Object Databases

26741 characters in 3668 words on 666 lines

Florian Moser

June 14, 2018

1 introduction

motivation
simplify application development
fully persist data from object oriented programming (OOP)
simplify design & evolution of data models
avoid translations between DB & OOP at run time
DB should support all advantages from OOP

orthogonal persistence
type orthogonality (can persist all objects in any way)
persistence by reachability (not related to type system)
persistence independence (long/short term data manipulation same)

persistence strategies
by inheritance (extend base persistent class)
by instantiation (with construction pass persist infos)
by reachability (if reachable from other persistent object)

2 object data management group (ODMG) standard

persistence by reachability
database exposes root objects, schema, types
database supports locking, transactions, metadata
objects have unique identifier, literals do not
has implementations in java, smalltalk, C++

2.1 meta

object management group (OMG)
tools & architecture for OO development
created unified modelling language UML
informal standards body for major vendors
promotes portability/interopabilities
but does not develop products

about ODMG
complementary to OMG
adds data management support

2.2 collections

extent
set of all active instances of type

types
set (unordered, no duplicates)
bag (unordered, with duplicates)
list (ordered, insert elements)
array (ordered, replace elements)
dictionary (key-value store)

operations
subset only for sets
union, intersect, difference only for bags, sets

relationships
many-to-many (collections for source, target)
many-to-one (collection for source, class inverse)
one-to-one (class for source, target)
no ternary

2.3 languages

object definition language (ODL)
supports constructing ODMG models
compatible to OMG interface definition language (IDL)
defines name, extent, exceptions, attributes, relationships
define method signatures (to be implemented later)

object query language (OQL)
looks like SQL
path expression like me.neighbour.name
union, intersection, flatten for collections
subqueries & aggregation (AVG, SUM, MIN, COUNT)
returns bags, sets (DISTINCT), lists (ORDER BY)
not complete, no explicit update

```
select values from collections where condition
where exists b in books:: b in a.books
where for all p in a.authors:: p.year < 1000
```

3 object oriented databases (OODB)

connection between OO world & database
avoids mismatch between objects/relations
provides a uniform data model
combines features & properties of both worlds
tagret is to make OO languages persistent

3.1 OODB manifesto

features of OO systems
has sets, lists, tuples, complex objects formed from simpler ones
possible to CRUD, check identity (same GUID) & equality (same state)
state only modified through public interface (encapsulation)
interface which describes only behaviour but hides implementation
substitution principle to use different type at runtime
method overriding to specialize method in child
method overloading for multiple versions of same method
late binding which chooses method call target at runtime

features of db systems
data survives program execution (orthogonal persistence)
secondary storage with indexes & buffers (efficiency)
ACID, serialization, multiuser (concurrency)
transactions, snapshots, logging (reliability)
declarative, high-level, efficient, portable query language

optional characteristics
multiple inheritance, type checking/inference
distribution, versions, long/nested transactions

need to implement
caching, query processing, locking, livecycle management
identity management for relationships & objects
new index/algorithms because joins now relationships

open choices
program paradigm (declarative vs imperative)
representation system (sets, lists, more?)
type system (generics?)
uniformity (type, method as objects?)
granularity per object/page/container (different performance)
physical references for fast dereferencing
logical references for more flexibility

beyond manifest
reflection, roles (OO)
administration, reflection, evolution, constraints (db)

4 object relational mapping (ORM)

map OO to relational model
persistence-related tasks already implemented

4.1 without ORM

duplicates (for repeated reads, need object identity)
multiple models (db conforms to program conforms to world)
impedence mismatch (OO not fully mapped to database)
transformation implementation (at design time, application specific)

transformation execution (at run time, performance impact)

4.2 set up ORM

code ↔ mapping ↔ db
top-down (code-first) or bottom-up (db-first)
inside-out (create mapping first) or outside-in (generate mapping)

4.3 hibernate (Java)

java to sql, lots of backends supported
requires a non-argument constructor & mapping
efficient to use at design time but runtime impact

mapping
XML/annotation which defines properties & relationships

supports inheritance
table per class hierarchy (sparse, subclass)
table per subclass (duplicated fields, joined-subclass)
table per concrete class (no abstract, union-subclass)

solved problems
object identity
program & db model the same
transformation is already implemented

remaining problems
impedence mismatch (not all OO features supported)
transformation needs to be done at run time
need to maintain the annotations

5 android (Java)

application model
activity renders UI & reacts to input
service executes computation
intent requests use of application components
manifest exposes components & defines start activity

data management
using an sqlite database
content resolvers processes URI requests
content provider encapsulates data management

critique
no orthogonal persistence
no completeness (definition not stored; versioning problems)
bad scalability (entire object graphs need to be persisted)

6 db4o (.NET, Java)

object-based architecture w/ physical identity
no conversion/mapping/changes to objects needed
local or client/server mode, ACID transactions, caching
uses reflection

6.1 object container

unit of work; owns one transaction
manages object identities, loads/unloads objects
refresh() after delete / rollback, special collections (consistency)

6.2 persistence

by reachability with fixed depth

transparent persistence
implement Activatable interface, commit() saves all changes

6.3 activation

only loaded to certain depth, fields set to default otherwise
occurs on collection element access, or explicit activate()

transparent activation
byte code insertions take care of instance creation

6.4 retrieve

by example (findBy, pass partly filled out POCO)
native (using predicates)
soda (decend("year"), contrain(class), uses the graph)

6.5 transactions

read-committed
can get db version of object to check for collisions
can set semaphores to avoid collisions

6.6 configuration & tuning

defragment (remove unused fields, classes, meta data)
statistics (log query, IO, network behaviour & performance)
indexes for fields (automatic or explicit)

object loading
configure activation depths (less loaded initially)
use multiple containers (less complex containers)
disable weak references (no lazy loading)

database tests
disable database schema change detection
disable testing of classes at startup (less validation)

query evaluation
set indexes appropriately
optimize native queries

6.7 distribution modes

embedded
clients use same VM with db file
direct file access (single thread, same user)
client session (multiple threads, same user)

client/server
clients use multiple VM, connect to server with db
can only use methods from object containers
use "out-of-band" signalling to transfer other messages

replication
multiple servers, redundant copies
snapshot (periodically, single-master)
transactional (operation based, immediately)
merge (clients send changes to master, periodically or instant)
developer defines replication with masters/slaves

6.8 replication

bridge between db4o & relational databases (using hibernate)
needs UUID, conflict handlers
call replicate(myObj), commit() for transactional
call entities = objectChanged(), replicate(entity) for snapshot

6.9 callbacks

event triggers (activate, deactivate, new, update, delete)
"can" prefix called before, "on" prefix called afterwards

6.10 control object instantiation

bypassing constructor (default method)
using a constructor with default values
using a translator if construction needs additional logic
using type handlers registered per class (writes/read to byte arrays)

6.11 discussion

no impedence mismatch
has orthogonal persistence (but explicit store/retrieve)

issues
depths of activation/deletion/update is a new burden
lack of synchronization on delete/update
"transparency" contradicts type orthogonality

7 Versant (Java)

object-based architecture w/ logical identity
byte code insertions take care of persistence
client has object cache, server has page cache per db

7.1 database volumes

system volume (class descriptions, object instances)
data volumes (increase capacity, optional)
logical log volume (transactions, redo-info)
physical log volume (physical data information)

7.2 versant manager

manipulates, caches, provides, marshals objects
does transaction management
distributes requests for queries & updates to server

7.3 versant server

updates, caches, retrieves objects
handles transactions, locks objects
logging, recovering, index maintenance

7.4 object descriptor table (ODT)

contains logical object identifier (LOIC)
refers to memory (direct access) or db location (retreive first)

7.5 architecture

client session with own cache, each own thread(s)
servers multi threading with locking, async IO within log buffer

7.6 java versant interface (JVI)

stores java objects (support with GC, multithreading), below java VM
client-server architecture (local cache, queries on server)

7.6.1 fundamental layer

database-centric, handlers manipulate objects
class & attributes builders define classes (schema definition)
handles from LOID/new instances (data manipulation)
FundQuery returns handle (query)

7.6.2 transparent layer

language-centric, maps classes & attributes to fundamentals
persistent java object caching & retrieval
first class object (FCO) with LOIC, tracking of changes
second class object (SCO) as part of FCO, owner has to mark SCO dirty

persistence categories
always or capable (call makeRoot() or makePersistent()) for FCO
never, explicit (call dirtyObject()) or transparent (automatic) for SCO

persistence model
persistence by reachability, elect named roots
navigate starting at identity, root, class, query

7.6.3 ODMG layer

language-centric, follows ODMG standard with transactions/collections

versant query language (VQL)
complex expressions, server-side sorting, indexing
query string which is compiled, optimized, executed on server
parametrization with $sign, late binding, can rebind

7.7 JVI client cache loader

client-side object cache, server tracks state of clients
contains query results, navigation results
dereference consists of RPC, object lookup, IO
breadth (other trees, same level) / depth (deeper level) path loading
improve efficiency with batch loading

7.8 event notification

class/object/transaction/user-defined events
register listeners to channels (global or class/object/query specific)
object hooks to intercept state changed

8 graph databases

ACID, scalable for graphs & big data, REST api
graph containing vertices, edges which may have key-value properties
API with CRUD, traversal of graph

8.1 graphs

node properties
degree centrality (number of direct connections; popularity)
betweenness centrality (if between two important nodes, influence)
closeness centrality (shortest paths to all others; monitor)

network structure
density (#direct_ties / #total_possible)
distance (number of nodes needed to connect two specific nodes)
clustering coefficient (likelihood two associates of node are associates)

social networks
connected people (calls, chats) represented as graphs
calculate degree, closeness, betweenness centrality
find out communication patterns & key users

8.2 infinite graph (C++, Java)

on top of Objectivity, has graph types & algorithms

navigator engine
result quantifier (whether to append path to results)
result handler (what happens with path in results)
path qualifier (continue path or not)
path guide (which way to continue, DFS or BFS)

8.3 Neo4j (Java)

cypher query language based on pattern matching

graph traversal
implement evaluate(Path p) returning exclude/include, continue/prune

9 objectivity DB (C++, Java, C#)

container-based architecture w/ physical identity
lot of frontends, OS
for complex structures & applications (like protein folding)

9.1 architecture

client
languages interfaces provide access to objects & schema
local storage / transaction cache
client objectivity server (data from local storage, processing done remote)
client side task splitter which can aim queries at specific dbs, containers

server
lock server which grants permissions
query server to run queries (with clustering, indexes)
data server which managers memory access

9.2 storage

scope
federation (schema, database catalog) as a file (world.fbd)
databases (container dialog) as a distributable file (person.world.DB)
container (page map, for logical partitions) consisting of pages

storage
pages exists as logical, physical, transferred & locked as an unit
objects (consisting of slots) stored in container, addressed with page id

page map
maps physical to logical pages, journal file saves mapping
on transaction, changes persisted to new defragmented page
on transaction commit, page map is updated, lock released

9.3 persistent objects

primitives & complex types (OID stored)

designer
generate federation files from POJO
partial classes to separate application/persistence code

relationships
unary, binary, to-one, to-many
referential integrity maintained by system (incl. inversion)
store directly inside objects or as array (for to-many relations)
propagate locks/copy/removal over relations

9.4 connection architecture

static functions to startup(), open connection, shutdown()
single connection to federation, many sessions, many threads

9.5 retrieval

with iterator (uses lazy filtering, therefore no sorting)
by scope name (like roots, at federation/db/container/object level)
by following references from already retreived objects
support for parallel queries, LINQ

10 ObjectStore (C++)

page-based architecture w/ physical identity
query may be executed on server
for embedded systems, desktop as small footprint database
for enterprise with multi-user, multi-db, distribution

10.1 virtual memory mapping architecture (VMA)

extends OS memory management for persistence
data is referenced with (database, segment, cluster, offset)
translated to place in virtual memory

persistent storage region (PSR)
sits between heap & stack, acts as new layer of cache
serves as secondary storage, persistent over transactions
if data not found in PSR then page fault to ObjectStore
if PSR full then retranslation of pointers, updates to DB

10.2 server

manages databases & transaction logs
enforces ACID with permits, 2PC
database stores pages of c++ memory

10.3 client

pages automatically fetched from db
memory file per process with fixed size
commseg memory file per process describing the cache (permits, locks)
cache manager shared by all clients on machine (handles commseg)

10.4 cache forward architecture (CFA)

permits are tracked by the server (represent ownership)
locks taken by client according to permit (no-lock, read, write)
on page request, client checks for permit states & resolves conflicts
read/write permits for multiple/single client
server does call-back to revoke read/write permit
fast because data/locks cached across transactions

10.5 queries

specify element type, query string, database
query can be string or regex
nested queries & simple function calls allowed

10.6 persistence

persistence by instantiation (overloaded new operator)
orthogonal to types, both transient & persistent objects

10.7 developing applications

static initialize(), static shutdown()
write persistent classes, schema file, application logic
need to compile schema & link binaries to pssg

11 the OM data model

extended entity-relationship model for OO data management
simple type graphs but rich classification due to separation

11.1 type layer (representation)

for representation, data format, operations, inheritance
do typing (inherits) for employee to person

supported values
object types with identity
primitive (int), structured (struct), bulk (array) types without identity

type units
type is represented by multiple type units
has fields with (name, type, bulk) like (name, string, uni), no inheritance

information unit
instances of type unit has corresponding information unit
browse information unit by casting to its type given object identifier

add/remove type instances
dress creates information unit with default values and attaches
strip removes information unit and discards values

11.2 classification layer (roles/kinds)

defined based on types from type layer
do classification (is-a) for male/female employees to employees

collections
unary (set) or binary (relations)
membership constrained by type (multiple possible)
can apply cardinality & evolution constraints
can react on membership changes (like state machines)

collection behaviours
set Person (no duplicates, no order)
bag <Person> (duplicates, no order)
sequence [Person] (duplicates, order)
ranking |Person| (no duplicates, order)

subcollection behaviours (denoted with arrows)
arrow from collection to other collection
equal (same elements, maybe different behaviour/types)
strict (subsequence, subranking)
total (all instances of type from larger collection)

structure
arrows from multiple source collections to single target collection
disjoint (at most in one source collection)
cover (in at least one source collection)
partition (in single source collection)
intersection (if in all source then in target, arrows point the other way)

associations
relation is of the form (domain, range) like (source(o1), target(o2))
cardinality constraints (0:$*$ means $\geq$0, 1:$*$ means $>$0)
behaviour represented same as normal collections (set, bag, etc)
ternary/attributed relations not supported
nested associations possible (domain/range can be relations too)

kinds, roles
kind (fixed not-changable classification like person or postgrad)
roles (changes during livecycle like baby to adult to senoir)

evolution
assume each collection has single root (definitely a kind)
kind is fixed classification in context of parent (senior of kind human)
therefore kind can migrate if roles connected to kind are lost
to migrate element x from C_1 to C_2, denoted as x :: Postgrad $\rightarrow$ Lecturer
(1) x does not belong to subcollection of C_1 (so x must be a leaf node)
(2) for all kinds (kinds(C_1) - kinds(C_2)), some role r $\notin$ roles(C_2)
example x :: Postgrad $\rightarrow$ Lecturer
(1) trivial because Postgrad is leaf node
(2) {Postgrad, Person} - {Person}, no r $\in$ roles(Postgrad) in roles(Lecturer)

11.3 object model language (OML)

data definition, manipulation & query language
declarative, object-oriented, for object data model

data definition language
object/structured type definition
method definition/implementation
define collections, associations, constraints

```
create type person subtype of contact { age :: int }
method getWork() returns (location: set of location)
create collection WorksFor as set of (person, organisation)
```

data manipulation language
CRUD, dress & strip operations

query language
expressions & functions for primitive types
operations to access properties & execute methods for objects
operations on collections based on collectional algebra

12 storage and indexing

manage large data set on persistent storage
structuring, clustering, management of complex objects
references, type inheritance hierarchies, multi-valued properties

12.1 terminology

query
point, range query (exact value, interval matching)
hierarchy / single class (instances with / without hierarchy)

index
unique, non-unique (key, non-key fields)

sequential, non-sequential keys (ordered, unordered)
one / multi-dimensional (index over single, multiple fields)
compound (one-dimensional index over multiple concatenated values)
placing/clustering (search physical structure)

12.2 mapping

sequential organisation
data organised in pages, index maps ids to physical locations
queries have to traverse entire data set

subspace mapping
data decomposed into (overlapping) subspaces
queries traverses B-trees, K-tress, grid files (x,y boundaries)
leaves contain physical location of entries

point mapping
data mapped to specific place in memory
query by using hash functions

12.3 data structures

B-tree
each block contains n indexes, n+1 references
balanced, O(log n)

B+-tree
like B-tree, but leaves contains direct value
redundant indexes pulled down for references to the right
easier for aggregated searches, breadth-first search

linear hashing
bucket for each hash result

extensible hashing
start with smallest index as possible (e.g. 2bit keys)
extend to more bits if needed

bloom filter (existence test)
m sized array with boolean entries, n hash functions
entry n times hashed, activates target boolean entries

inverted files
search structure called vocabulary (indexes distinctive keywords)
inverted list for each keyword, storing id of records with that keyword
contrary of inverted index

signature files
each record has a fixed width index information (bloom filter)
keywords are hashed, and looked up if index information indicates

k-d tree
tree with k dimensions indexed, splits region into subregions
search for key with multiple dimensions closest match in tree

r-tree
similar purpose than k-d tree
differences include balanced, disk-oriented, rectangle partitioning

other trees
H-tree, hB-tree, Quadtree, TV-tree, cell tree

12.4 type hierarchy indexing

when using an object in query implicitly using type hierarchy
build index with type or key as top-level element

single class index (SC-index)
construct search structure for each type containing its subtypes
query evaluator needs to traverse for all used components

class hierarchy index (CH-index)
one search structure for all indexes types
query evaluator scan through once, collects all types

class division index (CD-index)
compromise between query and storing all in one node
q (# of search structures for any type), r (# of replication of types)
types are stored aggregated in a node, or combined freely at runtime

multi-key type index (MT-index)
type membership is just another attribute
on evaluation, collect disk addresses and disregard if not qualifying

12.5 aggregation path indexing

avoid full traversal of paths & intermediate object loading
nested index provides direct access between start/end objects of relation
path index stores all paths to ending objects (predicates possible)

multi-index (MX)
divide paths into subpaths of length one
for each relation, stores source/target relation
for backwards traversal (start at attribute, then get objects)

access support relations (ASR)
can-extension aggregates paths from 0 to n
left-complete extensions with all paths from 0, possibly till n
right-complete extension with all paths to n, possibly starting at 0
full extensions with all partial paths between 0 & n
answer queries starting at end points efficiently

nested index (NX)
full backwards path traversal (n to 0, like ASRcan backwards)

path index (PX)
backwards path traversal (n to ?, like ASRright)

join index (JX)
binary join indexes at both sides of the relation

12.6 collection operation indexing

signature files (like bloom filter)
element signature summed up equal signature for object
create signature for query too, then compare with object signatures

13 version models

13.1 basic aspects

like long running & nested transactions

granularity
files, tuples of relation, attributes of class, entire object

organisations
set, list, tree, (directed acyclic graph) DAG

reference types
specific (reference single version of object)
generic (references object, upon usage dereference specific version)

storage
complete version of objects
forward/backward state/operation-based delta (changes) between versions

operations
create / branch / merge / delete

interaction models
automatic versioning
explicit high level user operations (check-out, commit, update)
implicit query expansions (when using generic references)

queries
need to select appropriate versions
for generic references dereference sequential/parallel versions

13.2 temporal databases

AS-OF operation (time of transaction, physical time)
WHEN operation (real world occurrence)
user defined time

classifications
snapshot (single version managed)
static roll-back (AS-OF, space overhead)
historical (WHEN, allows to change historic data)
temporal (both WHEN / AS-OF)

representations
tuple versioning (keep old rows, new column defines newst version)
attribute versioning (each attribute has temporal info)

bitemporal conceptual data model (BCDM)
adds columns transaction time, valid time, until changed, now (boolean)
TSQGL 2 with VALIDTIME, WHEN clauses

models
homogeneous if all attributes changed at the same time
heterogeneous if attributes changed at different times

anomalies
vertical if multiple tuples for single entity (tuple versioning)
horizontal if entity spread over multiple relations (attribute times vary)

storage models
primary store for current versions, for non-temporal queries
history store for other queries
reverse chaining (references to next older version)
accession lists (reference to version table, then version access)
stacked versions (combine accession lists & reverse chaining)

spatio-temporal data query
query moving objects with position uncertainty
(sometime/always,possibly/definitely) inside (trajectory, range, t_1, t_2)
therefore 8 possible operators, some semantically equivalent

13.3 engineering databases

handle complex & hierarchical object structures
support versioning through incremental development / trial and error
handle linear revisions and (non-sequential) variations

modelling primitives
component hierarchies (is-part-of)
version histories (is-derived-from)

design management
identify current version, describe dynamic configuration
change & constraint propagation (inherit attributes from related)

13.4 software configuration systems

goal to fully automate building final product
build around design objects (source code, modules)
manages dependencies & references

product space
describes product organisation
several design choices possible, e.g. software with different requirements
relationships (root is product, visualizes dependencies)
logical structure (modules import others explicitly)
file system (module in files, build file defines relationships)
data model (tree with dependencies as leaves, generate build info)

version space
defines how objects are versioned
keeps invariants and deltas
revisions keep track of history, variants capture alternatives
one-level representation (sequence, tree, DAG)
two-level representation (revisions, variants)

14 relational DB knowledge base

14.1 SQL

data definition language (DDL)
definition of data models
relations, attributes, keys
modify tables, attributes, keys
"class definition"

data manipulation language (DML)
creating & management of data
relation tuple, predicates
execute inserts, updates, deletes
"writing of values"

query language (QL)
concepts supporting retrieval of data
query, predicate, query result
use projections, selections, joins
"reading of properties"

14.2 transaction isolation levels

read uncommitted (even uncommitted values can be read)
read committed (only committed values can be read)
repeatable read (always same value read)
serializable (possible serial execution order)