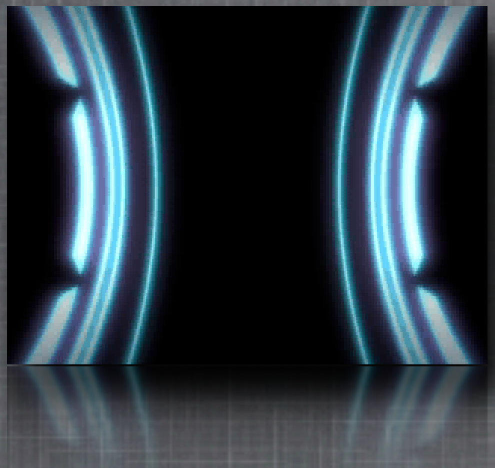


ODRA: A next-generation object-oriented environment for rapid database application development

Michał Lentner
Polish-Japanese Institute of Information Technology
Warsaw, Poland

This work is supported by
the European Commission 6-th Framework Programme,
Project VIDE - VIsualize all moDel drivEn programming,
IST 033606 STP

ADBIS 2007

Plan of the presentation

1. Motivations
2. Overview of the ODRA system
3. Detailed discussion of some critical decisions
4. Comparison with major existing solutions

Motivations

Complexity of current technologies designed for database application developers (eg. Java EE): dozens of frameworks, languages, servers, XML descriptors, code generators etc.

- Low-level programming/object-relational mapping due to impedance mismatch when Java/C#/C++/... is used.
- The need for object-oriented databases is still valid. Relational databases: simple data model, poor performance (joins), no support for object-oriented design and analysis (UML).
- The ODMG standard failed, new database architectures (stream, column-based) designed to solve other problems.
- Middleware (eg. CORBA brokers) do not support declarative, bulk data processing.

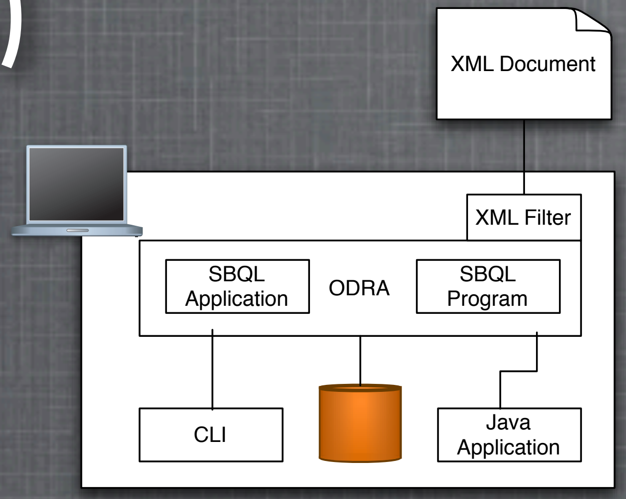
ODRA

A homogeneous development environment, consisting of three, highly integrated elements:

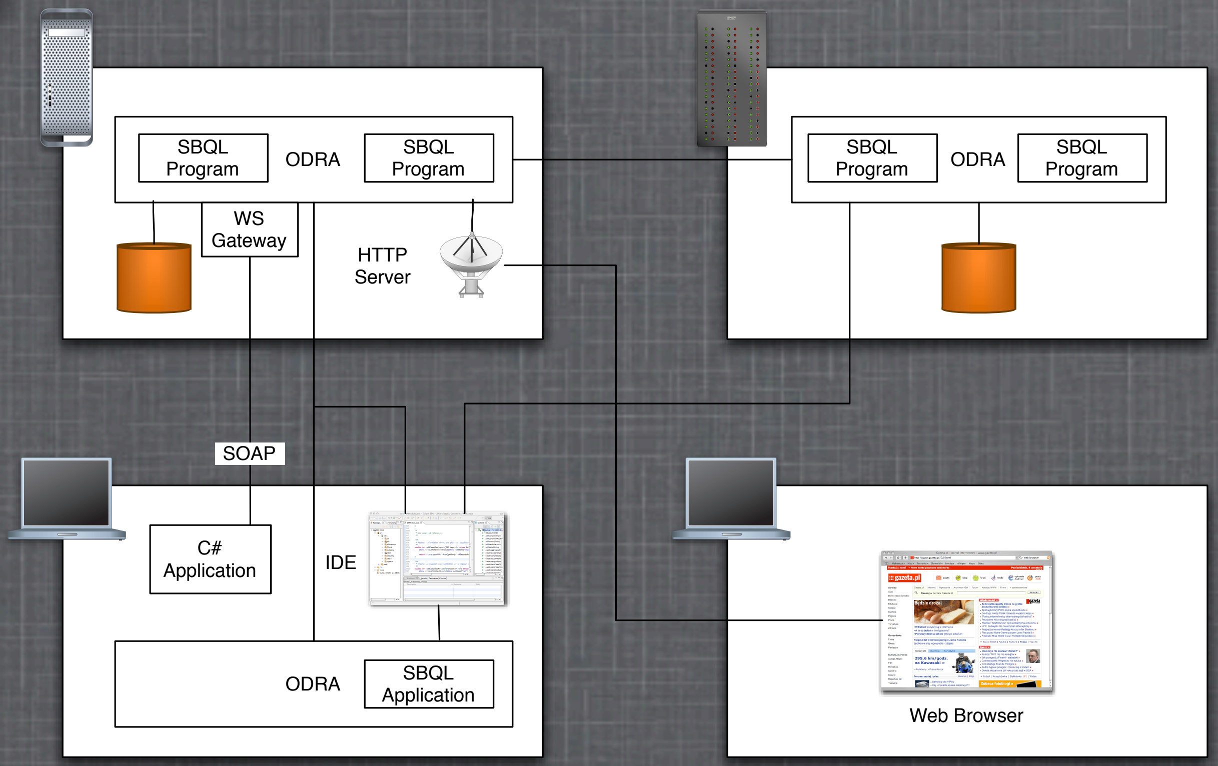
- object-oriented DBMS, completely different from ODMG-like architectures
- object-oriented query/programming language (based on the Stack-Based Approach and SBQL), with queries treated as expressions
- middleware based on updatable views and ideas known from federated databases

Scenarios of application

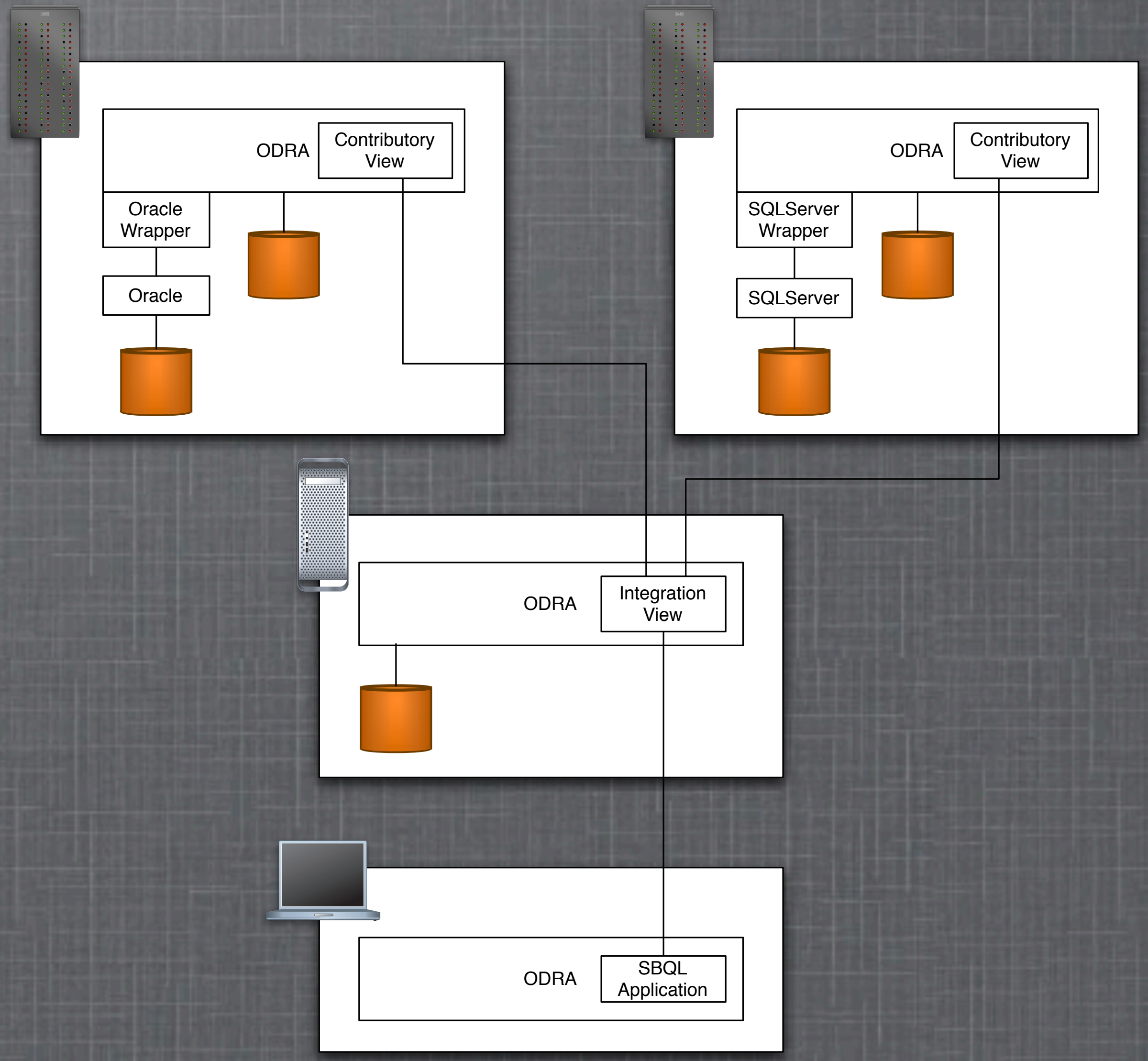
A)



B)



C)



a) non-distributed application
b) 3-tier client-server
c) federated database

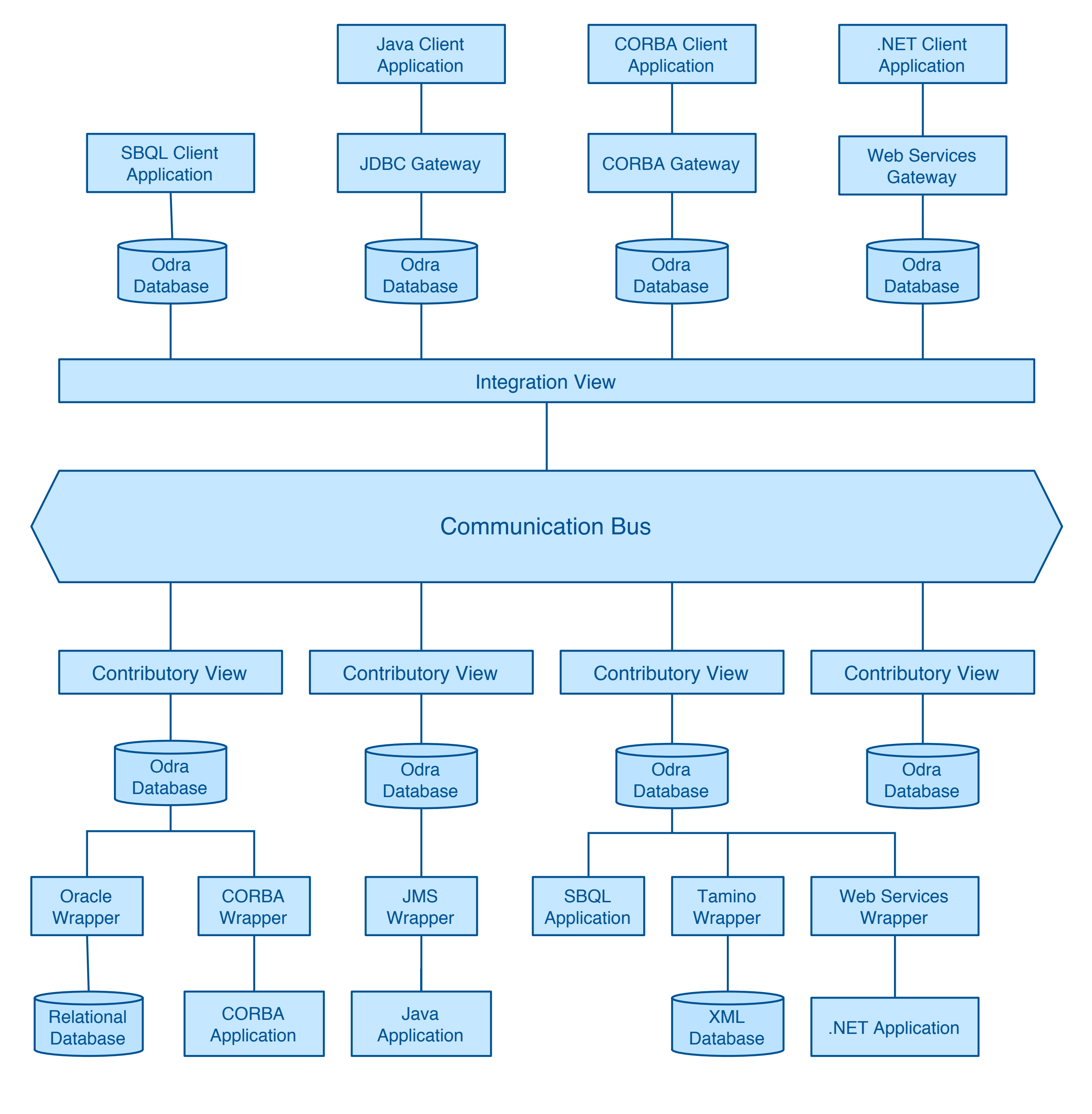
A simple, distributed application

```
module client {
   dblink aps appuser/apppasswd/appuser.appserver@my.appserver.pl;

   main() {
      print aps.count_employees(“Smith”);
   }
}
```

```
module appserver {
   dblink dbs1 dbuser1/apppasswd/dbuser1.dbserver@my.dbserver1.pl;
   dblink dbs2 dbuser2/dbpasswd/dbuser2.dbserver@my.dbserver2.pl;

   count_employees(n : string) : integer {
      return count (dbs1.emp union dbs2.emp) where ename = n;
   }
}
```

```
module dbserver {
   emp : record { ename : string; salary : integer; job : string; } [0..*]
}
```

eGovBus architecture



e-Gov Bus - Advanced e-Government Information Service Bus
(European Commission 6-th Framework Programme, IST-26727)

SBQL in ODRA

SBQL is a prototype query language that is used to explain the semantics of the Stack Based Approach.

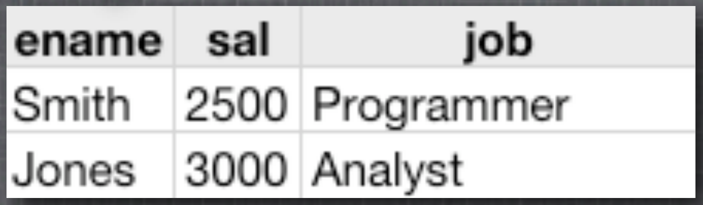
- SBQL in Odra has been extended to a database application programming language
- declarative, high-level, object-oriented programming
- queries as expressions
- typical programming language (modules, procedures, classes, etc.) and database (indexes, triggers, etc.) mechanisms
- semi-strong static type checking
- compile-time (e.g. query rewriting) and runtime (e.g. indexes) optimizers
- updatable views

Data model

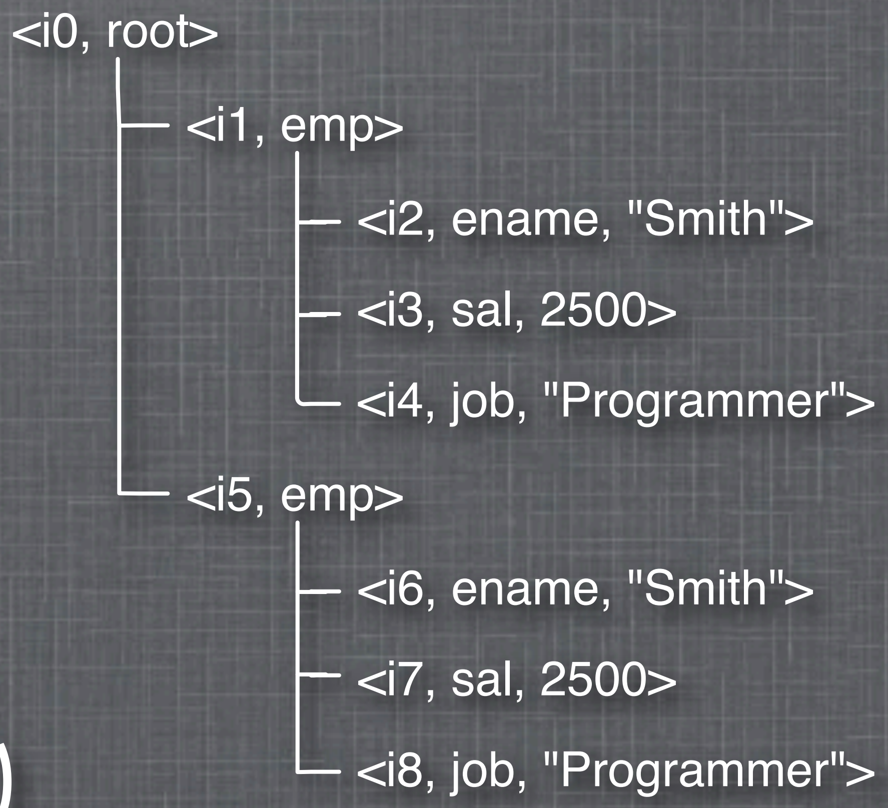
Basic data model (M0) is formed by three kinds of objects:

- simple objects <OID, name, value>
- reference objects <OID$_1$, name, OID$_2$>
- complex objects <OID, name, { object$_1$, object$_2$, ... }>

emp



ename	sal	job
Smith	2500	Programmer
Jones	3000	Analyst



<i0, root>

<i1, emp>

<i2, ename, "Smith">

<i3, sal, 2500>

<i4, job, "Programmer">

<i5, emp>

<i6, ename, "Smith">

<i7, sal, 2500>

<i8, job, "Programmer">

Other data models (M1, M2, ...) extend M0 by more and more advanced object-oriented constructs (classes, dynamic roles, interfaces, etc.)

SBQL queries

Basic grammar:

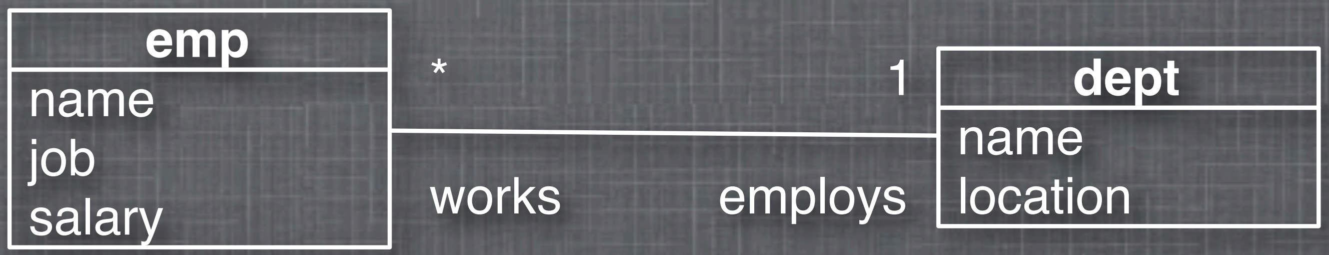
```
query ::=
        literal
      | name
      | unary_op query
      | query binary_op query
```

List of names and jobs of employees receiving salary = $2000 and working in departments located in Varna:

(dept where location = “Varna”).employs.(emp where salary = 2000).(name, job);

Query results:

1. Simple value (1, true, “cat”, etc.)
2. Reference
3. Binder (pair <name, result>)
4. Bag (collection)
5. Sequence (collection)
6. Structure
   (<single result$_1$, single result$_2$, …>)

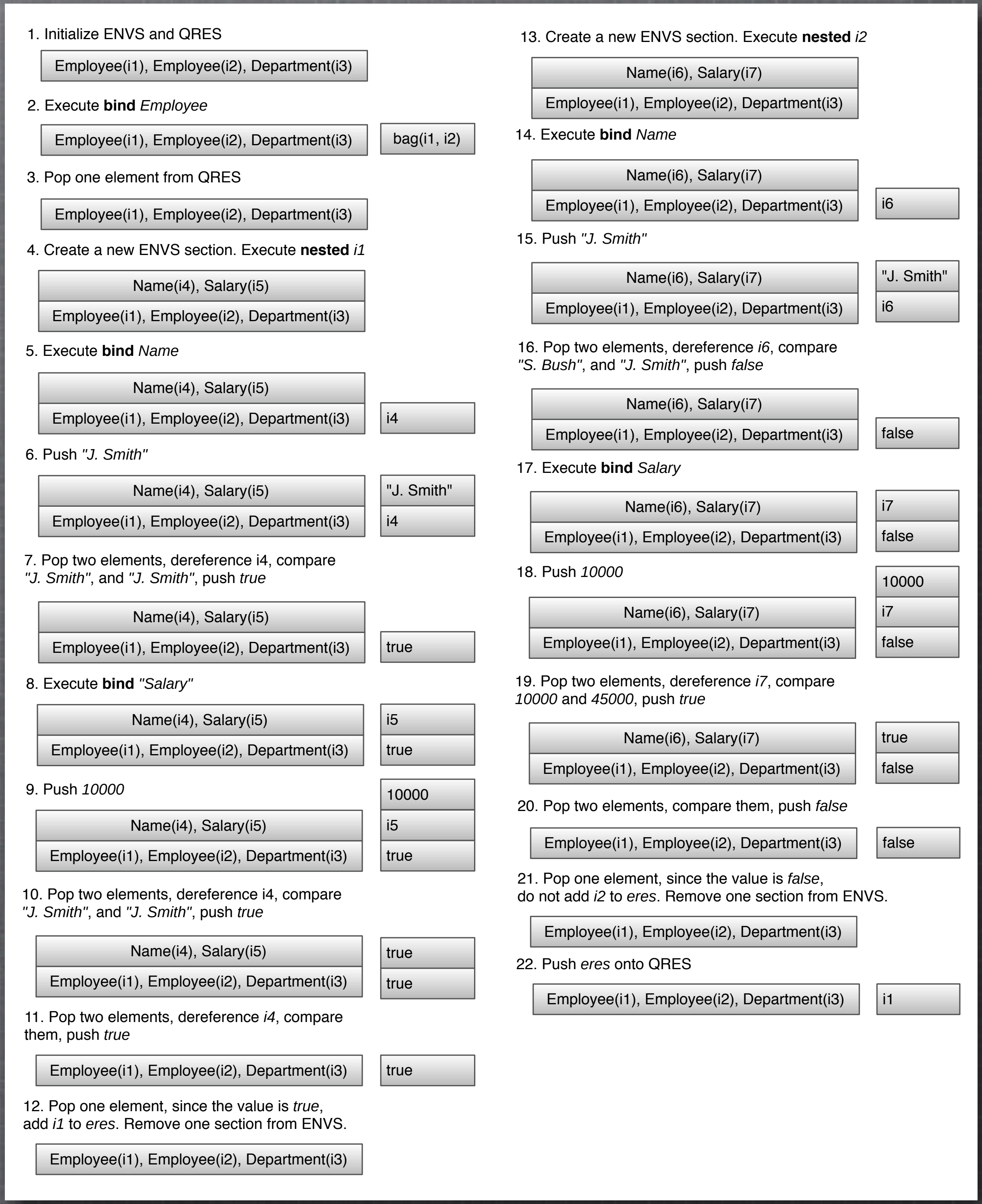


```
bag {
   struct { name1_OID, job1_OID },
   struct { name2_OID, job2_OID },
   …
}
```

Sample query evaluation

```
Employee where
   Name = “J. Smith”
   and salary > 10000
```

```
<i0, entry,
     <i1, Employee,
          <i4, Name, “J. Smith”>
          <i5, Salary, 65000>
     >
     <i2, Employee,
          <i6, Name, “S. Bush”>
          <i7, Salary, 45000>
     >
     <i3, Department,
          <i8, Name, “Sales”>
          <i9, Location, “London”>
     >
>
```



1. Initialize ENVS and QRES

Employee(i1), Employee(i2), Department(i3)

2. Execute **bind** *Employee*

Employee(i1), Employee(i2), Department(i3)
bag(i1, i2)

3. Pop one element from QRES

Employee(i1), Employee(i2), Department(i3)

4. Create a new ENVS section. Execute **nested** *i1*

Name(i4), Salary(i5)
Employee(i1), Employee(i2), Department(i3)

5. Execute **bind** *Name*

Name(i4), Salary(i5)
Employee(i1), Employee(i2), Department(i3)
i4

6. Push *"J. Smith"*

Name(i4), Salary(i5)
Employee(i1), Employee(i2), Department(i3)
"J. Smith"
i4

7. Pop two elements, dereference i4, compare *"J. Smith"*, and *"J. Smith"*, push *true*

Name(i4), Salary(i5)
Employee(i1), Employee(i2), Department(i3)
true

8. Execute **bind** *"Salary"*

Name(i4), Salary(i5)
Employee(i1), Employee(i2), Department(i3)
i5
true

9. Push *10000*

Name(i4), Salary(i5)
Employee(i1), Employee(i2), Department(i3)
10000
i5
true

10. Pop two elements, dereference i4, compare *"J. Smith"*, and *"J. Smith"*, push *true*

Name(i4), Salary(i5)
Employee(i1), Employee(i2), Department(i3)
true
true

11. Pop two elements, dereference *i4*, compare them, push *true*

Employee(i1), Employee(i2), Department(i3)
true

12. Pop one element, since the value is *true*, add *i1* to *eres*. Remove one section from ENVS.

Employee(i1), Employee(i2), Department(i3)

13. Create a new ENVS section. Execute **nested** *i2*

Name(i6), Salary(i7)
Employee(i1), Employee(i2), Department(i3)

14. Execute **bind** *Name*

Name(i6), Salary(i7)
Employee(i1), Employee(i2), Department(i3)
i6

15. Push *"J. Smith"*

Name(i6), Salary(i7)
Employee(i1), Employee(i2), Department(i3)
"J. Smith"
i6

16. Pop two elements, dereference *i6*, compare *"S. Bush"*, and *"J. Smith"*, push *false*

Name(i6), Salary(i7)
Employee(i1), Employee(i2), Department(i3)
false

17. Execute **bind** *Salary*

Name(i6), Salary(i7)
Employee(i1), Employee(i2), Department(i3)
i7
false

18. Push *10000*

Name(i6), Salary(i7)
Employee(i1), Employee(i2), Department(i3)
10000
i7
false

19. Pop two elements, dereference *i7*, compare *10000* and *45000*, push *true*

Name(i6), Salary(i7)
Employee(i1), Employee(i2), Department(i3)
true
false

20. Pop two elements, compare them, push *false*

Employee(i1), Employee(i2), Department(i3)
false

21. Pop one element, since the value is *false*, do not add *i2* to *eres*. Remove one section from ENVS.

Employee(i1), Employee(i2), Department(i3)

22. Push *eres* onto QRES

Employee(i1), Employee(i2), Department(i3)
i1

A sample program

```
module empdept {
   import another.module;

   type emptype is record {
       name : string;
       salary : integer;
       job : integer;
       works : ref dept [0..1];
   }

   type depttype is record {
       name : string;
       location : string;
       employs : ref emp [0..*];
   }

   emp : emptype [0..*];
   dept : deptype [0..*];

  count_unemploed_employees() : integer {
     return count emp where not exists works;
  }

  find_employees_by_name (n : string) : ref emp [0..*] {
     return emp where name = n orderby salary;
  }

  create_employee(ename : string; dn : string) : ref emp {
     return create emp :=
                       n as name,
                       (dept where name = dn) as works;
  }

  get_max_int(x : integer [0..*]) : a(b(integer)) {
     return (max x) as a as b;
  }
}
```

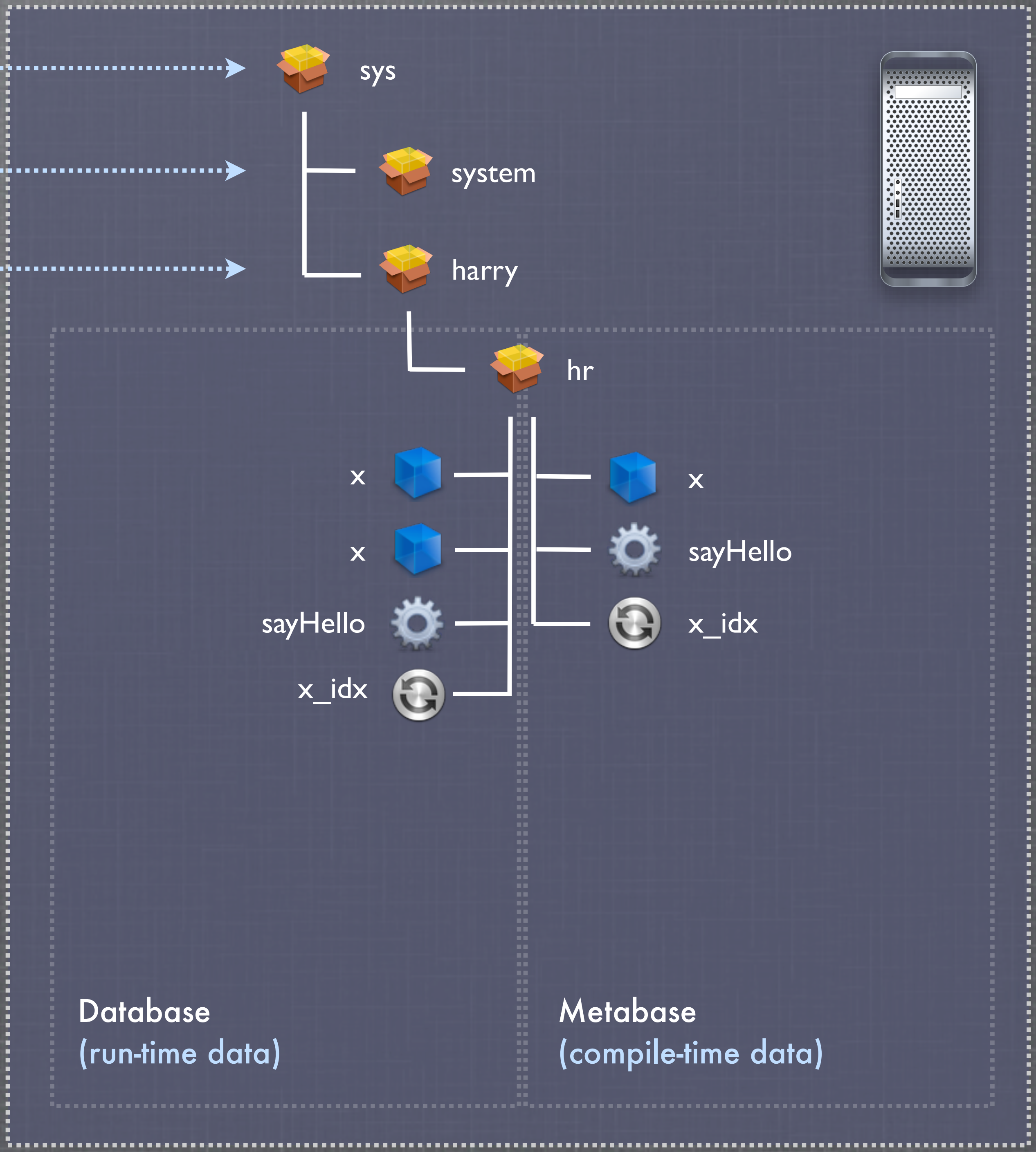
Database organization

Standard environment

Data dictionary, standard library and other system data

Root module of the user “harry” schema

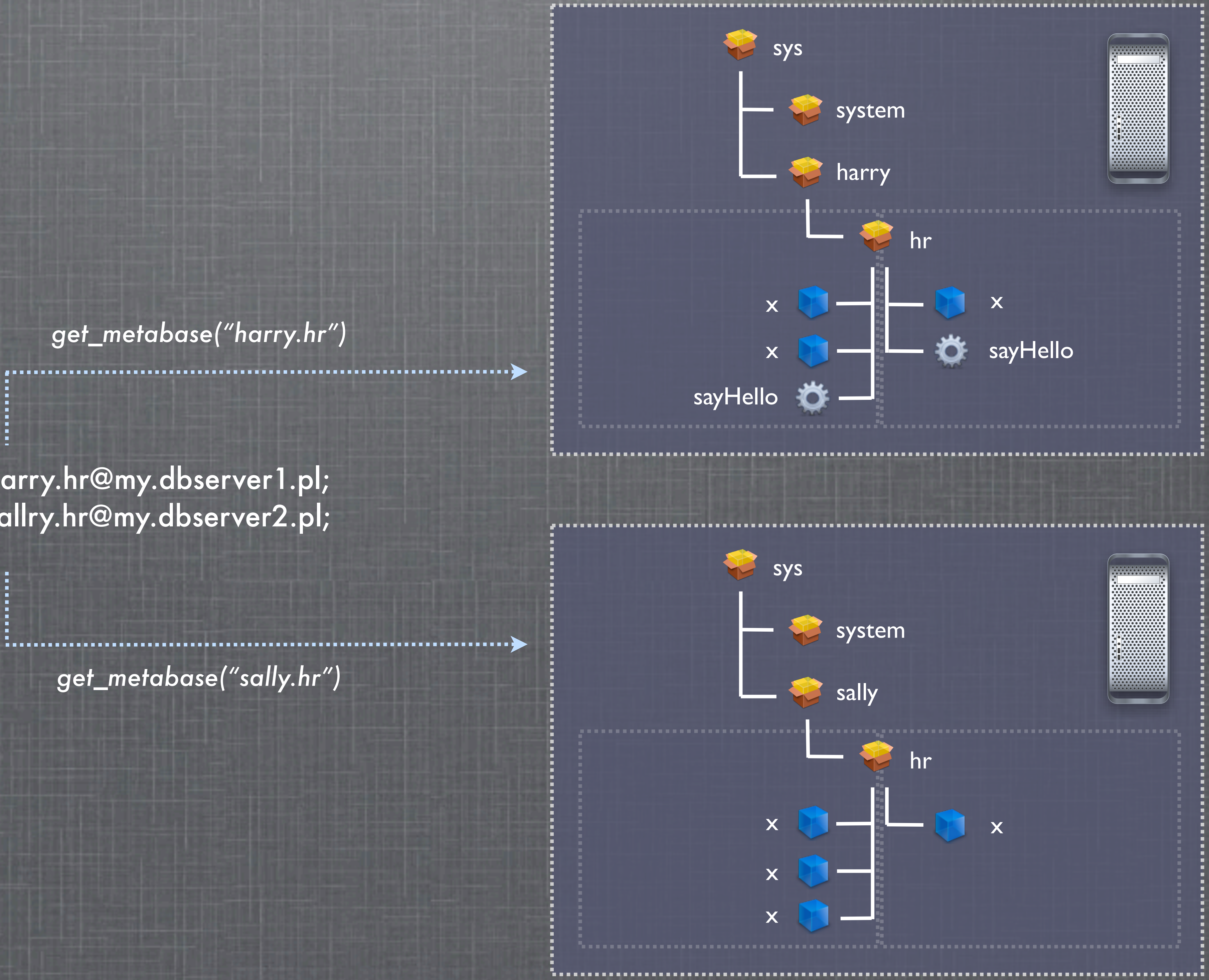
```
module hr {
  x [2..*] : integer;

  sayHello(name : string) {
    print “Hello, ” + name;
  }

  index x_idx on x;
}
```



Database
(run-time data)

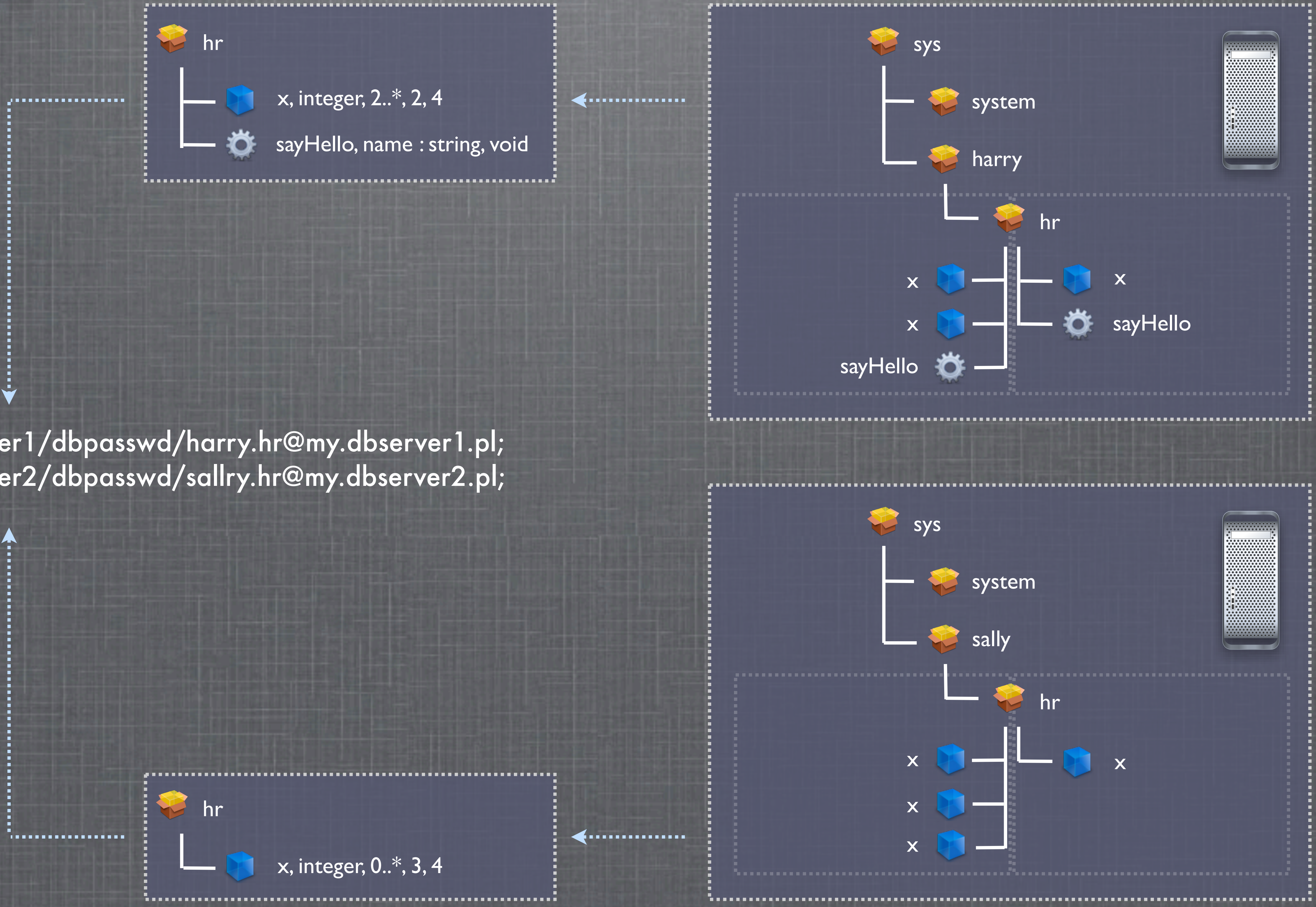
Metabase
(compile-time data)

Distributed communication



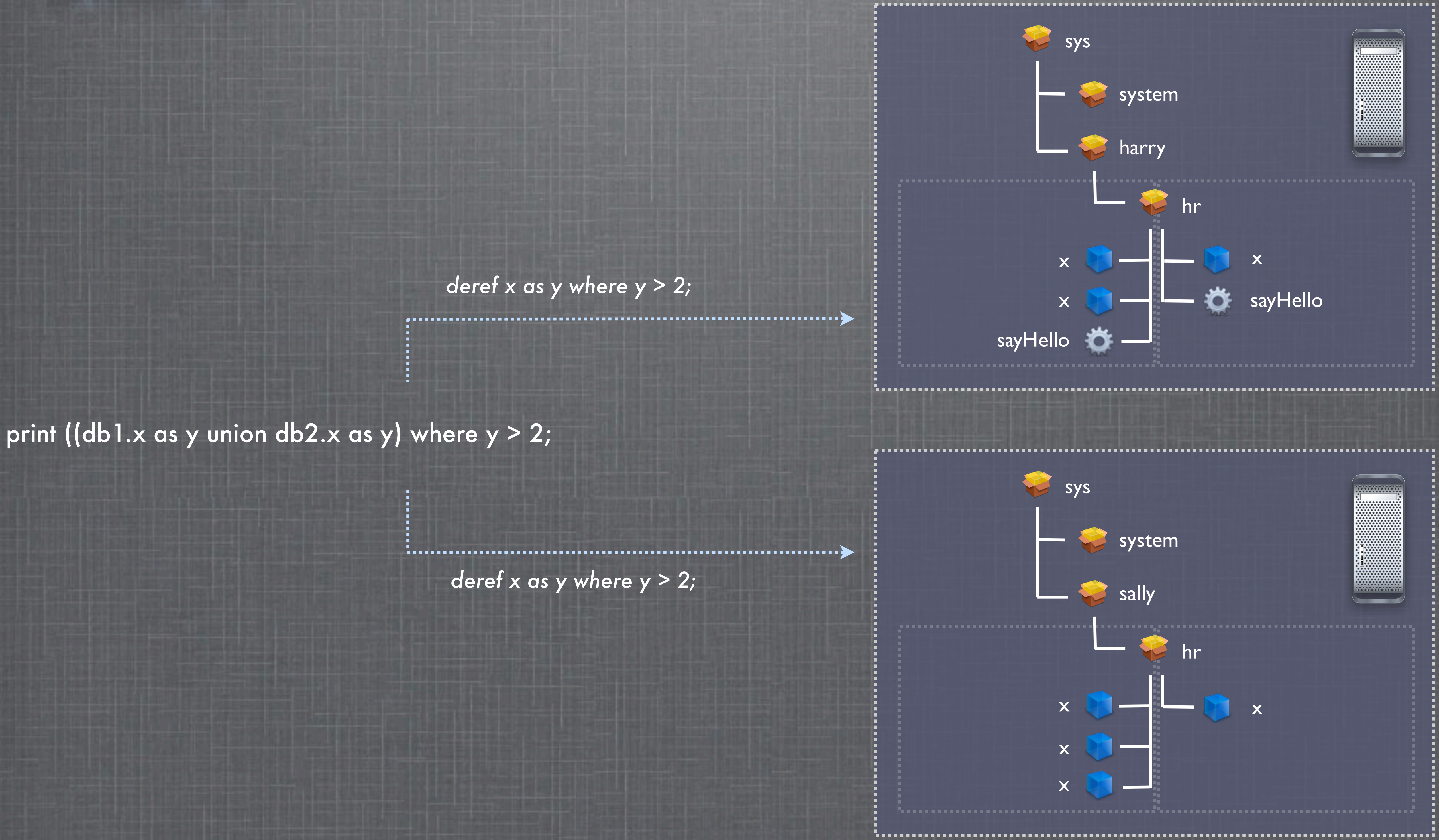
```
dblink dbs1 dbuser1/dbpasswd/harry.hr@my.dbserver1.pl;
dblink dbs2 dbuser2/dbpasswd/sallry.hr@my.dbserver2.pl;
```

Distributed communication



```
dblink dbs1 dbuser1/dbpasswd/harry.hr@my.dbserver1.pl;
dblink dbs2 dbuser2/dbpasswd/sallry.hr@my.dbserver2.pl;
```

Distributed communication

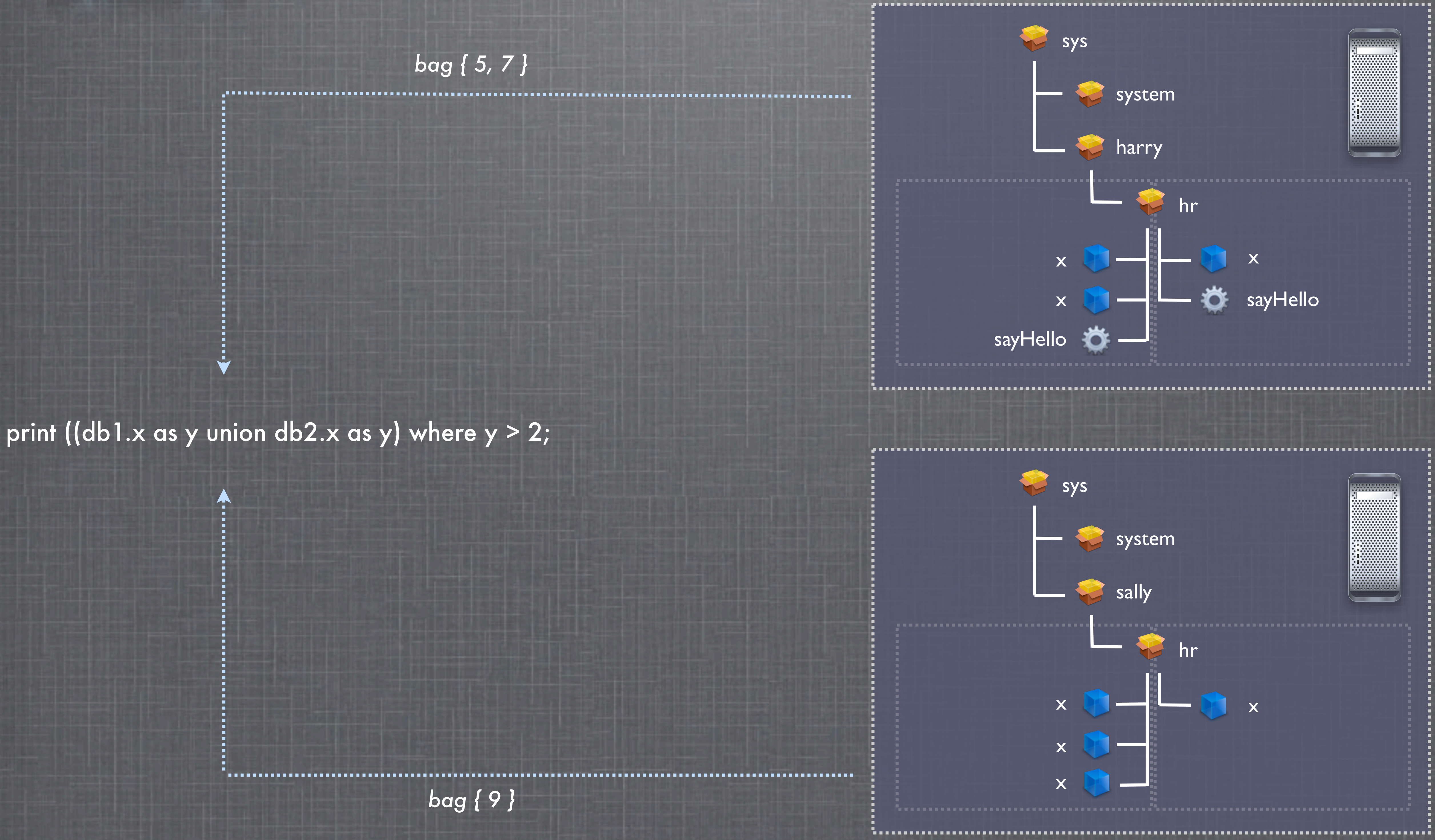


print ((db1.x as y union db2.x as y) where y > 2;

deref x as y where y > 2;

deref x as y where y > 2;

Distributed communication



bag { 5, 7 }

```
print ((db1.x as y union db2.x as y) where y > 2;
```

bag { 9 }

Updatable views in SBQL (1)

- view structure:

```
view viewname1 {
   virtual objects objectname1 { ... } // returns references to virtual objects
   on delete { ... } // optional (executed when a virtual object is deleted)
   on update { ... } // optional (executed when a virtual object is updated)
   on insert { ... } // optional (executed when an object is inserted into a virtual object)
   on retrieve { ... } // optional (executed when a virtual object is dereferenced)

   view viewname2 {
      virtual objects objectname2 { }
      on delete { ... }
      ....
   }
   ...
}
```

- references to virtual objects:

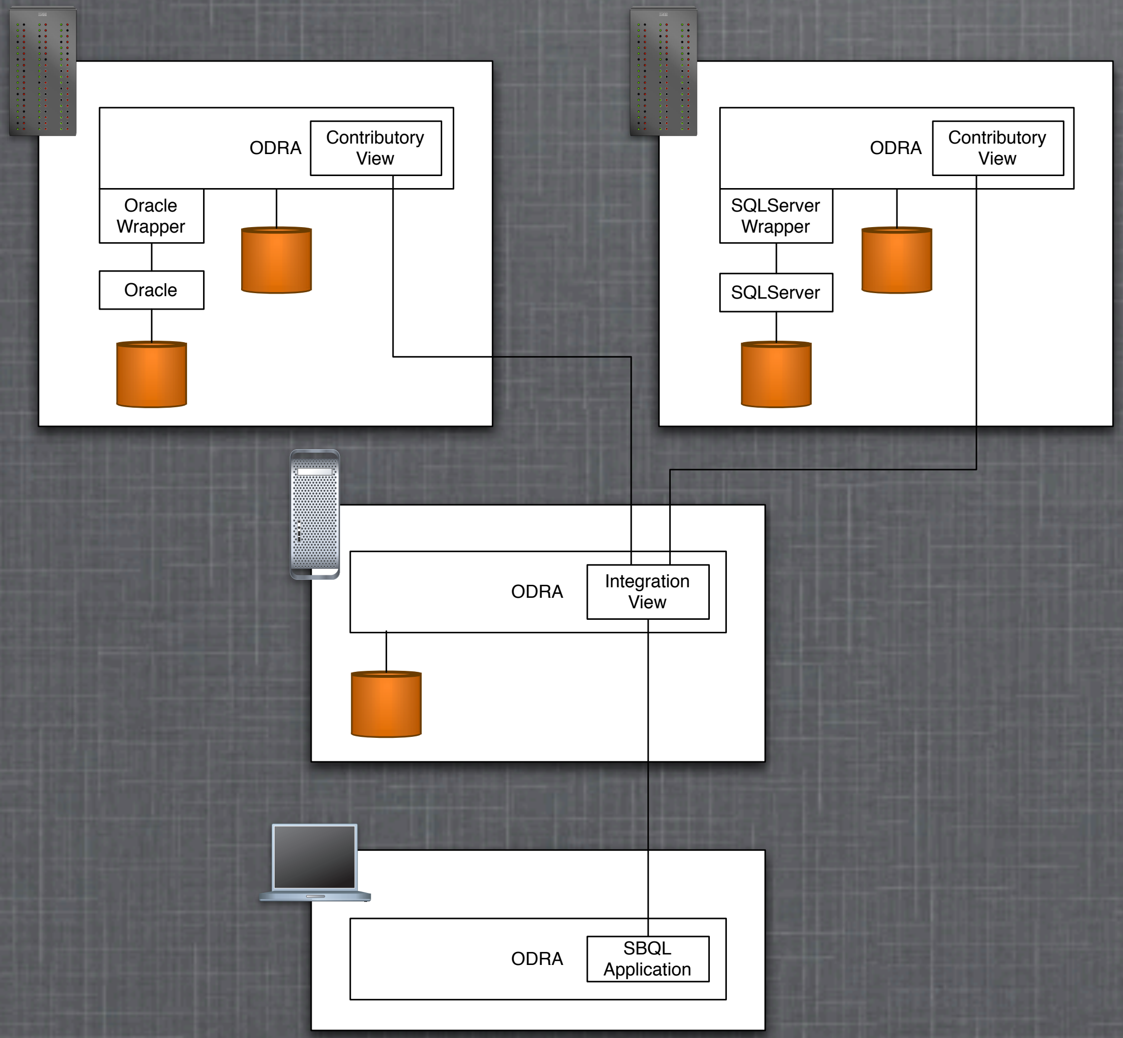
```
<i'm virtual,
   <view1_OID, seed1>,
   <view2_OID, seed2>,
   ...
 >
```

Updatable views in SBQL (2)

Relational databases:

```
CREATE VIEW salview AS
   select avg(sal) as sal from emp;
```

~~UPDATE salview SET sal = 1000;~~



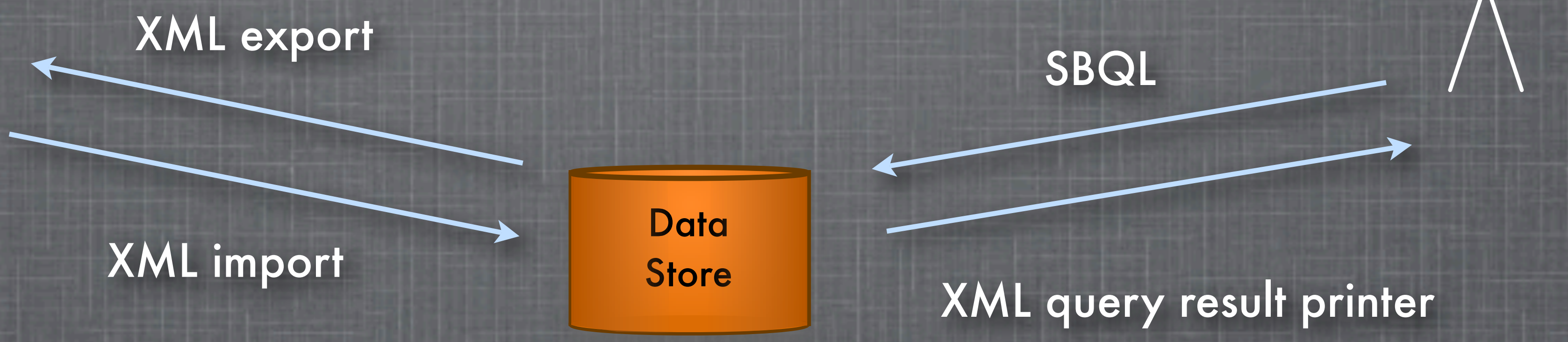
ODRA:

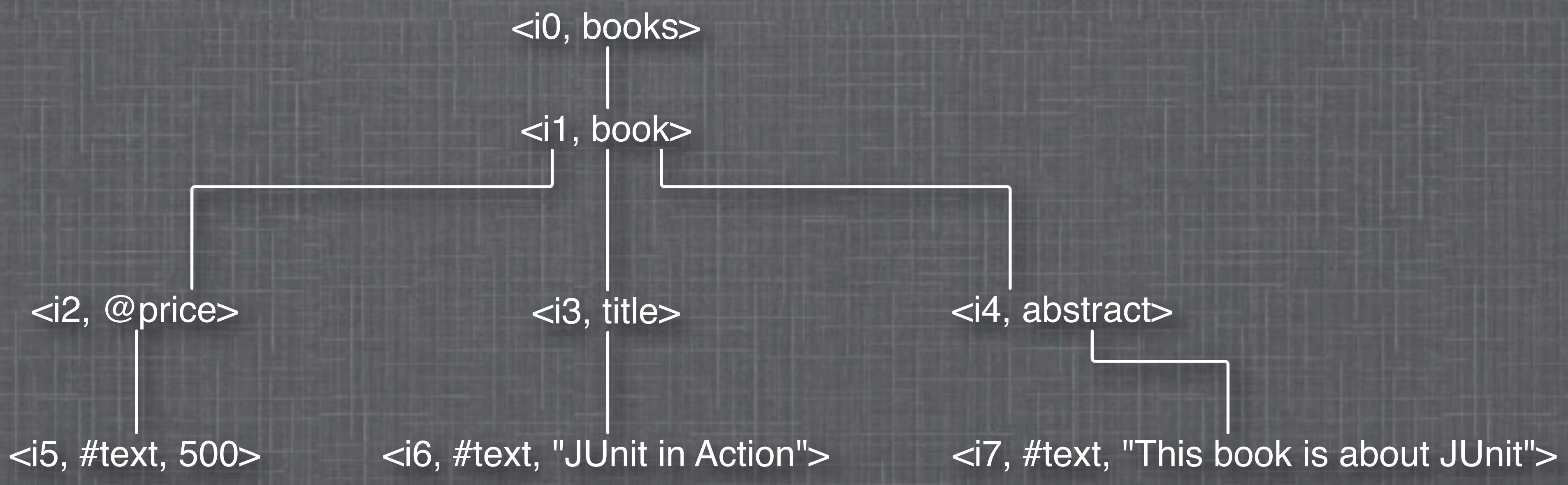
```
view salview {
   virtual objects sal : x(ref integer) {
      return avg(emp.sal) as x;
   }

   on retrieve : integer {
      return x;
   }

   on update (n : integer) {
      for each (emp)
         sal := (integer) n / count(emp);
   }
}

sal + 5;

sal := 1000;
```

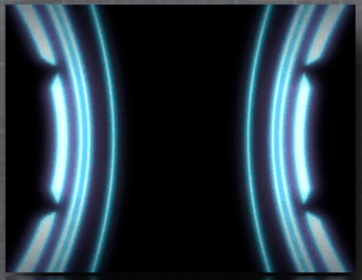
ODRA and XML

GOAL: no XML inside the database

```
<books>
  <book price=”500”>
    <title>JUnit in Action</title>
    <abstract>This book is about JUnit</abstract>
  </book>
</books>
```







SBQL and XML

```
(1 union 2) as x groupas y
↓
 y(bag { x(1), x(2) })
↓
<y>
  <x>1</x>
  <x>2</x>
</y>
```

```
(books.book where
  @price < 1000).(title, @price)
  as cheapbook groupas books;
↓
books(cheapbook(struct{ i3, i2 }))
↓
books(cheapbook(
  struct {
     title(“JUnit in Action”),
     @price(500)
  }
))
↓
<books>
  <cheapbook price=”500”>
     <title>JUnit in Action</title>
  </cheapbook>
</books>
```

```
type bookstype {
   book : record {
      @price : string;
      title : #text(string);
      abstract : #text(string);
      author : #text(string)[0..2];
   } [0..*];
}
b : bookstype;

set_author(b : booktype) {
   x : integer := count b.books
      where not exists author;

   foreach (b.books as x where
    not exists author)
      x.(create author(“unknown”));
   return x;
}
↓  proc();
1
```

ODRA vs. other solutions

- ODRA vs. relational databases: rich, object oriented data model.
- ODRA vs. database application programming languages: no impedance mismatch, queries as expressions, database-like services (eg. persistence).
- ODRA vs. ODMG/JDO/Hibernate/EJB/Cω/Linq: well defined object-oriented query language, no impedance mismatch for queries, fewer problems with optimization, no need for “two worlds of objects”, no code generation, no XML descriptors.
- ODRA vs. db4o (native queries): optimization techniques work on queries, not on low-level byte code.
- ODRA vs. XQuery: independend of XML, syntax much easier to understand, can be used to create client-side application (with GUI), static type checking.
- ODRA vs. CORBA: no code generation, automatic code optimization.

Thank you!

More information on SBQL and our projects:
http://www.ipipan.waw.pl/~subieta