Object Databases

51161 characters in 6965 words on 1408 lines

Florian Moser

June 13, 2018

1 introduction I

approach a new project
show the problem first
identify MVP (whole solution stack)
then present a solution

product owner
lives for project
has background infos (technology stack, DevOps)
able to define iterations (specifically MVP)
supports other developers (chooses the right architecture)

basic implementation tipps
show how UI is supposed to look with mockups to customer
model database close to real world, as this does not change

OO vs database
class vs relation
object vs row
value vs cell
attribute vs domain

programming paradigns
declarative programming (functional, logical)
imperative programming(procedural, object-oriented)

2 introduction II

2.1 motivation

simplified application development
OO data should be fully persistable
database should benefit OO advantages
simplify design & evolution
no translation at run time

2.2 orthogonal persistence

type orthogonality
all objects can be persisted in any way

persistence by reachability
identifying persistent objects not related to type system
livetime of object determined by reachability of root elements

persistence independence
manipulation of long/short term data looks same

2.3 persistence strategies

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

2.4 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"

2.5 types

2.5.1 specification

properties (attributed, relationships)
operations
exceptions

interface
defines abstract behaviour

class
defines abstract behaviour & state

literal (struct, int)
defines abstract state

2.5.2 implementation

language binding, multiple per specification

representation
data structures motivated by abstract state
instance variable for each abstract variable

methods
procedure bodies motivated by abstract behaviour
possibly private methods not included in spec

subtyping (relationships)
is-a (behaviour)
extends (state & behaviour)

3 OMDG standard

3.1 object management group (OMG)

tools & architecture for OO development
distributed object management
create unified modelling language UML
informal standards body for major vendors
promotes portability/interopabilities
does not develop products

3.2 object data management group (ODMG)

complementary to OMG
adds data management support
object definition language (ODF)
object query language (OQL)
bindings to java, smalltalk, C++

object model
based on OMG object model
objects have unique identifier, literals do not
state defined by properties
behaviour defined by operations
categorization possible, common properties & operations

extent
set of all active instances of type

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

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

relationship
many-to-many (collections for source, target)
many-to-one (collection for source, class inverse)

one-to-one (class for source, target)
no ternary

persistence
by reachability
database exposes root objects, schema, types

other concepts
database operations, locking, transactions, metadata
built in dates, times, intervals

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
can use path expression like me.neighbour.name
not complete, no explicit update
returns bag, set with distinct, list with order by
can use subqueries
aggregation for collections (AVG, SUM, MIN, COUNT)
union, intersection, except for sets
can flatten collections

OQL example
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

4 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
managing object data, making OO languages persistent

4.1 OODB manifesto

define features, optional characteristics, open choices
but important properties missed (according to relational db people)

4.2 features of OO systems

complex objects
complex objects formed from simpler onces, constructors
tuples (represent entities & attributes)
sets (collection of entities)
lists (capture order)
need transitive retrieval, CRUD, copy
need identity (same GUID), equality (same state)
db only knows sets, atomic tuples

identity
if GUID or state matches then equal
sharing with references
shallow & deep equality

encapsulation
object data & methods implement interface
state only modified through public interface
data exposed for declarative queries

types
define object properties, static safety
object structure & behaviour, separated from interface

generalizations
enable better modelling, reuse, semantic complexity
inheritance (attributes & methods from superclass)
specialize or generalize

inheritance
substitution (more operations, based on behaviour)
inclusion (based on structure)
constraint (like inclusion, sub is constraint on super)
specialization (sub contains more specific info)

method overriding
redefine, specialize method in sub

method overloading
multiple versions of method exist

late binding
appropriate method call is selected at runtime

computational completeness
any computable function expressible

extensibility
devs can add native types to database

4.3 features of db systems

persistence
data survives program execution
orthogonal implicit persistence
orthogonality of type system & persistence (all can be saved)

efficiency
secondary storage (indexes, buffers) optimize

concurrency
multiuser, serialization
atomicity, consistency, isolation, durability

reliability
resilient to user, software, hardware failures
transactions, snapshots, logging

declarative query language
high level (conciseness of non-trivial tasks)
efficient execution (can be optimized further)
application independent (any database may execute it)
say ”what” not ”how”

4.4 optional characteristics

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

4.5 open choices

program paradigm (declarative vs imperative)
representation system (sets, lists, more?)
type system (generics?)
uniformity (type, method as objects?)

4.6 more tools

database administration
view definition / view derived data
objects with roles (dynamically add/remove, like traits)
database evolution (migrate gracefully)
set integrity, semantic, evolution constrains
define, modify & enforce constrains

4.7 technical overview

architecture
vary greatly, not simple server-client model
all do caching, locking for query/transaction processing
need livecycle management for objects

granularity
per object/page/container
different runtime / performance characteristics

querying
try to remove join concept and replace with relationships
performance depends on execution place, flexibility of query, indexes

identity management
for relationships, uniqueness
physical for fast dereferencing but limited flexibility
logical for immutability but needed access translation

5 object relational mapping

5.1 mappings

db $\leftrightarrow$ program (interface)
db $\leftrightarrow$ world (enterprise modelling)
program $\leftrightarrow$ world (simulation)

5.2 problems

object identity (serial/read out produces duplicates)
multiple models (db, program, world)
impedence mismatch (map OO to database)

implement transformation (design time)
execute transformation (run time)
application-specific transformation (therefore reimplement often)

5.3 relational database

ok for small applications
not ok for large with inheritance, multi-value attributes

5.4 object relational mapping (ORM)

map OO to relational model
persistence-related tasks already implemented
persistence API (java, annotations define mapping)

set up ORM
top-down (oop → mapping → rdbms)
bottom-up (oop ← mapping ← rdbms)
inside-out (oop ← mapping → rdbms)
outside-in (oop → mapping ← rdbms)

5.5 hibernate

java to sql, lots of backends supported

requirements
ensure there is a non-argument constructor
create mapping (id, property, set (key, relationship))

configuration
connect to database, specify name & DBMS

session factory
factory = new Configuration().configure().buildSessionFactory()
factory.close()

session
session = factory.openSession()
session.beginTransaction()
save, close, query, update, delete
session.beginTransaction()
session.createQuery("FROM persons")
session.getTransaction().commit()
session.close()

associations
unidirectional, bidirectional, ordered
one-to-one, many-to-one, many-to-many

inheritance
table per class hierarchy (sparse, subclass)
table per subclass (duplicated fields, joined-subclass)
table per concrete class (no abstract, union-subclass)
strategy can be defined for each part of application

annotations
replace xml with annotations, same keywords

solved problems
object identity
implementation transformation
application specific transformation

remaining problems
impedence mismatch
multiple models
transformation at run time
annotation maintenance (new)

discussion
good at design time, bad at run time

6 android

application model
activities for UI
services for computation
providers for data
intents request use of application components
manifest files exposes components & defines start activity

livecycle
activity reacts state changes

data management
SQLiteOpenHelper creates SQLiteDatabase
Cursor c = db.rawQuery("SELECT * FROM persons")
c.moveNext(), c.isLast(), c.getString(2), c.getInt(3)
ContentValues object for CRUD (simple map)

content resolver
reacts to URIs like content://ch.acm.personprovider/pictures/12
content resolver invokes correct provider

content provider
encapsulates data management, can be exposed
CRUD, getType() methods, uses Cursor, URI, ContentValues

example location
multiple providers (GPS, WLAN)
listener waits for updates
manager chooses best provider, has last known location

lack of
orthogonal persistence (type orthogonality, independence, identity)
completeness (definition not stored; versioning problems)
scalability (entire object graphs need to be persisted)

7 db4o

object-based architecture w/ physical identity

7.1 meta

open source native object database for .NET, java

key features
no conversion/mapping needed
no changes to objects to be persistent
local or client/server mode, single line of code
ACID transactions
caching & integration in native garbage collection

7.2 architecture

file/in-memory database
I/O adapter
ACID transactional slots

object part
marshaller
reference system, reflector, class metadata
api

query part
class/field index b-trees
index query processor
SODA query processor
native & SODA queries, query by example

7.3 object container

connection to database
unit of work; owns one transaction
manages object identities, loads/unloads objects
starts new transaction after commit/abort
commits implicitly after closing container

persist
store(), delete()
on create persistence by reachability
on update depth is 1 per default, only primitive values
on delete no cascade per default
objects linked by weak reference
config.common().objectClass(Author.class).cascadeOnDelete(true)

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

consistency
refresh() syncs DB state, call after delete()
database-aware collections

7.4 transparent persistence

let objects implement Activatable interface
objects bound to framework on retrieve
modify at will, then call commit to save changes

7.5 activation

only loaded to certain depth, fields set to default otherwise
occurs on collection element access, or explicit activate()
depth tradeoff is manual activate() vs heavy memory usage

transparent activation
on property set, activate property
database registers itself on instance creation
byte code insertions does this automatically

7.6 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)

7.7 transactions

thread-safe, but single-thread core
no data loss, automatic recovery on system failure
rollback discards changes, call refresh() to clean up
read-committed (only committed values can be read)

collision detection
peekPersisted() to get unbound instance of db version
get read committed or stored values (configurable)

collision avoidance
db.ext().setSemaphore(GUID, MAX_TIME_WAIT)

7.8 discussion

no impedence mismatch

orthogonal persistence
independence (yes)
data type orthogonality (yes)
identification (yes)
but explicit store/retrieve application logic

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

7.9 configuration & tuning

defragment
remove unused fields, classes, meta data
compact database file

statistics
log query behaviour & performance
log IO/network activity

log
get all objects & classes in db file

indexes
optimize query evaluation
tradeoff between query & modifications
B-trees on single fields; automatic or explicit

7.10 speed tuning

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

7.11 distribution

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

client/server mode
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

7.12 replication

separated from core
bridge between db4o & relational databases
uni/bidirectional replication between hibernate, db4o
transfers data between providers

steps
configure to use UUIDs & commit timestamps
create replication object & define conflict handler
configure direction (bidirectional is default)
call replicate(myObj), commit() on each object (transactional)
call objectsChanged and then replicate(iterator) (snapshot)
call close()

7.13 callbacks

event triggers (activate, deactivate, new, update, delete)
can prefix called before, on prefix called afterwards
implement as much as needed

use case
record/prevent updates
check integrity / set default values
create indexes dynamically if used often

7.14 control object instantiation

can be configured per class/project

bypassing constructor (default method)
if no constructor works, if framework supports

constructor usage
all constructors are tested with default values
first one which works is used from now on
if none found, object is not persisted

translator
implement ObjectConstructor interface
convert object to custom entity (like Object)

type handlers
like translators, but at lower level
translator with byte arrays, handler converts objects

8 Versant

object-based architecture w/ logical identity

8.1 meta

commercial OODBMS
object database for java

company
now owned by actian
market leader in ODBMS
telecom, military, financial, transportation

architecture
RAID, NAS, raw devices/filesystems
virtual system layer
versant server (logical, physical log file)
versant network layer
versant manager
C, C++, java interface

languages
java versant interface (JVI)
versant query language (VQL)

8.2 dual cache

client has object cache
server has page cache (for each db)

8.3 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)

8.4 versant manager

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

8.5 object descriptor table (ODT)

logical object identifier (LOIC) → memory, db location
used on property access, defines if retrieved from memory or db

8.6 versant server

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

8.7 thread architecture

clients
multiple session with own cache
have multiple assigned client threads

servers
multiple server threads process client requests
access page cache, respect its lock table
log buffer thread with async IO of uncommitted writes
background page flusher which writes modified pages

8.8 java versant interface (JVI)

store java objects (fine with GC, multithreading)
client-server architecture (local cache, server queries)
sits below java VM

8.8.1 fundamental layer

database-centric, handlers manipulate objects
class & attributes builders define classes
create handles with LOID, new instances
use handle to get/put values
fundamental query/result

examples

```
//schema definition
AttrString name = session.newAttrString("name");
AttrBuilder nameBuilder = session.newAttrBuilder(name);
var attrBuilders = session.withAttrBuilders({ nameBuilder })
ClassHandle person = attrBuilders.defineClass("Person");
person.createIndex(\name", Constants.UNIQUE_BTREE_INDEX);

//data manipulation
person = session.locateClass("Person");
Handle florian = person.makeObject();
florian.put(name, "Florian");
String val = florian.get(name);
String loid = florian.asString();
florian = session.newHandle(loid);

//querying
FundQuery query = new FundQuery(session, "select name from Person");
FundQueryResult result = query.execute();
Handle resultSetMember = result.next();
```

8.8.2 transparent layer

language-centric, maps classes & attributes to fundamentals
persistent java object caching & retrieval

first class object (FCO)
have LOIC, save, query, retrieve individually
changes saved automatically, applied on commit
references to other FCO always valid
transient fields not in db
deleteObject() for db, collected by GC later (then finalize() called)

second class object (SCO)
saved as part of FCO, can't be queried
java byte stream if no versant type
transient fields not in db
FCO references serialized separately
changes applied only if owner marked as dirty on commit
if in two FCO, will be two different instances after fetch
delete implicitly by reference removal

persistence categories
for FCO (p, c) possible, marked dirty on modification
parent class must be same category
(p) always, new instances directly persistent
(c) capable, makeRoot(), makePersistent() or reachable persistence
for SCO (d, a, n) possible
(d) transparent dirty owner, sets owner as dirty
(a) persistence aware, can modify FCO, must call dirtyObject() explicitly
(n) not persistent, can't access fields of persistent object

persistence model
persistence by reachability
can elect named roots of graphs for retrieval
navigate starting at identity, root, class, query
versant transparently locks & retrieves

example

```
TransSession session = new TransSession("myDB");
Set<P> ps = session.findRoot("rootName");
Person florian = new Author("Florian");
ps.get(0).addAuthor(florian);
session.commit(); session.endSession();
```

8.8.3 ODMG

language-centric, transaction-model, collections
are FCO, follow ODMG standard

queries
additional functionality, iff persistent collection, (p) objects
existsElement, query, select, selectElement
only elements of collection queried

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

example

```
Publication pub = new Publication("Web 2.0 Survey");
String q = "select name from Author where name = $name";
Query query = new Query(session, q);
query.bind("name", "Stefania Leone");
QueryResult result = query.execute();

q = "select name from Author where Author::Books subset_of $books";
query = new Query(session, q);
query.bind("books", new [] {florian});
```

8.9 application development

persistence aware java classes
deal with sessions, transactions & concurrency
specify persistence category for classes
enhancer performs byte-code changes
create db & run application

byte code enhancement
create object in db on first instance construct
read/write objects, attributes to/from db
FCO pointers evaluated using ODT

8.10 sessions

all actions must be performed in sessions
access to db, methods, data types, persistent objects
multiple sessions possible
must close sessions explicitly
client session has object cache, ODT
server session has page caches in shared memory

8.11 transactions

always in transaction, commit/rollback starts new one
endSession commits last one
atomic, consistent, independent, durable, coordinated (with locking), 2PC
commit() flushes cache, releases locks
checkpointCommit() retains caches, locks
commitAndRetain() retains caches, releases locks

8.12 object livecycle

creation of persistent objects (memory, versant cache)
commit (data written to db, hollow proxy remains in memory)
rollback (new objects will be dropped)
query (evaluated on server, proxy created for each result)
access (fetch or deserialize object)

8.13 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

improve efficiency
vendor specific batch loading

configurable strategies
breadth (other trees, same level), depth (deeper level), path loading

8.14 collections

standard collections supported (list, array, hashtable)
FCO collections (VVector, VHashtable)
SCO collections (DVector)
FCO large collections (LargeVector, fine-grained locking)
ODMG collections

8.15 event notification

from db to registered clients
class events (CRUD of any instance)
object events (CRUD of certain element(s))
transaction demarcation (begin/end transaction)
user-defined events

event channels
register listeners to channels
global namespaces of channels over applications
class, object, query based
EventClient, ChannelBuilder

persistent object hooks
at any sort of state changes
transient attributes, caches, housekeeping of integrity
activate, deactivate, pre/post read/write, delete
can change other objects in hooks

8.16 schema evolution

add/rename/remove leaf classes
change class methods, attributes
does lazy updates of instances

8.17 polymorphic indexes

enhance performance of retrieving object with its subclasses

9 social network analysis

9.1 social networks

connected people (calls, chats) represented as graphs
calculates degree, closeness, betweenness centrality

find out
key persons of a group of people
message traversal through group
communication patterns

nodes
key/value pair of people

edges
key/value with properties & assigned weight
uni/bidirectional, explicit/implicit, short/long, single/multiple
traffic with particular keyword
work/behaviour patterns
activity such as transfers, payments

9.2 graph

9.2.1 path lengths

150 max social relationships (dunbars number)
4.56 avrg distance of publications connected to erdos (erdos number)
2.946 avrg distance of movies made with kevin bacon (bacon's number)
6 avrg distance to know everyone on the world

9.2.2 node properties

degree centrality
number of direct connections of a node

betweenness centrality
between two important nodes
high influence over what flows through the network

closeness centrality
node with shortest paths to all others
can monitor information flow best

9.2.3 network structure

network centralization
centralized if one or few central nodes
removing these nodes leads to fragmented network

density / cohesion
#direct_ties / #total_possible

distance
minimum number of nodes to connect two specific nodes

clustering coefficient
likelihood two associates of certain node are associates too
high means high clustering

9.3 general model

combines different sources, different formats to single model
uniform analysis, uniform result presentation
description in triplets (subject,attribute,object) like Linked Data

10 graph databases

10.1 general

meta model
graph containing vertices, edges
edges, vertices may have key-value properties

API
supports CRUD of metamodel
maybe support traversal of graph
maybe has graph algorithms implemented

characteristics
ACID, scalable for graphs & big data, REST api

examples
Objectivity InfiniteGraph, Neo4j, OrientDB

10.2 infinite graph

on top of Objectivity, has graph types & algorithms
distributed graph database

usage
extend BaseVertex, BaseEdge to use
markModified() after modify, fetch() before read
graphDb.addVertex(v), grapgDB.addEdge(v1, v2, EdgeKind.BIDIRECTIONAL)
graphDb.getNameVertex("name")
trans = graphDb.beginTransaction(AccessMode.READ_WRITE)

navigator engine
result quantifier (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)
implement Qualifier.qualify(Path p)

10.3 Neo4j

usage

new GraphDatabaseFactory().newEmbeddedDatabase("name")
graphDb.beginTx() tx.success(), tx.failure(), tx.finish()
n = createNode(), n.setProperty("prop", "val")
rela = n.createRelationshipTo(n2, R.MY_TYPE),
rela.setProperty("name", 30)

integrate with java
write wrapper with Node n as private final

graph traversal
td = graphDb().traversalDescription().breadthFirst()
td.relationships(R.MY_TYPE, Direction.outgoing), td.evaluator(eval)
tra = description.traverse(n), for Path p: tra
implement evaluate(Path p) returning exclude/include, continue/prune

algorithms
shortest paths, given length paths, all paths between n1, n2

rest APi
CRUD, traversals, algorithms

cypher
query language based on pattern matching (START, MATCH, RETURN)
START n=node(12) MATCH n-[:author]→b RETURN b.email
relationship patterns include (A) → (B), A-[:coauthor]→B

11 objectivity DB

11.1 general

container-based architecture w/ physical identity

meta
OODBMS since 1995, v10, C++
java, c#, python, smalltalk, ... frontends
data replication, fault tolerance options
all platforms like windows, solaris, linux, mac
cloud computing in AWS
customers from all branches

ideal applications
store, process complex structures (trees, collections, graphs)
relationship hunting, protein structure, correlation analysis

client architecture
languages interfaces provide access to objects & schema
local storage / transaction cache
client objectivity server (data from local storage, remote processing)

server architecture
lock server which grants permissions
query server to run queries
data server which handles the memory

performance
clustering & multi-dimensional indexing
client side, cross-transactional caching

parallel query engine (PQE)
client side task splitter which can aim queries at specific dbs, containers
can split query to multiple agents for parallel processing

storage scopes
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
each scope/hierarchy can contain up to 2ˆ16 of lower level
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

11.2 c# persistence designer

used to update schema
generates c# objects, can generate federation files

11.3 persistent object model

basis types ("primitives")
complex types, embedded in parent or referenced (OID stored)
enumerations, collections, relationships

11.4 relationships

unary, binary, to-one, to-many
referential integrity maintained by system (incl. inversion)

storage
default, non-inline (array stores (identifier, OID))
inline (stored as fields on object, to-many stored in array)
binary associations as complete separate construct

propagation
deletion, locking can be propagated over relations
developer specifies how

versioning
when object is copied, specify what happens to relations
copy (new, old associated with same objects)
drop (copy does not have references set)
move (copy has the references set, original does not)

11.5 domain classes

partial classes from .NET to separate application/persistence code
persistence implemented in base class
author.cs contains public get;set to private props
authorpd generated with private get;set; which do persistence stuff
support class w/ schema class, attributes & its properties, proxy cache

11.6 connection architecture

static functions to startup(), open connection, shutdown()
one connection to federation per application
n sessions w/ cache, transaction state, has one, many threads
cache kept after commit, flushed on abort
if update too big, overflow pages prewritten to disk

interaction
connection - > create session → begin transaction, get federation
federation will lookup database, create new database

persist objects
give reference of db, container, other entity in constructor
they are connected, persisted automatically

persistent collections
sets, lists, maps as ordered, unordered and scalable, non-scalable variants

iterator
provides access to objects meeting certain criteria
scope is collection, container, db, federation
criteria is PQL predicate as a string
not efficient unless indexes preconstructed
lazy filtering, therefore no sorting

scope name
can name object, collection at each storage hierarchy (like roots)

retrieve objects
by scope name, link following, lookup with keys & iterators
parallel query with PQE
content-based filtering (supporting primitive types, group lookups)

retrieve objects by group
use object iterator for storage hierarchy, name maps, root names, name scope
use collection iterator for lists, sets, object maps

LINQ
language integrated queries, transforms query in method calls
objects (in memory), SQL (MSSQL), XML, DataSet (ADO.NET)
other providers possible like to db4o

12 Object Store

page-based architecture w/ physical identity
query may be executed on server

12.1 meta

personal edition
lightweight object database
large, single user database
multithreading, small memory footprint
for embedded systems, mobile computing, desktop applications

enterprise
distributed multi-user

object caching
for clustering, online backup, replication, high availability

12.2 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 (the PSR, see below)

persistent storage region (PSR)
PSR sits between heap & stack
acts as new layer of cache, data of client inside this area
serves as secondary storage, persistent over transactions
implemented with indirect pointer

page fault
if data accessed in PSR which is not there yet
ObjectStore maps from memory or server to location

address translations
done when data is fetched into cache
retranslation may occurs when PSR full

12.3 server side architecture

server
manages databases & transaction logs
serves content in pages
enforces ACID with permits
2PC with other servers
recovery mechanisms

database
binary files storing pages of c++ memory

transaction log
pending changes of transactions, on commit executed on db
for recovery, faster commits, multi-version concurrency control MVCC

12.4 client side architecture

client
C++ using ObjectStore library
pages automatically fetched from db according to demand

cache
one cache memory file per process with fixed size
contains pages fetched from the server (over transactions)

commseg
one commseg memory file per process describing the cache
stores permit, lock, meta data for all pages (kept between transactions)

cache manager
one process by client, shared by all clients on machine
handles permit revokes, read/write to cache & commseg

persistent storage region PSR
reserved area between heap/stack
logical pointer addresses of client map into this area
ranges with fetched pages, ranges with to-be-fetches pages
if (PSR full) then updated pages back to DB, read-only dropped
if (end of transaction) then PSR cleared to be reused

12.5 fetching/mapping pages

client fetches automatically, lazily
server permits & client locks acquired automatically

steps
ObjectStore override segment violation SIGSEGV handler
program obtains pointer p to page x
access to x causes exception, catched by SIGSEGV handler
page fetched from server, put into cache
execution continues

12.6 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

locks
read permits to take readlock (multiple clients simultaneously)
write permits to take writelock (only single client)
else client must ask server to escalate permissions

server call-backs
to revoke read permit (if server needs write)
to revoke write permit to read
ACK response if client has no lock taken
NACK response if client has lock, but needs to give up after transaction

allows high performance
data cached across transactions
fewer locks need to acquired
cached data in globally consistent state

12.7 distribution & heterogeneity

transactions can include objects from different db
different client platforms as runtime marshals

12.8 persistence

persistence by instantiation C++
orthogonal to types
single db can be used for both transient, persistent objects

allocation
overloaded new(allocation, type specification, count = 1)
locate transiently on heap
locate on database, segment, cluster, next to other object

12.9 transactions (ACID)

atomicity (commit saves changes, abort removes changes)
consistency (impossible to apply, lose data on write)
isolation (2PL serialisability, MVCC snapshots for read-only transactions)
durability (changes to transaction log, background processes write to db)

types
read (throws exception if write requested)
write
local (initiating thread can execute)
global (all threads in session can execute)
lexical (thread-local, start-end in same code block, automatic retry)
dynamic (lower level, for multi-threaded application)

12.10 database layout

clusters hold segments, saved as pages

segments
logical partitioning of object
0 for schema, db roots
2 default
4+ user created till 2ˆ32

clusters
group closely related objects
0 default cluster
1+ user created

12.11 developing applications

12.11.1 basic constructs

objectstore
runtime of ObjectStore
static initialize(), static shutdown()

os_database
database functionality, applies updates automatically
static create(), save(), close() (does not save state), destroy() (deletes)

os_transactions
transaction handles & functions, can be nested arbitrarily
static initialize(), static get_current(), use mixins block pre/postfix
abort(), commit()

os_typespec, os_ts
determine type specification
Author∗ = new(db, os_ts<Author>::get(), 1) Author("Stein")

os_database_roots
creation, retrieval, removal of roots

os_segment
segment management

os_cluster
cluster management
static of(object_reference) returns cluster of that objects

os_database_root
labelled persistent objects
create_root(), find_root(), get_value()

12.11.2 development process

writing of persistent classes, schema file, application logic
schema file compilations with pssg compiler
C++ compilation, then link with pssg binaries

12.12 relationships

reference in both objects using os_collections

12.13 os_collections

fit for traversal, manipulation, retrieval
set, bag, list, array, dictionary, also available as generics (templates)
new(db, os_Set<T>::get_os_typespec()), insert(), delete keyword
reusable cursor with T first(), T next(), boolean more()

queries
specify element type, query string, database
query can be c++ condition as string or regex
nested queries & basic function calls allowed

13 the OM data model

13.1 elements

multiple inheritance, instantiation, classification
collections, associations
cardinality, classification, evolution constraints

13.2 about

extended entity-relationship model for OO data management
distinguishes typing, classification (representation vs roles)
data represented as objects (attributes, methods)
multiple inheritance, instantiation, classification
collections (set of persons), binary collections (person $\leftrightarrow$ location)
integrity, classification, evolution constraints
data definition, manipulation & query language OML

13.3 typing & classification

clears up issues by separating the two concepts
less complex type graphs but still rich classification possible

typing
employee extends person (naive) vs collection concept (semantic grouping)
for representation, data format, operations, inheritance

classification
person, employee $\rightarrow$ semantic grouping $\rightarrow$ contact, address, name, jobinfos
for roles, semantic grouping & constrains (member types) in collections
to define relationships in collections (no embedding in objects)
to add/remove roles dynamically

13.4 vs state machines

view collection membership as state
assign methods when collection membership changes
react on events

13.5 OM data model layers

13.5.1 type layer

objects represented by multiple object types
object types define type units
dress operation to add type, strip to remove (dynamically)
multiple inheritance (multiple supertypes, developer handles conflicts)
multiple instantiation (can have unrelated types from different hierarchies)

supported values
base types without identity (string, int, uri)
object types (objects with identity)
structured types (structures without identity)
bulk types (collection of same member, no identity) (arrays)

type units
corresponds to type, no inherited fields, methods (traits)
has attributes with name, type, bulk (name, string, uni)

information unit
instances of type unit has corresponding information unit
contains the data of the type unit
browse information unit by casting to its type given object identifier
person(o01) = ("max"), private(o01) = (1234, Bern)

bulk types
uni (single), set, bag, sequence, ranking

dress, strip
can dynamically add/remove type instances
dress create information unit with default units and attaches
strip removes information unit and discards values

13.5.2 classification layer

defined based on types from type layer
defines semantic grouping, multiple classification, collections, associations

collections
concept for semantic grouping, associations to link objects together
membership constrained by type (multiple collections can have same constraints)
kinds & roles
ext(Persons) = { person(o1), person(o2) }

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 (in all source $\Rightarrow$ in target, arrows other way)

associations
source, domain collection $\Rightarrow$ relation collection $\Rightarrow$ target, range collection
relation collection is of the form (source(o1), target(o2))
cardinality constraints (0:* means ≥ 0, 1:* means >0)
behaviour represented same as normal collections (set, bag, etc)
ternary, attributed relations not supported

nested associations
domain, range collection can be relations too (for n-ary relationships)
ternary relationships decomposed into primary, secondary
allows uniform query constructs, clearer semantics of relations

kinds, roles
kind (fixed not-changable classification, person; professor; postgrad)
roles (change during entity livecycle, baby $\rightarrow$ adult $\rightarrow$ senoir; student)

constraints, structures
applicable to subcollection, superconnections
for cardinality (describe associations)
for evolution (govern object livecycle)

evolution
assume each collection has single root (is a kind), single maximal collection
kind is fixed classification in context of parent (senior of kind human)
therefore kind can only be striped if parent can be striped too

determine valid evolution
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 (trivial if C_1 is leaf node)
(2) let K be kinds(C_1) - kinds(C_2) then all roles(K) can't be in C_2

example valid evolution
x :: Postgrad $\rightarrow$ Lecturer
(1) trivial because Postgrad is leaf node
(2) K = {Postgrad, Person} - {Person}, no r $\in$ roles(Postgrad) in roles(Lecturer)

13.6 object model language (OML)

declarative, object-oriented, for object data model

OML data definition language
object, structured type definition
method definition, implementation
collection, association, constraint definition

OML data manipulation language
CRUD, dress & strip operations

OML query language
expressions, functions for base type values
operations to access properties, execute methods for objects
operations on collections based on collectional algebra

13.7 schema definition

create type contact { name : string, webpage : uri }
create type person subtype of contact { age : int }
method getWork() returns (location: set of location) ( )

13.8 classification definition

create collection Contacts as set of contact
create collection WorksFor as set of (person, organisation)
create constraint con1 association on WorksFor from Person(0,∗) to Organisation(0,∗)
create constraint con2 subcollection Person restricts Contacts
create constraint con3 classification (Person, Organisation) partition Contacts
create constraint con4 classification Person is kind

13.9 queries

applies to collections, binary collections

selection
retrieve person object matching predicate
all $p in persons having ($p.title = ”prof”)

map
retrieve table with attributes as columns
map $p in persons by ($p.title x $p.name)

extraction
specify first/last, top nth elements, max/min
the 3 in persons; first persons, max persons.getAge(),years

reduce
calculate aggregated value
reduce $p in persons aggregate $a by ($p.age + $a) default 0

domain, range (binary)
get left (domain), right (range) part of binary collection
domain locatedAt, range locatedAt

domain/range restriction (binary)
retrieve associations which match predicate
locatedAt dr (all $p in persons having (p$.title = ”prof”))

domain/range subtraction (binary)
contrary of restriction (takes other results)
locatedAt rr (all $p in persons having (p$.title = ”prof”))

inverse (binary)
swap domain/range of collection
inverse locatedAt

nest (binary)
groups range of each domain in set
nest locatedAt

compose (multiple binary)
join range of first with domain of second
situatedAt compose inCountry

closure (binary)
compose binary with itself (for hierarchic relations)
closure partOf

division (binary)
divide binary by collection, keep domain/range in all results
situatedAt div cities

13.10 collection algebra

union U, intersection N, difference -, selection %, map alpha, reduce o+, flatten +-
extend standard collection algebra for bags, sequences, rankings

bags
as tuples, <a,a,b> → {(a,2), (b, 2)}
union by taking max(n1, n2) of both sets <a,a> U <a, b> = {(a,2), (b,1)}
addition by taking sum(n1,n2)
intersection by taking min(n1, n2)
B1 - B2 by taking n1 if not in B2, or n1-n2 if in both
B % P = {(x, n) | (x,n) ∈ B ˆ P(x) = true} for P(x) = x → bool
B alpha f = o+ B((x,n), B1) → ({(f(x), n) U B1}) for f(x) = x → y
o+ B f v = f(x, o+ B’ f v) where B = B’ union {(x, 1)} for f(x) = x1, x2 → y
+- B = o+ B(x, B1) → (ext(x) U B1) (flattens 3D to 2D)

14 storage and indexing

type hierarchy indexing
aggregation path indexing
collection operations

14.1 motivation

manage large data set on persistent storage

more requirements than relational
structuring, clustering, management of complex objects
access through references, query predicates
type inheritance hierarchies
relationships
multi-values properties & collections

14.2 object oriented storage

layouts not very different from relational systems

new algorithms
data structures for complex objects
grouping/clustering of complex objects
grouping/clustering of references
management of free space / buffer

14.3 storage model

value structures
consisting of data sets (records, attributes, domains)
divided in pages (disk block, sequence of disk blocks)
functions to map records to pages

access structures
query/update algorithms
search data structures

14.4 terminology

query
point, range query (exact value, interval matching)
single class / hierarchy (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)

sequential organisation
pages organised by data set, new entries are placed into newest page
queries have to traverse entire data set
data index maps ids to physical locations

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

point mapping
data set directly mapped to specific place in memory
query by using hash functions
directory with all points references linked pages

14.5 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 if the need is up for it

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

inverted files
search structure called vocabulary (indexes distinctive keywords)
inverted list for each keyword, storing id of records with that keyword
sort inverted list to apply compression

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

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

H-tree
fractal built up tree

others
hB-tree, Quadtree, TV-tree, cell tree

14.6 type hierarchy indexing

when using 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

h-tree
skipped, explanation not clear at all

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 combine 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

14.7 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 paths, possibly starting/ending at 0/n
answer queries starting at end points efficiently

nested index (NX)
backwards traversal of the full path
equivalent to backwards traversal of ASR can-extension

path index (PX)
backwards traversal of right-complete ASR

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

14.8 collection operations

muli-valued attributed evaluation

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

15 version models

15.1 domains

temporal databases
computer aided design & manufacturing
software configuration / engineering

15.2 basic aspects

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
space vs performance tradeof

operations
create / branch / merge / delete
similar to transactions (long running, nested)

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

queries
active versions guide to query sequential versions
main derivation guides to query parallel versions
may combine for generic reference

15.3 temporal databases

first application of version models

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

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

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

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 atttibuted 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)

15.4 engineering databases

for computer aided design/manufacturing CAD/CAM
for dev/maintenance of objects

requirements
handle complex, hierarchical object structures
support versioning through incremental development / trial and error

dimensions
linear revisions
(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)

15.5 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
objects can be specified at different granularities, representations
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)