The ObjectStore Database System

Charles Lamb
Gordon Landis
Jack Orenstein
Dan Weinreb
Slides based on those by Clint Morgan

Overall Problem

- Impedance mismatch between application code and database code (eg, C++ and SQL)
- Need a programmatic interface to both persistent and transient data.

Motivations: add persistence to C++ (1/2)

- Ease of learning: C++ plus a little extra.
- No translation code: persistent data is treated like transient data.
- Expressive power: general purpose language (as apposed 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.

Motivations: add persistence to C++ (2/2)

- Type checking: 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

~5 minutes

Discussion in pairs

Assumption: Everyone here has used an OO programming language and has used a relational database with it. Try to draw from your experience to answer this question.

The following goals are given as motivation for ObjectStore:

- No translation between query and program code
- Reusability
- Ease of use
- Expressiveness

- Do you feel that ObjectStore satisfies all of these goals better than a RDBMS? Explain.
- Is language mismatch a problem you encountered?

Application Interface

- Three programming interfaces: libraries for C and C++, and an extended C++ language. We focus on language extension.
- Keyword **persistent**. Used when declaring variables
- A few other keywords (**inverse_member**, **indexable**) for defining how objects in the DB relate.

```
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();
}
```

Collections

- Similar to arrays in PL or tables in DBMSs
- Allow performance tuning: developers specify access patterns and an appropriate data structure is chosen
- Similar to using collection interfaces in modern libraries (Java, C#)
- Elements may be selected from collections with queries (more on this to come).

Relationships

(this can be skimmed or skipped as needed)

- Pairs of inverse pointers which are maintained by the system.
- One-to-one, one-to-many, and many-to-many are supported.
- Syntactically, relationships are C++ data members, however, updating causes its inverse to be updated.
- How does this work for the library interface?

Queries

- Selection predicates can be applied to collections.
- Special syntax: `[: predicate :]`
- Eg.

```
employees [: salary >= 10000 :]
```

- Queries may be nested…. But no real joins; only semijoins (i.e., can find which tuples match other tuples, but not say what matches what). Example from Wikipedia:

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Production

Dept

DeptName	Manager
Sales	Bob
Sales	Thomas
Production	Katie
Production	Mark

Employee ⋉ ***Dept***

Name	EmpId	DeptName
Sally	2241	Sales
Harriet	2202	Production

Accessing persistent data

- Overhead is a major concern.
- 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

Query optimizations

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

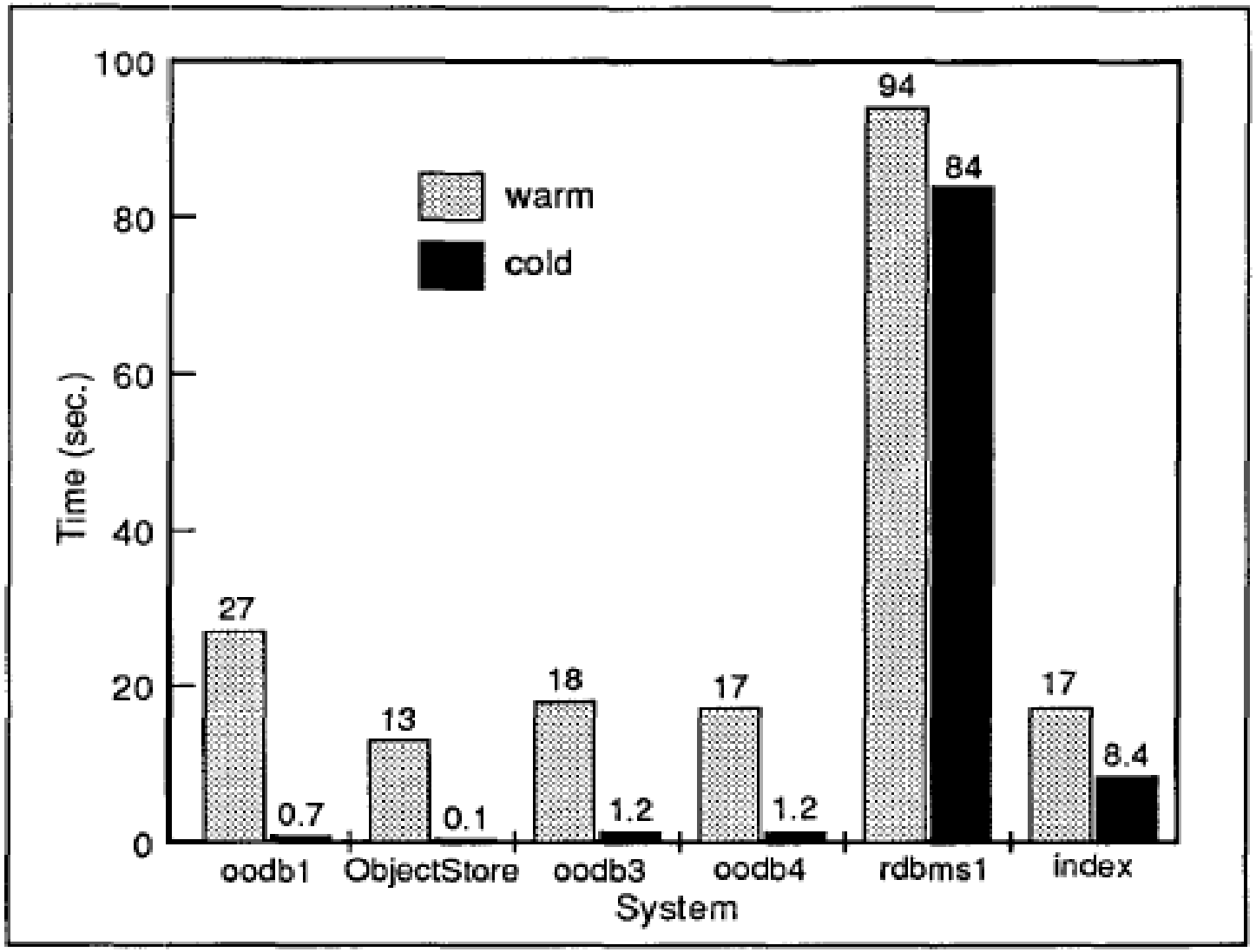
- Collections are not known by name
- Join optimization is less of a problem
  - paths can be viewed as precomputed joins
  - optimization is index selection
  - "true joins" are rare… or at least not supported
- Index maintenance is more of a problem

7-10 minutes

Discussion in groups of 4-5

- Do you feel the following features are limiting or improving the usefulness of OODBs:
  - Tied to C++ (or other PLs).
  - Pre-computed joins (references).
  - Caching commonly used variables.
- Can you think of an application that is better suited for OODBMSs than RDBMSs? Would the above features help or hinder the development of such an application?

How caching helps (note: bars are backwards)



Conclusion

- Performance experiments show caching and virtual memory-mapping architecture work.
- Small case study shows productivity benefits
- ObjectStore provides
  - Ease of use
  - Expressive power
  - Tight integration with host environment
  - High performance due to VM mapping architecture

Of Objects and Databases: A Decade of Turmoil

Carey, M.J.; DeWitt, D.J.
(1996)

Slides based on slides by Ricardo Pedrosa

Objects and Databases. Major types of systems:

- Extended relational database systems.
- Persistent programming languages.
- Object-oriented database systems.
- Database system toolkits/components.

Extended relational database systems

- Allow the addition of new, user-defined abstract data types (ADTs).
  - ADTs are implemented in an external language.
  - After being registered with the database, ADT's functions can be used in queries.
- Projects:
  - Ingres
  - Postgres
    - Query optimizers with ADT's properties and functions awareness.
    - Support for storing and querying complex data types.

Persistent Programming Languages

- Add data persistency and atomic program execution to traditional object-oriented programming languages.
- Problems addressed:
  - Orthogonality.
  - Persistence models.
  - Binding and namespace management for persistent roots.
  - Type systems and type safety.
  - Alternative implementation techniques for supporting transparent navigation, maintenance, and garbage collection of persistent data structures.
- Projects: various PL vendors

Object-Oriented Database Systems

- Combine all of the features of a modern database system with those of an object-oriented programming language, yielding an object-oriented database (OODB) system.
- Focused on:
  - Support for querying, indexing and navigation.
  - Addressing version management needs of engineering apps.
- Projects:
  - Gemstone (Smalltalk).
  - Vbase (CLU-like language).
  - Orion (CLOS).

Database system toolkits/components

- Provide a DBMS that can be extended at almost any level, using mostly kernel facilities plus additional tools that help building domain-appropriate DBMS.
- Projects:
  - EXODUS.
    - Storage manager for objects
    - E: a persistent Prog. Language.
    - Query optimizer generator.
  - Starburst.
    - Clean architectural model that facilitates storage and indexing extensions.
    - Rule-based extensible query subsystem.

What happened?

- System toolkits & persistent programming languages:
  - In spite of some interesting results these were a failure from a commercial point of view.
- OO database systems:
  - Many results from the academic point of view. Not expanded commercially as expected by its developers.
- Language-specific object wrappers for relational databases:
  - New approach that appears to be important for building OO, client side apps.
- Extended relational DBS:
  - Renamed as Object-Relational DBMS. Appears to be settling in terms of providing objects for enterprise DB apps.

The Database Toolkit approach problem.

- Require a lot of expertise.
- End up in being inflexible, awkward or incomplete.
- As OO and O-Relational database systems provide enough extensibility, it's not worthy to start from scratch even given a toolkit to help in the process.

Why EXODUS failed?

- The client/server architecture introduced an unwanted level of indirection when users tried to use EXODUS to implement their own object servers.
- E programming language: Too general for skilled database implementors and too low-level for application-oriented programmers.
- The query optimizer was inefficient and hard to use.

Was all that bad after all?

- Interesting research by-products relevant to OODBMS and ORDBMS.

Discussion in pairs

~5 Minutes

- Are you surprised by the death of Exodus? Why/why not?
- Do you think Starburst faced the same challenge? Would you classify it as a toolkit or extended RDBMSs? What about Volcano?

Persistent Programming Language

- No commercial implementation of such a language.
- Still active as a research area in academia.
- Work on this area has had a significant impact and has been transferred to OODBMS.
  - Navigational programming interfaces.
  - Persistence models.
  - Garbage collection schemes for persistent data.

What must OODBMS support?

- Complex objects.
- Object identity.
- Encapsulation.
- Inheritance and substitutability.
- Late binding.
- Computationally complete methods.
- Extensible type system.
- Persistence.
- Secondary storage management.
- Concurrency control.
- Recovery.
- Ad hoc queries.

What might OODBMS support?

- Multiple (vs. single) inheritance.
- Static (vs. dynamic) type checking.
- Distribution.
- Long transactions.
- Version management.

Optional issues.

- Programming paradigm.
- Exact details of the type system.
- Degree of fanciness of the type system.
- Degree of uniformity of the object model.

Players on this game.

- ObjectStore
- Ontos
- O2
- Versant
- Objectivity
- GemStone
- Poet

What went wrong with OODMS?

- Lack of standards.
- OODBMS products are behind RDBMS in some terms (eg. no view facilities).
- Painful schema evolution.
- Tight coupling between an OODBMS and its application programming Language.
- Low availability of application development tools.

Main tenets for ORDBMS

- Provide support for richer object structures.
- Subsume RDBMS.
- Be open to other subsystems (tools and multidatabase middleware products).

What ORDBMS should provide?

- A rich type system, inheritance, functions and encapsulation, optional unique ids and rules/triggers.
- A high-level query based interface, stored and virtual collections, updatable views and separation of data model and performance features.
- Accessibility from multiple languages, layered persistence-oriented language bindings, SQL support and a query-shipping client/server interface.

Fully integrated solution

Object relational servers will provide:

- Support for OO ADTs. (not fully)
  - Inheritance among ADTs.
  - ADT implementation in various programming languages.
- Full OO support for row types. (no)
- Support for middle-tier and desktops applications. (no)
  - Provide a development environment where the same object model will describe the DB in all levels, both for querying and navigational programming.
- Methods and queries will be run on cached data on servers or clients depending on where's faster. (no)

- OODBMSs will be niche solutions (yes, modulo XML)

Research Challenges

- Server functionality and performance
- Client integration
- Parallelization
- Legacy data sources
- Standards

Discussion in groups of 4-5

7-10 Minutes

The authors list a set of predictions for 2006 and how things are going to look then for objects and databases. The list again:

- Support for ADTs with inheritance
- Full OO support for row types.
- The same object model will describe the DB in all levels.
- Intelligent use of cache on servers or clients.

- Do you feel these predictions were reasonable given the state of the research presented in the paper? Explain.
- Any factors you believe the authors failed to consider?
- Would anyone like to add to these predictions? Perhaps something you noticed becoming a trend of late that can be fulfilled by 2026?