

Java Universal Binding

Storing Java Objects in Relational and Object-Oriented
Databases

Florian Xhumari

Cassio Souza dos Santos

Marcin Skubiszewski

INRIA Rocquencourt

O₂ Technology

Introduction

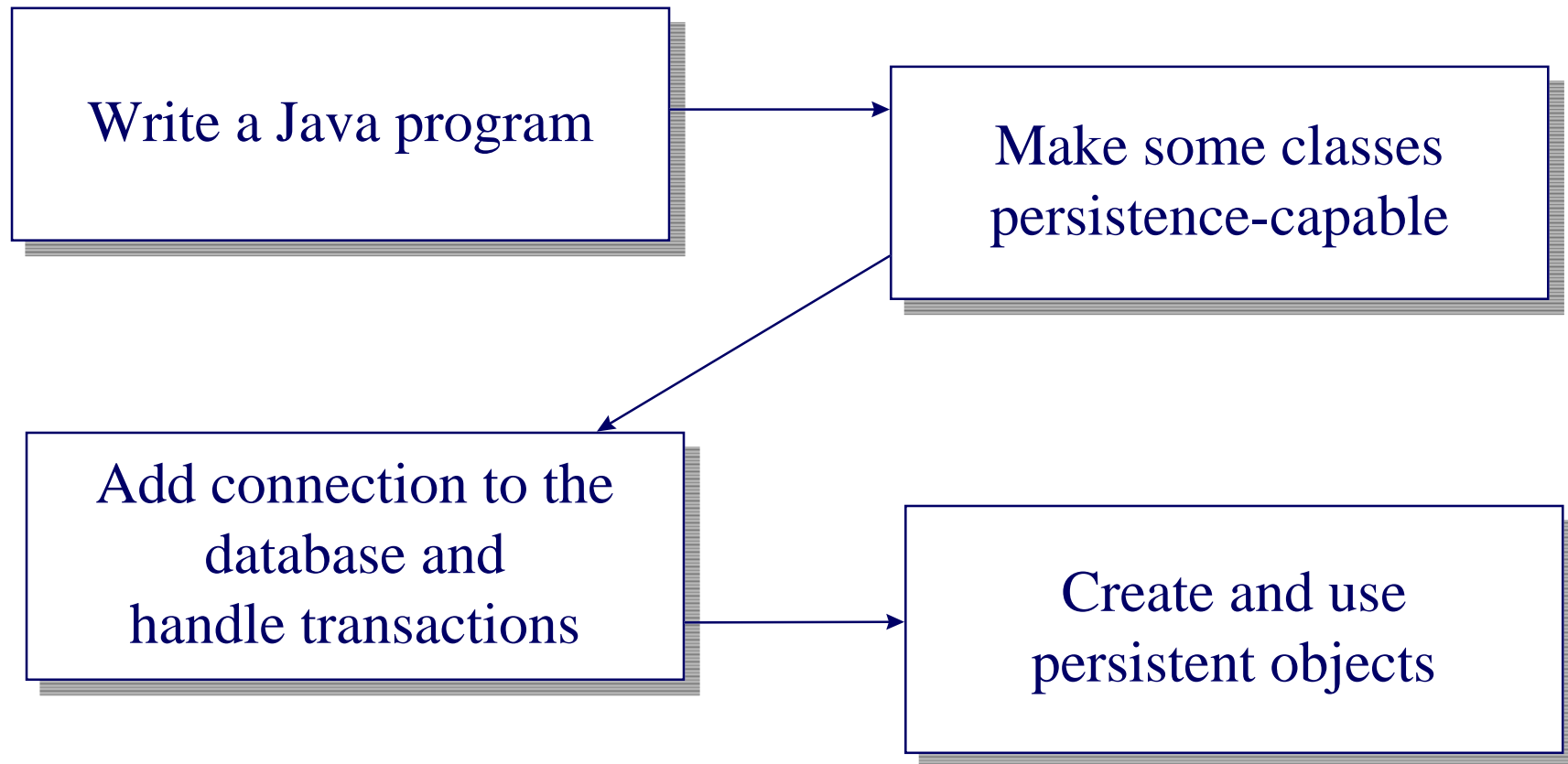
- ◆ Java Universal Binding (JUB)
 - Access databases from Java
 - Persistent Java implementation
- ◆ Existing approaches
 - JDBC drivers (most relational database systems)
 - » Execution of SQL statements and retrieval of the results
 - Language bindings (most OODB systems)
 - » ODMG Java binding
 - Extended JVM or compiler, support from the OS
 - » Java as persistent language

Our goals

- ◆ Uniform access to different databases
 - Relational systems: Oracle, Sybase or using JDBC
 - Object-Oriented databases: O₂
- ◆ Easy addition of new database systems
- ◆ ODMG-compliant Java binding

A Programmer's View on JUB

Writing a program using JUB



Persistence-capable types

- ◆ User-defined classes
 - Made persistence-capable by the *import* tool
- ◆ Built-in types
 - Considered persistence-capable
- ◆ Arrays
 - Persistence-capable iff the element type is so

The import tool

- ◆ Creates the necessary structures in the database
 - Tables in relational databases
 - Classes in O_2
- ◆ Modifies the imported class
 - Adds inheritance from an interface (`PersistentObject`)
 - Generates methods which implement this interface
- ◆ Creates a peer helper class (`MetaData`)

Making objects persistent

- ◆ Support for
 - Persistence by attachment
 - Explicit persistence
- ◆ Persistent roots: class static variables (optional)
- ◆ Class extents (optional)
 - Relational: using tables
 - O₂: automatic maintenance of collections

Using persistent objects

- ◆ Transparent use of persistent objects
 - Code automatically modified for notifying access:
 - » Methods `access()` and `markModify()`
 - » Inserted in the code by the import tool
- ◆ Execution of queries on class extents
 - SQL query for relational databases
 - OQL query for O_2
 - » Investigating the possibility to use Java methods in the OQL query

Using persistent objects (continued)

◆ Transient fields

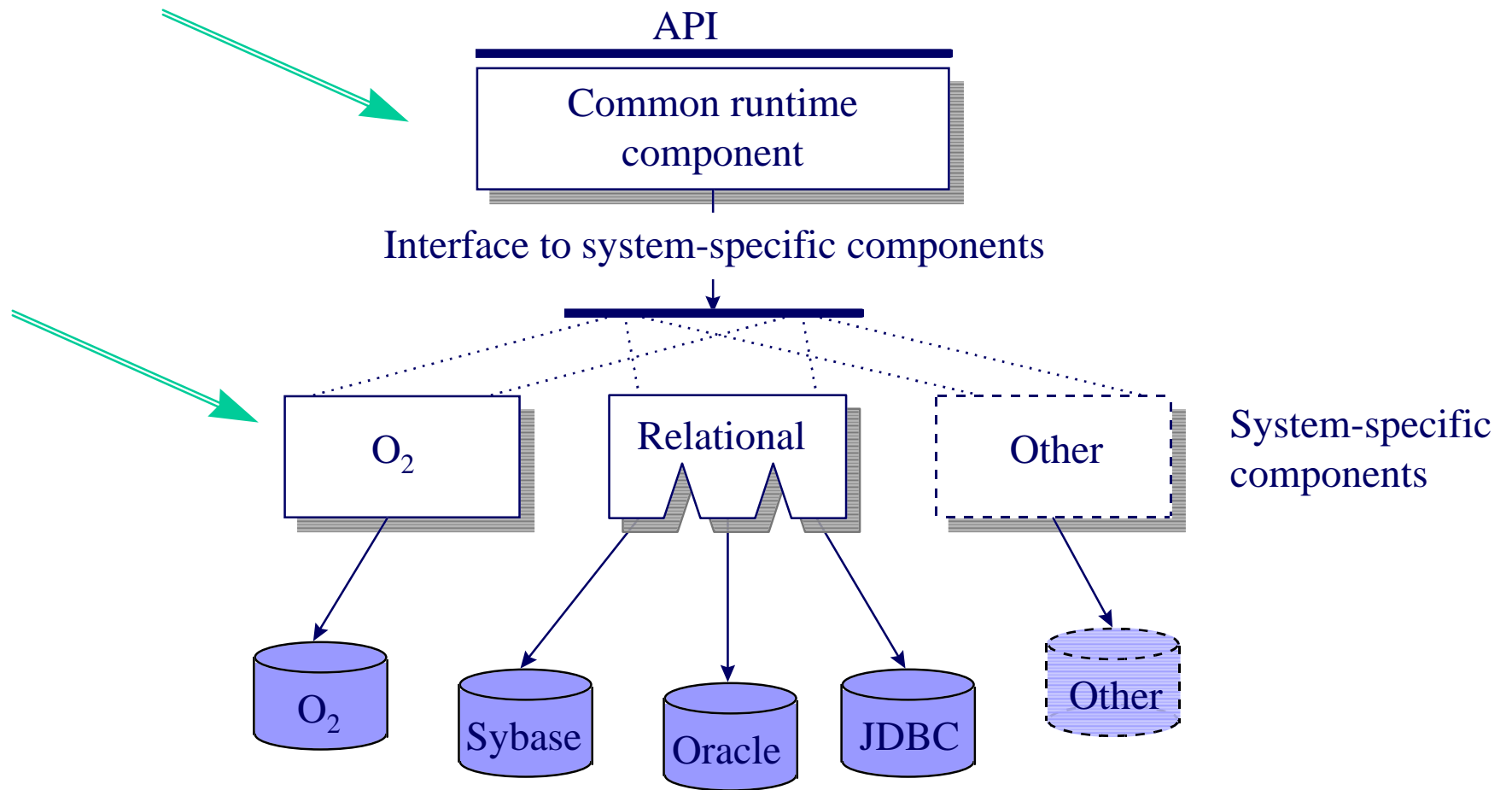
- Useful in some cases
 - » A field points to a transient object
 - » A field serves as a cache for the object
- Method `activate()` called when the object is loaded
 - » Used to initialize transient fields
- Method `prepareToWrite()` called before saving the object
 - » Used to update the persistent fields from transient fields

Storing Java bytecode

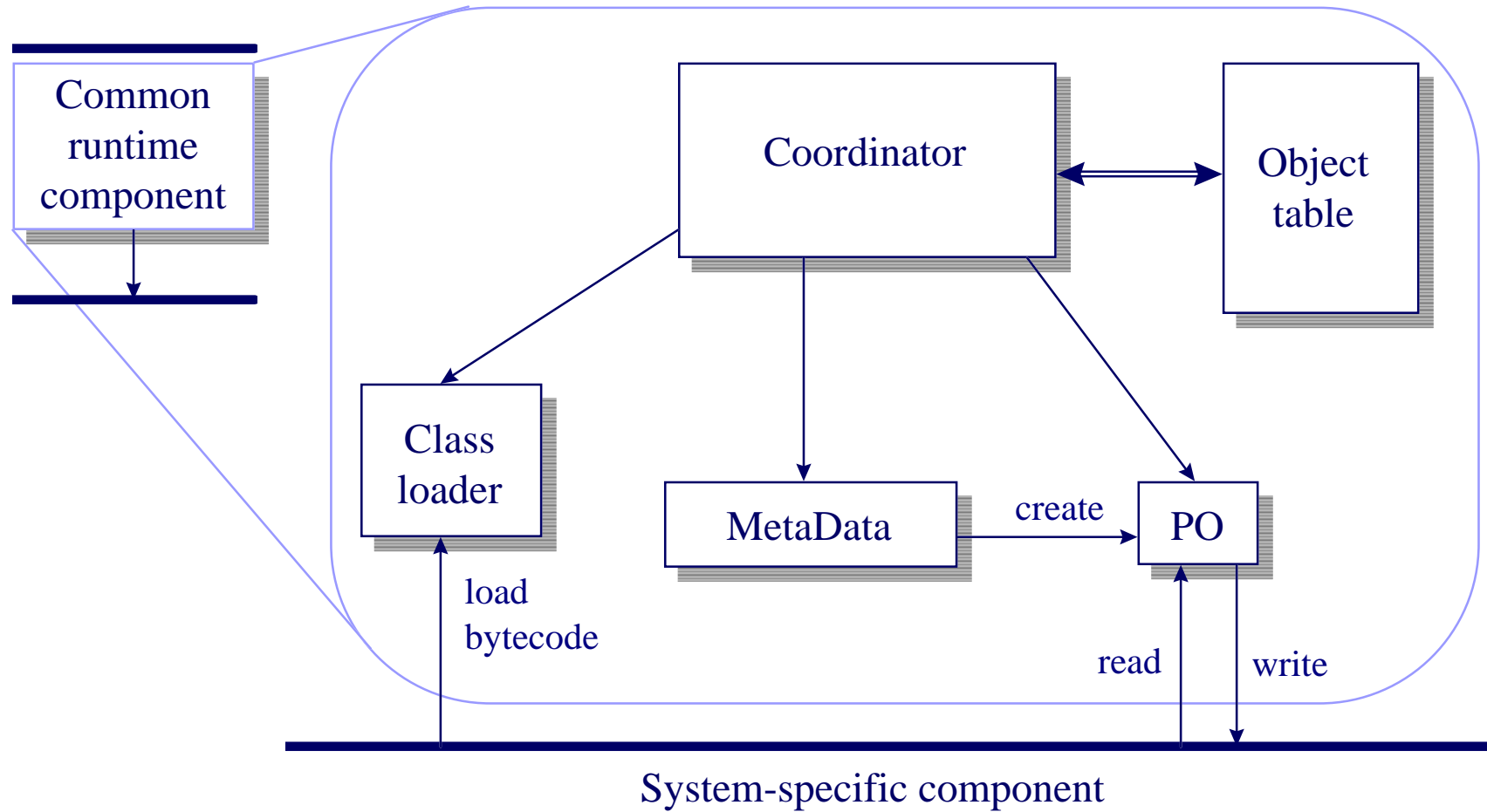
- ◆ The bytecode of persistent classes may be stored in the database
- ◆ Classes are loaded from the bytecode repository, if not found in CLASSPATH
- ◆ Implemented via a subclass of `ClassLoader`
 - Similar to loading classes from the network

The Architecture of the Runtime System

General structure



Common runtime component



Common runtime component (cont.)

- ◆ The object table

- Maintains the correspondence

Persistent object in Java \leftrightarrow Object identifier in the database

- » The object table references every persistent object loaded

- ◆ MetaData class

- Generated by the import tool for each imported class

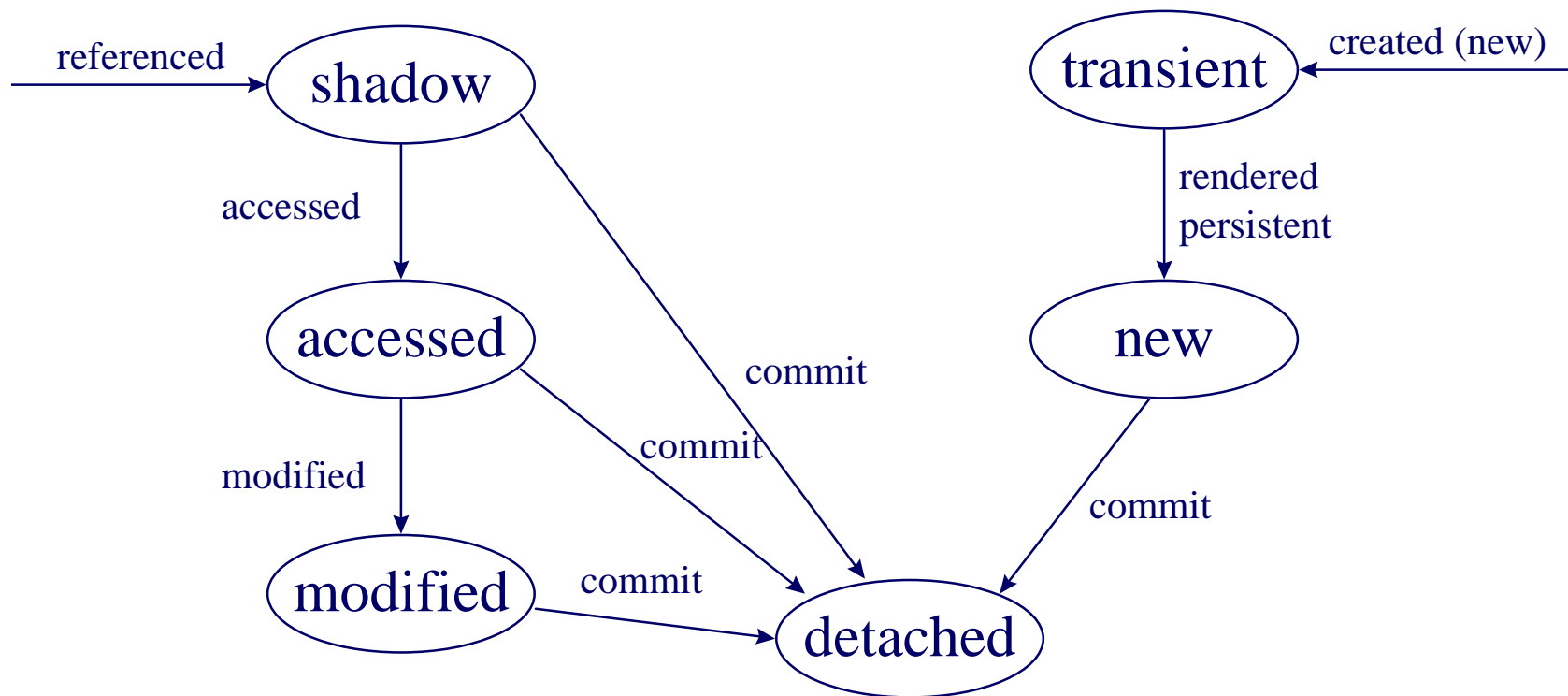
- Contains information about the class

- » Name, parent, interfaces, names of fields, their types

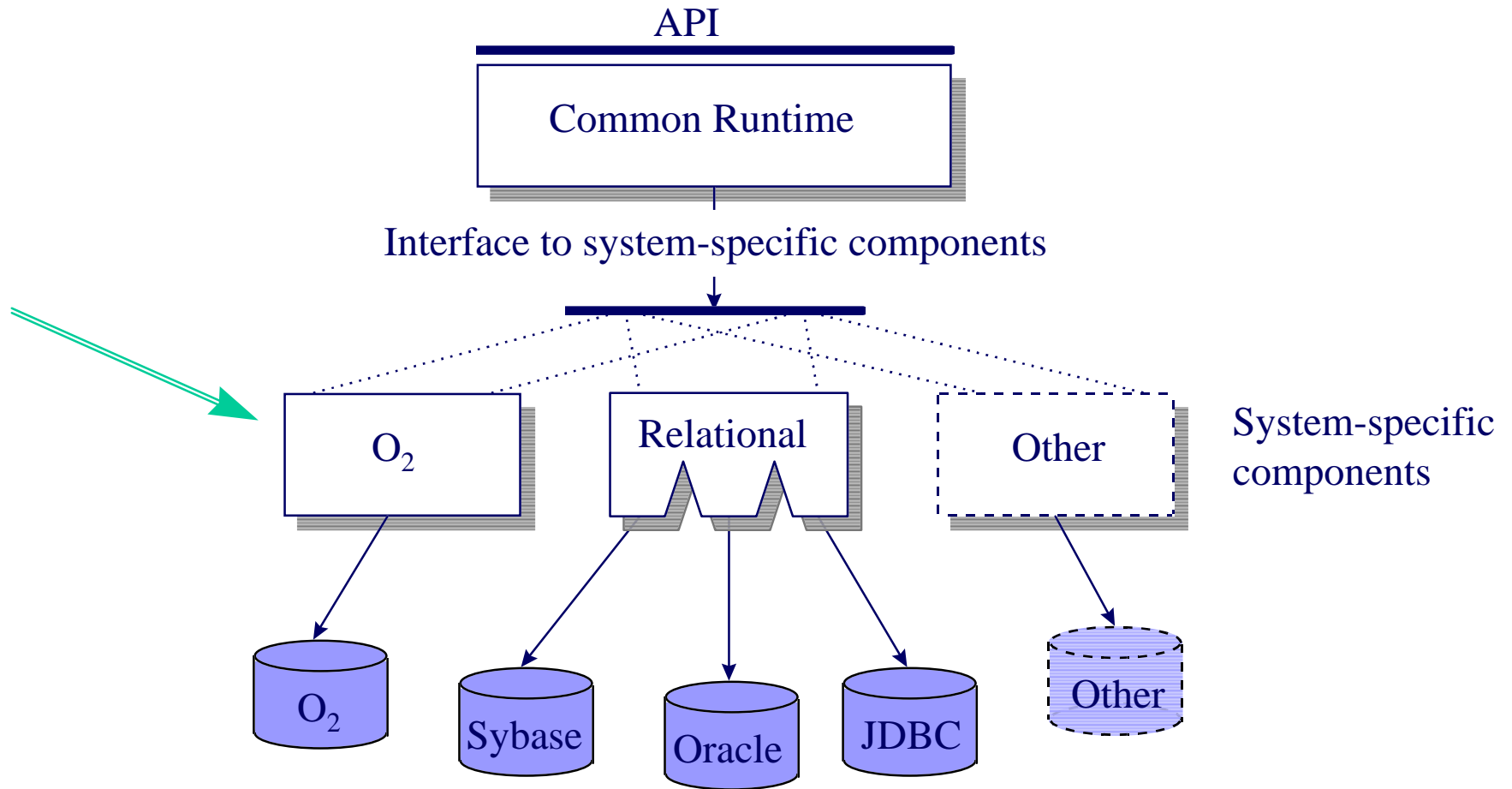
- » A method to create uninitialized instances of the class

Lifecycle of a persistent object

◆ State graph



O₂-specific component



The data model of O₂

◆ Values

- Simple: `integer`, `real`, `char`, `string`
- Complex: `tuple(...)`, `list(...)`

◆ Classes and objects

- `class A type tuple(...)`

◆ Inheritance

- Multiple inheritance
- Inheritance between complex values
 - » If B inherits from A, then `list(B)` inherits from `list(A)`

Mapping of Java types to O₂

- ◆ General rule: use native O₂ types
 - Allows access to databases from different languages
 - » Java, C++, O₂C, Smalltalk
- ◆ Scalar types
 - Mapped to the smallest O₂ type large enough
 - Exception raised on overflow
 - Special case: long (64 bit) coded as two successive integers

Mapping of Java types to O₂ (cont.)

◆ Strings

- Mapped to O₂ string values
- Strings are objects in Java but values in O₂
 - » Strings lose their identity when loaded from the database
- Reasons of this choice
 - » Performance considerations
 - » Strings are immutable objects
 - » Programs rarely compare pointers to strings

Mapping of Java types to O₂ (cont.)

◆ Classes and interfaces

| Java | O ₂ |
|--------------------------------|--------------------------------------|
| <code>class A {...}</code> | <code>class A type tuple(...)</code> |
| <code>interface I {...}</code> | <code>class I type;</code> |

◆ Inheritance

| | |
|---------------------------------------|------------------------------------|
| <code>class B extends A ...</code> | <code>class B inherit A ...</code> |
| <code>class C implements I ...</code> | <code>class C inherit I ...</code> |

◆ Java attributes are mapped to O₂ fields

Mapping of Java types to O₂ (cont.)

◆ Arrays

- Mapped to classes of type list

A[] class o2List_A type list(A)

A[][] class o2List_o2List_A type list(o2List_A)

- Inheritance is preserved

B[] class o2List_B inherit o2List_A type list(B)

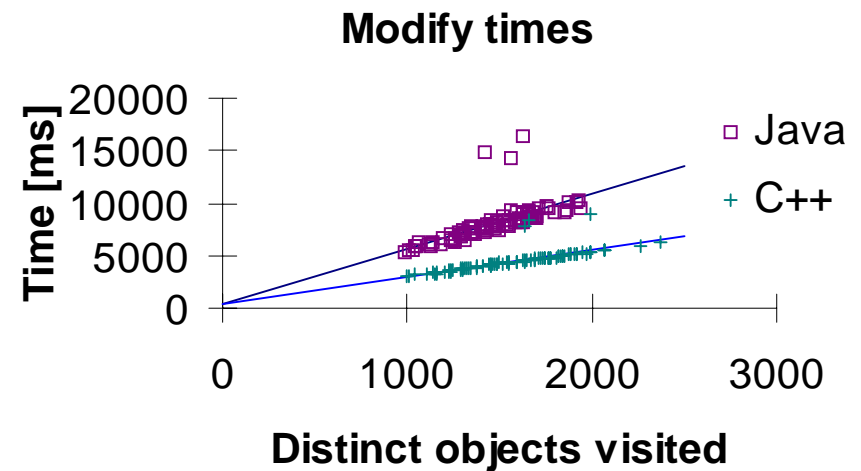
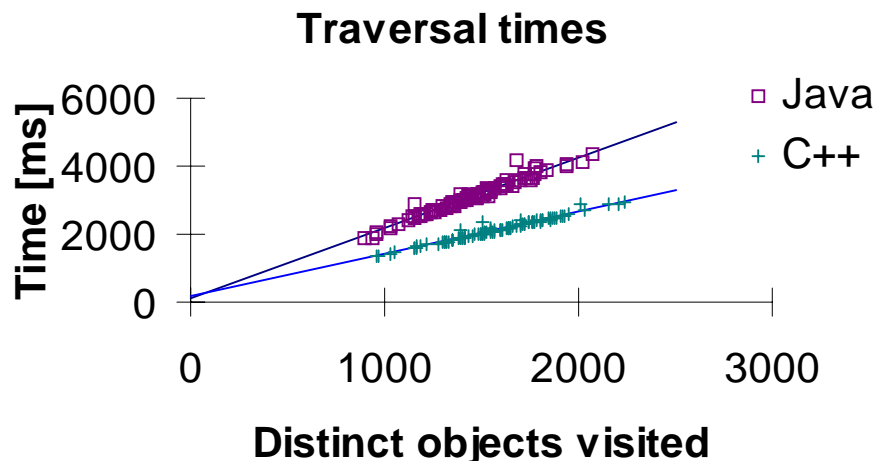
- New persistent arrays are filled with nulls

Performance Measurements

The benchmark

- ◆ Based on OO1 benchmark
 - Same database schema
 - » A graph of `Part`s (each `Part` is connected to three others)
 - Experiments: *traverse*, *create*, *modify*
 - » Added the experiment *modify*: same as *traverse*, but modify objects while traversing
- ◆ Measured time for
 - cold and warm runs
 - small (4Mb) and big (40 Mb) databases

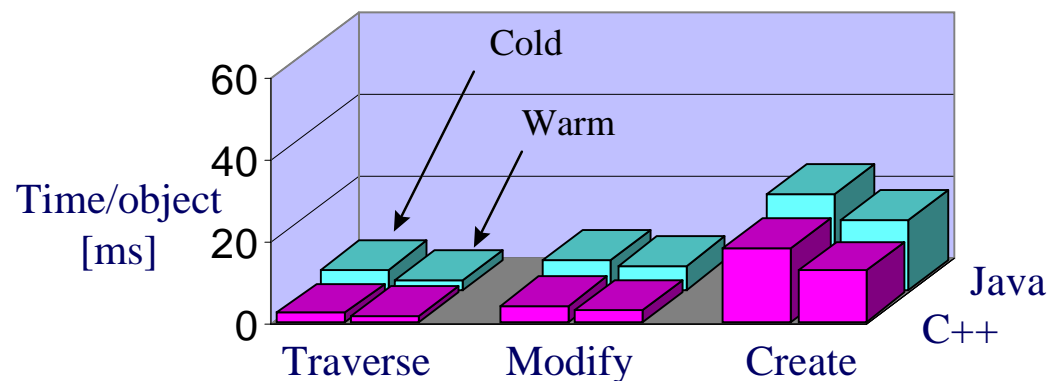
Benchmark results



- ◆ We make 3280 visits, but we access less objects (some objects are visited more than once)
- ◆ The execution time is proportional to the number of distinct objects visited

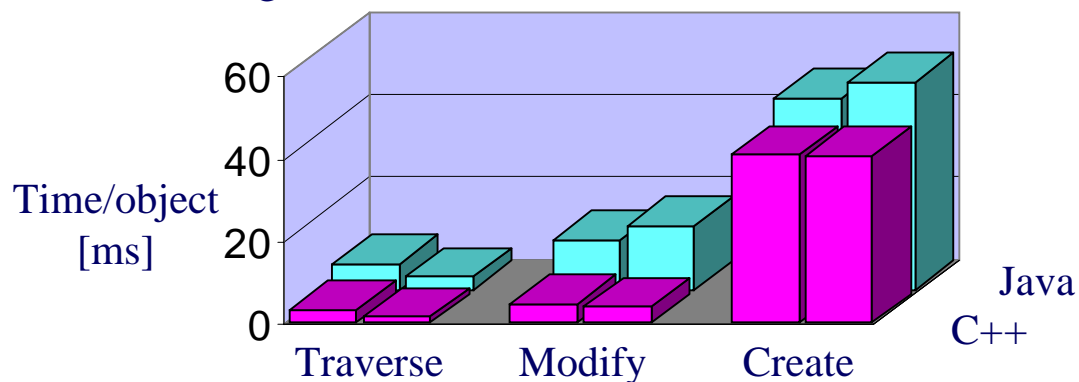
Benchmark results (continued)

◆ Small database



- ◆ C++ faster than Java
- ◆ Reasonable performance
 - We used an interpreting JVM

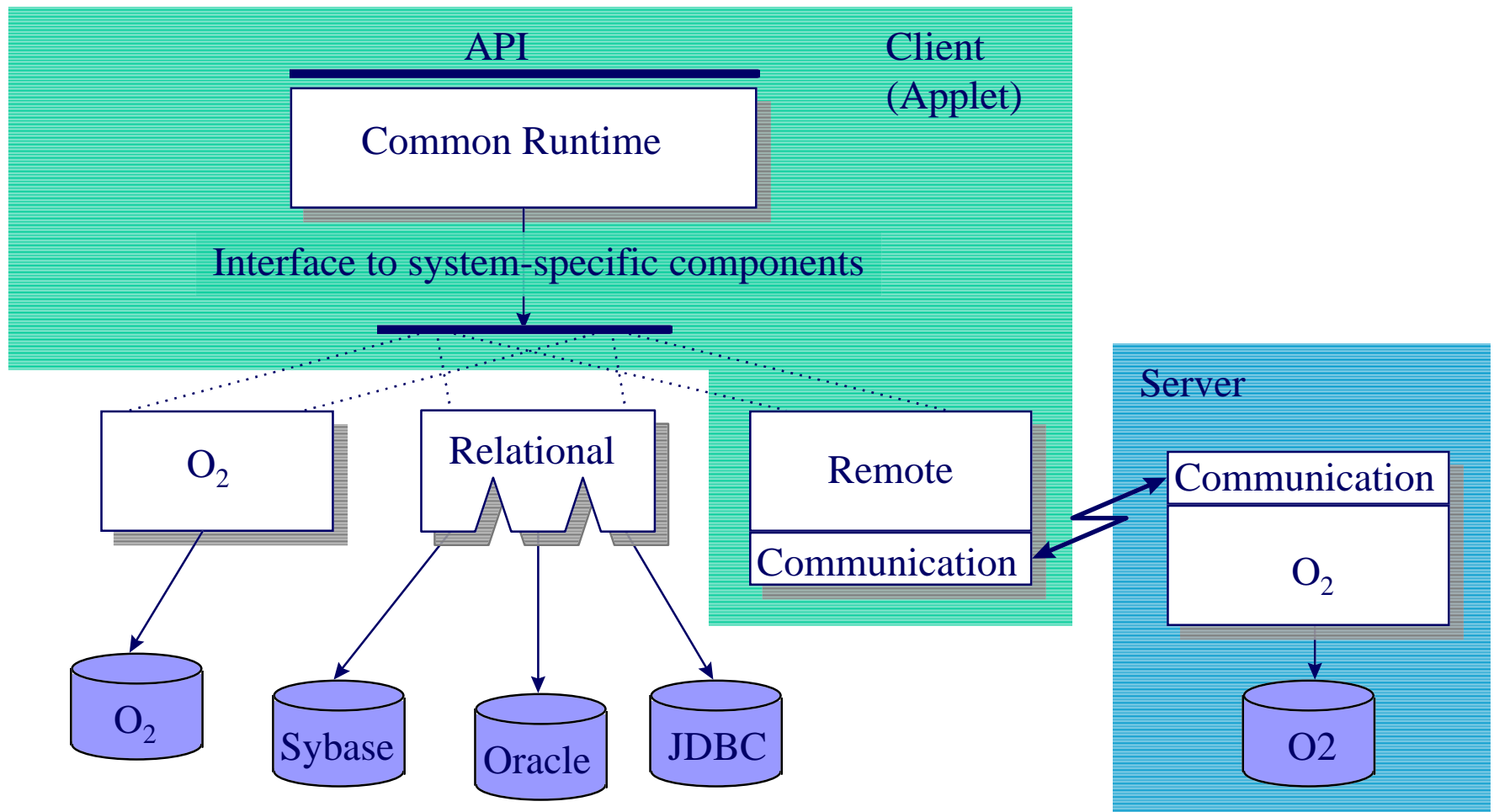
◆ Big database



Current status of implementation

- ◆ A beta version released
 - A production release planned for the end of 1997
- ◆ Still in development:
 - Bytecode postprocessor
 - Full ODMG compliance
- ◆ In perspective:
 - Addition of a 100% pure Java component using remote access

Future work: a remote access component



Conclusion

- ◆ Uniform access to various databases
- ◆ Open architecture, easy integration of other database systems
- ◆ Multi-language support for O₂
- ◆ Evolving towards an ODMG-compliant binding