is signaled during an atomic transaction, the entire transaction is aborted with a rollback.
- If the enclosed code exits the transaction for any reason – voluntary or involuntary, the transaction is aborted with a rollback
- If an atomic transaction succeeds it will have committed the changes.

```
(atomic (:retries 3)
      (let ((entry (get-instance ‘address-entry :name “Dave”)))
        (setf (address-entry-telephone entry) “520-123-4567”)))

  {07075AB2-88B3-11DE-94C3-00254BAF81A0} ;; the TID of the committed transaction
```

When Things Go Wrong

- If a remote connection is dropped during a commit, all of the databases will be asked to roll-back.
- If the dropout occurs before the 2nd phase of the 2-phase commit, there is no problem. All databases will roll back the transaction.
- If the dropout occurs after the 2nd phase, then either:
  - The remote database did the commit but couldn't confirm to the transaction session,
  - Or the remote database never got the commit command, in which case it will automatically rollback after some timeout period.
- In either case, some of the databases may be inconsistent with others:
  - Some of the other databases may have already received a commit order, and still other databases that follow the remote database in sequence won't be given the commit order as a result of the communications problem.
  - In either case there may be some inconsistency among the databases that must be corrected later. Transaction logs will help to isolate any problems.
- In general, this is a known intractable problem:
  - See the "Two Generals Problem"

Still More Possible Problems…

- A database residing on the user's host machine might suffer a disk crash or a power outage.
- If the problem arose before the 2nd phase of the 2-phase commit, then there is no problem. The transaction was rolled back by Mother Nature (or Murphy).
- But if the problem arose during phase 2 of the 2-phase commit, the transaction may be only partially applied.
- The transaction log will show an incomplete transaction. What changes have been made can be backed out with the help of the log.
- Every update to an existing object logs a pointer to its prior version and its timestamp.
- You can ask to retrieve any prior version of an object, even if the object has been "deleted".
- Objects in a database are never destroyed. However, under user control, rebuilding a database might elide deleted objects and old versions.

High-Performance

- Under the hood, Okeanos™ runs a high-performance, direct memory-mapped image of the database file, its B-Trees, and Heaps.

- A pool of page mappers is maintained in an LRU chain. Mappers nominally represent one page of memory, but expand on demand to encompass an entire data transfer. Mappers are assigned on demand to file pointers. An existing mapper is reused when possible, else the oldest in the LRU chain is taken and remapped to the required file region. (*Remote database connections take no memory on the remote client.*)

- The Operating System furnishes an already polished page management system and file cache system. No need to duplicate its efforts. We take advantage of the OS maturity of design.

- Databases are typically recorded in the native byte-order of its host machine. But foreign databases might have opposite byte-order. Okeanos™ senses byte order when it opens the files and retains whatever convention was originally used for those files.

- Performance has been measured showing direct memory-mapped operation as 100x faster than buffered streaming I/O.

Capacity Limits

- Essentially, None…
- OID's can address a nearly unlimited number of objects. 128-bits wide, containing a MAC address and a 60-bit timestamp to the nearest 100 ns. (*That's wide enough for a new object created every 100 ns for the next 3,670 years. And we can accommodate that on each of 281 Billion machines!*)
- File pointers inside one database file are 64-bits wide, addressing up to 18 Exabytes (18×10^{18}, or 18 Billion GB).
- Logfiles, and the associated object store, can number up to 4 billion, each at least 256 MB in size (*max 4 GB*). That's a total of an Exabyte of object store – in addition to the B-Trees and Heap storage in the fast memory-mapped file.
- Max single-object size is 4 GB.
- And there are no inherent limits on the number of simultaneously opened local and remote databases. The aforementioned limits are per-database.

B-Tree Design

- High-performance B-Trees (some variant thereof).
- Each Node occupies one VM Page (4 KB) at page aligned file positions
- Leaf nodes can contain up to 505 data pointers
- Interior nodes can contain up to 253 node & data pointers.
- Only 7 levels needed to reach 10^{18} objects
- Fast Binary-search within Nodes
- Delayed node-splitting and merging, looking both left and right, (2 for 3 speedup)
- Bulk-loading capabilities (approx 5x faster than direct loading of large tables)
- Speed of bulk-loading measured at around 6,700 records / sec for records with one indexed field.

Main Object Store

- The transaction log **IS** the object store
- Transaction directory shows where transactions begin and end, who committed them, and when.
- OID mappings point to objects serialized to the file during the transaction commit.
- Object serialization and deserialization is inherently a streaming operation, and so does not benefit as greatly with direct memory mapped file I/O.
- Every object maintains a back-pointer to its prior version and its timestamp, even when “deleted”.
- Inter-object references (*not internal self-references*) are by way of stored OID’s naming linked objects.
- Object stores grow to around 256 MB and are then sealed off as read-only repositories, while a new logfile is created to become the current read/write repository.

Data Serialization

- High quality, high capability, data serialization
- Network neutral design, same encoder used for Butterfly inter-node communications
- Encodes/decodes any portable data object
  - Excludes native machine code, session environments, and functional closures
    - Data are portable items, not checkpoint-restart binaries
  - Includes
    - Packages,
    - Class Definitions,
    - Object Instances (incl unbound slots)
    - Condition Objects
    - Structure objects
    - Lists (proper and improper),
    - Vectors (incl adjustable and fill pointers),
    - Arrays (multidimensional and incl overlays),
    - Hash Tables
    - Characters
    - Strings (incl Unicode),
    - Byte vectors
    - Pathnames
    - Symbols (KW, Interned, Not-Interned),
    - Numbers (fixnums, bignums, floats, complex, any precision),
    - Named Functions
    - Self-referential objects
    - Special items: OID, UUID, UPID, UNODE
- No need to provide your own application level serialization/deserialization code
- Encoding, except for some String types, is not human readable

OK-Sets

- Ordered collections of arbitrary data
- Only one copy of data with identical content
- Uses underlying B-Trees for fast access to individual data items
- `ADD-TO-SET, REMOVE-FROM-SET, SET-MEMBER-P, MAP-SET, DOSET`
- Example: storing OID's of items having certain characteristics, retrieving those items

OK-Maps

- Ordered collections of key-value pairs
- Key and value can be arbitrary data
- `GETMAP, (SETF GETMAP), UNMAP, MAP-COUNT, FIRST-KEY, LAST-KEY, MAP-MAP`

```
(let ((my-map (make-instance ‘ok-map :name “Physical Constants”)))
  (setf (getmap my-map :C-LIGHT) 3e10
        (getmap my-map :PLANCK)  1.38e-16))

(commit)

(let ((my-map (deref (find-ok-map “Physical Constants”))))
  (deref (getmap my-map :PLANCK)))

  1.38e-16
```

OK-Tables

- Fast, efficient, storage of C-compatible data records
- Heap allocated records in the Memory-Mapped portion of the Okeanos^{TM} database
- More akin to conventional RDBMS than to Object Store
- Uses OK-Schema to designate record layouts
  - Schema are collections of column definitions
  - Columns indicate data field C-Type, byte offset to field, value constraints, normalizing functions, field name, fetch & store functions
- Fields of records can be indexed for fast retrieval based on field value
- Less flexible than Object Store
  - Records always have a fixed length, but could contain OID's to point to variable data
  - Not as easily evolved for changes in field content or number of fields
  - Intended for efficient storage and retrieval of native data items
    - INT8, INT16, INT32, INT64, FLT32, FLT64, STRING PTR, OID
    - Strings can be variable length and are stored in the string pool while pointers to these strings are stored in the data records
- Actually used internally by Okeanos^{TM} for its own management purposes
- `BULK-LOAD-TABLE, INSERT-ROW, DELETE-ROW, FETCH-ROW, MAP-ROWS, more…`

Example OK-Schema

```
(fli:define-c-struct stock-detail_t
  (date   uint32)
  (filler uint32)
  (open   flt64)
  (high   flt64)
  (low    flt64)
  (close  flt64)
  (volume flt64))

(make-instance 'ok-schema
               :name "Markets/Price-History-Entry"
               :cols '((:mjd               :number
                        :c-access-spec    date
                        :c-type           uint32
                        :constraint       mmf:i>0
                        :compare-function <
                        :value-normalizer normalize-mjd
                        :indexed          :unique)

                       (:open              :number
                        :c-access-spec    open
                        :c-type           flt64
                        :constraint       mmf:d>=0
                        :compare-function <)

                       (:high              :number
                        :c-access-spec    high
                        :c-type           flt64
                        :constraint       mmf:d>=0
                        :compare-function <)

                     … more …

               :c-type 'stock-detail_t)
```

Example OK-Table Retrieval

```
(show-stock “DIA”)

 Stock: DIA
 1426 entries
 Index  MJD         Date          Open      High       Low     Close         Volume
 ---------------------------------------------------------------------------------
    1: 732008  2003-12-16    100.67    101.56    100.60    101.50      5,991,500
    2: 732009  2003-12-17    101.34    101.86    101.11    101.86      4,867,400
    3: 732010  2003-12-18    101.89    102.83    101.70    102.81     10,491,800
    4: 732011  2003-12-19    103.04    103.07    102.38    102.74     10,810,400
    5: 732014  2003-12-22    102.61    103.63    102.61    103.61      6,459,300
 [more…]
```

Distribution

- Multiple database connections can be simultaneously open and inter-operated
- Commit and Rollback are maximally parallel across remote databases
- Objects can be copied from one database to another
  - OID remains intact – it is unique in the universe
  - Transactions across multiple databases have the same timestamps and transaction ID's
  - Rollback negotiates for a supremum timestamp across all open databases
- Optimistic Locking with TO semantics
  - No write locks until commit

Deadlock-Free

- Okeanos^{TM} is Deadlock Free
  - Read/Write locks are only used internally
  - Read/Write locks are not available to user code
  - Read Locks are held only very briefly
  - No path leads from a Read-Lock state to a Write-Lock request
  - No "Deadly Embrace" can occur
  - Write Locks used only during Commit
  - All/None Distributed Write Locking with back-off and retries
  - Read/Write locks forcibly removed when threads take too long to release them, or when threads die
  - A forcibly removed write -lock produces a permanent log-file entry indicating that something severe may have happened to the database.