ObjectStore

- Object-oriented DBMS
- Some different packages (C++, Java)
- C++ package
  - Closely integrated with the C++ language
  - Persistent storage capabilities for C++ objects
  - Associative queries
  - Transaction management
  - Distributed data access

Motivation

- Target applications (CAD, CAE, GIS…)
  - Complex manipulations
  - Large databases of objects with intricate structure
- Impedance mismatch between application code and database code
- A uniform programmatic interface to both persistent and transient data.

Goal: add persistence to C++

- Ease of learning:
  - C++ plus a little extra.
- No translation code:
  - Persistent data is treated like transient data.
- Expressive power:
  - General purpose language (as opposed to SQL)
- Reusability:
  - Same code can operate on persistent or transient data
- Ease of conversion:
  - Data operations are syntactically the same for persistent and transient data.

Goal: add persistence to C++

- Type checking:
  - The same static type-checking from C++ works for persistent data.
- Temporal/Spatial locality:
  - Take advantage of common access patterns.
- Fine interleaving:
  - Low overhead to allow frequent, small database operations
- Performance:
  - Do it all with good performance compared to RDBMSs
Discussion #1

- What are the pros and cons of merging programming languages & databases?
  - For example:
    - "Expressive power": You can express more queries using a programming language as compared to, say, SQL. What are the pros and cons? Are there alternate solutions?
    - "Reusability": Does the data model become more or less reusable across applications?
    - "Using the data": Does manipulating the data in the application become easier or difficult?
    - Other?

Application Interface

- Three programming interfaces
  - OC library interface
  - C++ library interface
- Extended C++ language
  - Collection facility
  - Relationship facility
  - Accessing persistent data
  - Query facility

Collection facility

- Object class library
  - Ordered collections (os_list)
  - Collections with or without duplicates (os_bag or os_set)
- Behaviors
  - insert(e), remove(e), create(e), …
- Looping construct (Cursor iterator)

Collection facility (cont.)

Relationship facility

- Modeling complex objects
- A pair of inverse pointers
- Maintaining the integrity of the pointers
- Relationships
  - One-to-one
  - One-to-many
  - Many-to-many

Relationship facility (cont.)
Accessing Persistent Data

main()
{
    database *db = database::open("/company/records");
    persistent<db> department* engineering_department;
    transaction::begin();
    employee *emp = new(db) employee("Fred");
    engineering_department->add_employee(emp);
    emp->salary = 1000;
    transaction::commit();
}

Accessing Persistent Data (cont.)

● Manipulation of persistent data like an ordinary C++ program
● Protecting the integrity of database
   ○ Automatically set read and write locks
   ○ Keep track of what has been modified
   ○ Access to persistent data guaranteed to be transaction-consistent, and recoverable

Discussion #2

○ ObjectStore employs page level locking as the only mode of locking
   ○ What implications does it have for transactions and concurrency?
   ○ Should other granularities of locking be provided as well? If yes, which ones?

Query Facility

● Closely integrated with the host language
   ○ Expressions operating on collections
   ○ Producing a collection or a reference to an object
● Selection predicates can be applied to collections.
   ○ Special syntax: [: predicate :]
   ○ Eg.
     employees [: salary >= 10000 :]

Query Facility (cont.)

○ Queries may be nested to form more complex queries

os_Set<employee*> &work_with_fred =
   all_employees : query ("employee*",
   "dept \ employees [: name == \Fred\ :]");

Memory-mapped Architecture

● Goal: object-access speed for persistent data equal to that of an in-memory dereference of a pointer to transient data
● Once objects have been retrieved, subsequent references should be as fast as an ordinary pointer dereference
● Similar goals as a virtual memory system—use VM system in OS for solution:
   ○ Set flags so that accessing a non-fetched persistent object causes page fault
   ○ Upon fault, retrieve object
   ○ Subsequent access is a normal pointer dereference
Distributed Data Access

- Client/Server communication method
  - Local area network
  - Shared memory, local sockets
- During transaction
  - Whole pages of data brought from server to client
  - Placed in the client's cache
  - Mapped into virtual memory
  - Objects stored on the server in the same format
- Transaction finish
  - All the pages removed from the address space
  - Modified pages written back to server

Distributed Data Access (cont.)

- Applications control the placement of objects within databases
  - Cluster objects that are frequently referenced together
- Objects can cross page boundaries
  - Ex. Image data
  - Page-granularity transfer
- Many small objects can reside on a single page
  - Locking granularity on a per-page basis
  - Clustering → decreasing locking overhead

Query optimizations

Some RDBMS query optimization techniques don't work or make sense

- Collections are not known by name
- Join optimization is less of a problem
  - Paths can be viewed as precomputed joins
  - Optimization is index selection
  - "true joins" are rare
- Index maintenance is more of a problem
  - Data members (indexable) → potential index keys

Conclusions

- ObjectStore provides the applications
  - High productivity
  - High performance
- Achieved by a virtual memory-mapping architecture
- Support for conceptual modeling constructs by collection, relationship, and query facilities