Inverse Functions in the AquaLogic Data Services Platform

Nicola Onose (UCSD)
joint work with
Vinayak Borkar and Michael Carey (BEA Systems)

Outline

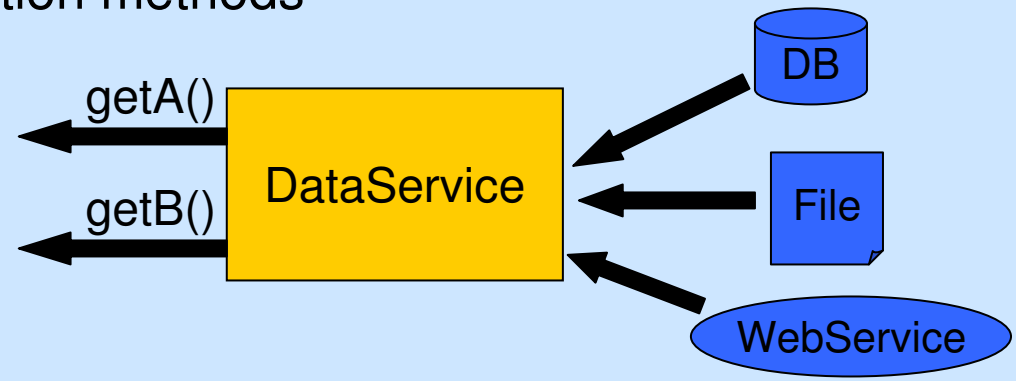
- Background on AquaLogic Data Services Platform
- Motivating Example
- Using Inverse Functions
- Implementation
- Experiments
- Conclusions

Intro

- Legacy applications: relational DB + business objects implementing application logic
- Web applications need to integrate data
- A first solution: **Web services**.
- Disadvantage: black boxes, no information regarding the semantics underneath
- A second step: **data services**
- AquaLogic Data Services Platform (ALDSP) uses data services to integrate data coming from various sources

Data Services

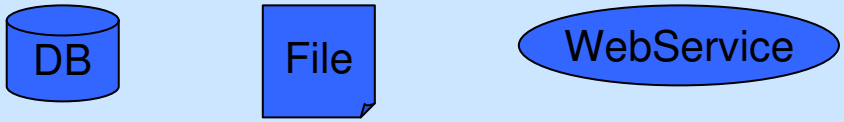
- Modeling data with data services:
  service = schema + set of XQuery functions (methods)
  - read methods
  - navigation methods



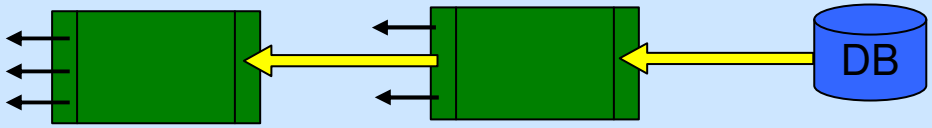
- Data service ≈ ***view*** (in the classical DB world)
- Data service methods typically contain **data transformations** (function calls).
- Generic functions vs. optimizations ⇒ in this talk

Query Processing: The Big Picture

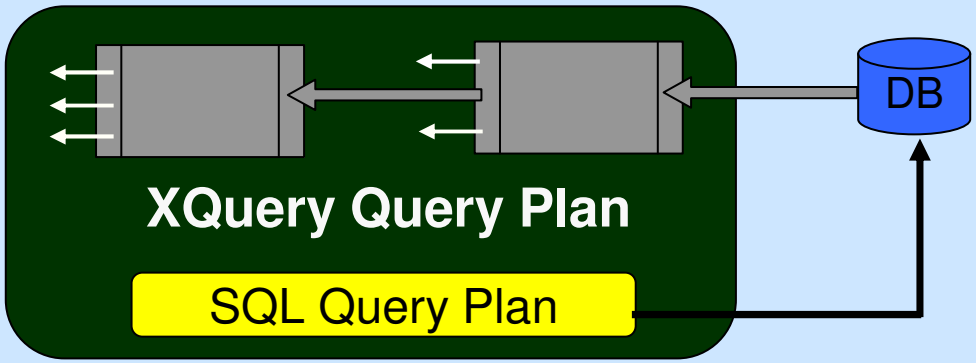
- start from the data sources



- build a hierarchy of services



- query plan ⇐ function inlining / view unfolding



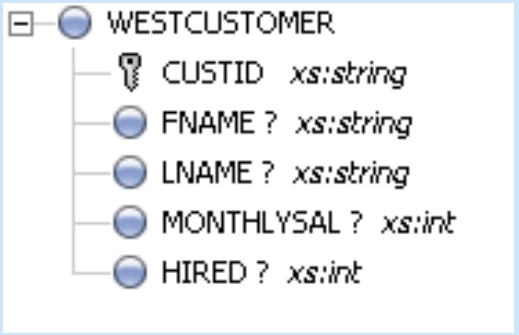
Our Problems and Approach

- Services can use **external functions** when building **views** over the physical data. (examples later)
- Query plan may contain **selections or joins** over views.
- If **functions not supported** by the DB
  - conditions **cannot be pushed** to the DB engine
  - mediator needs to do all the work (kills most of ALDSP optimizations)
- Also, such views are **non-updatable**.
- Use **inverse functions** and other function properties to enable optimizations and updates.
- How: explained in this talk.

Step 1: mapping the source data into XML

- West Customers example: accessing the data (job placement firm)

```
CREATE TABLE WESTCUSTOMER (
  CUSTID VARCHAR(10) NOT NULL,
  FNAME VARCHAR(20),
  LNAME VARCHAR(20),
  MONTHLYSAL INTEGER,
  HIRED INTEGER );
```

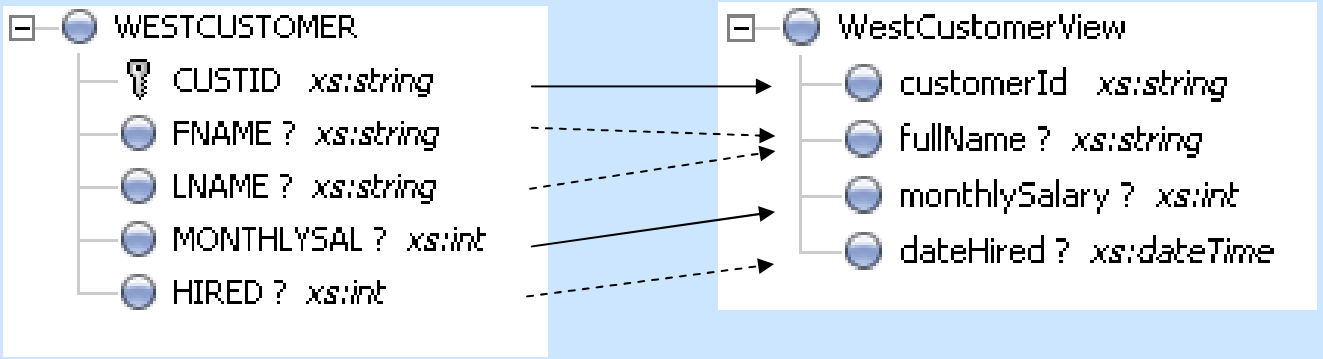


data stored in a relational table

XML view (using default mapping) = a *physical service* in ALDSP terminology

Step 2: design and implement the data service

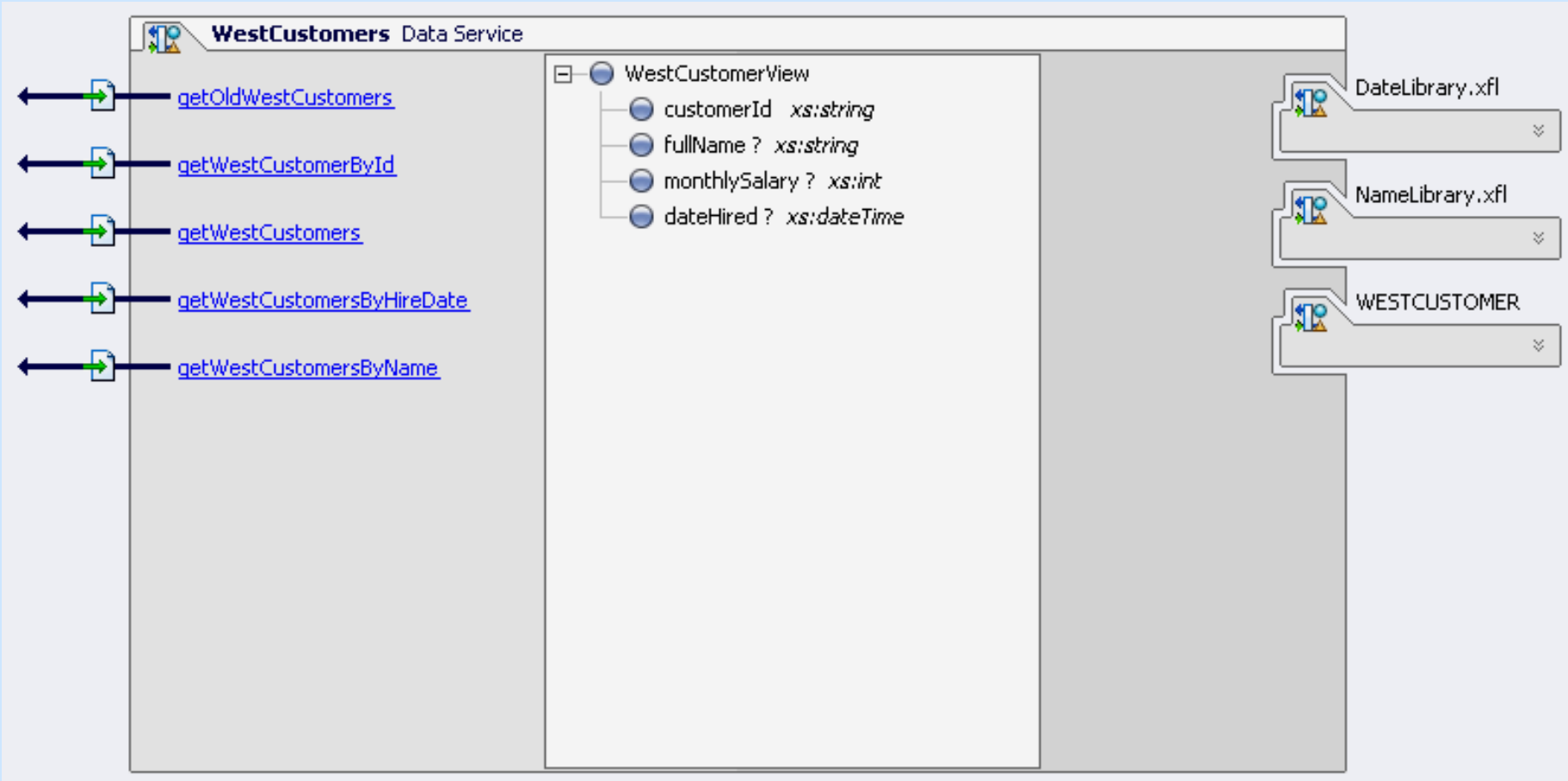
- West Customers example: adding application logic



Dashed arrows: the target element is created by a transformation (described later)

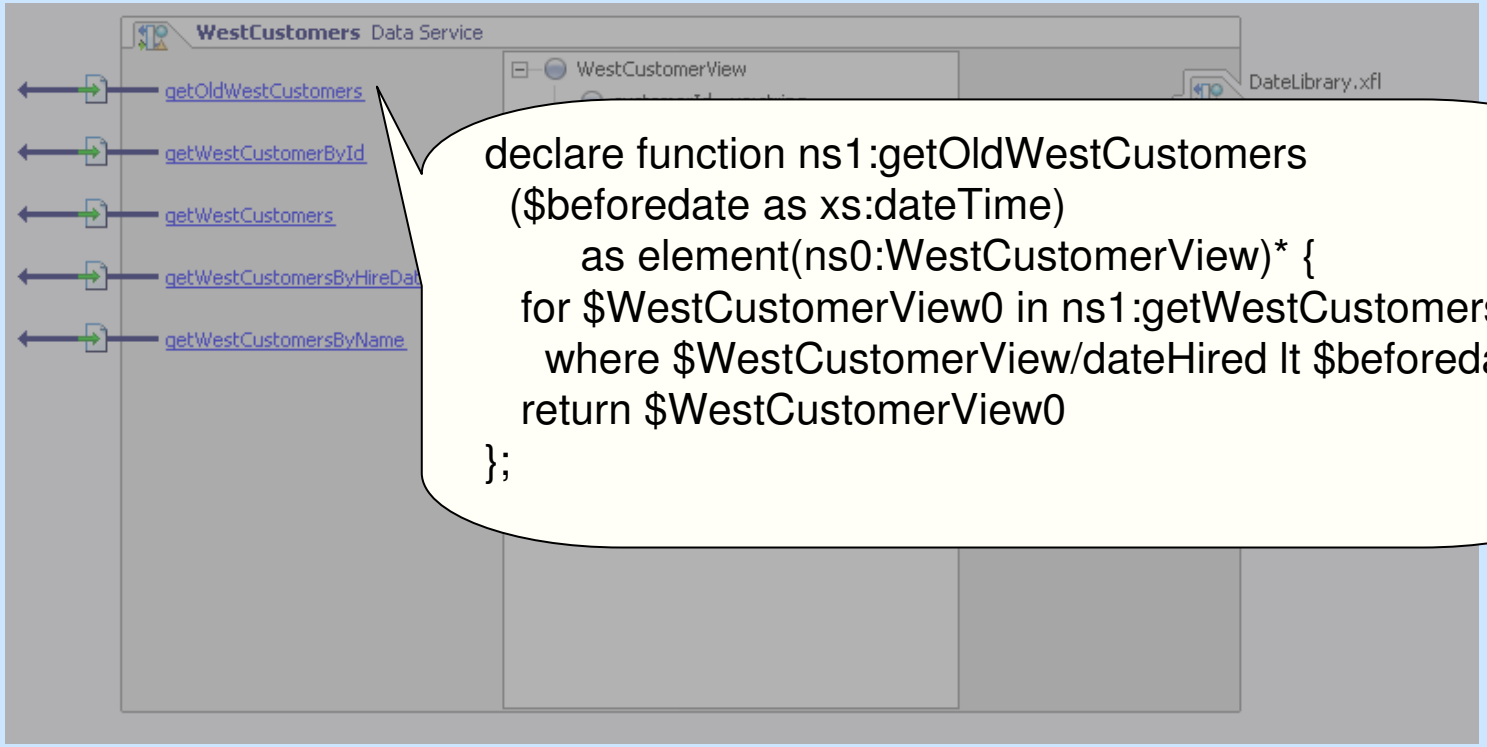
Step 2: design and implement the data service

- West Customers example: adding application logic



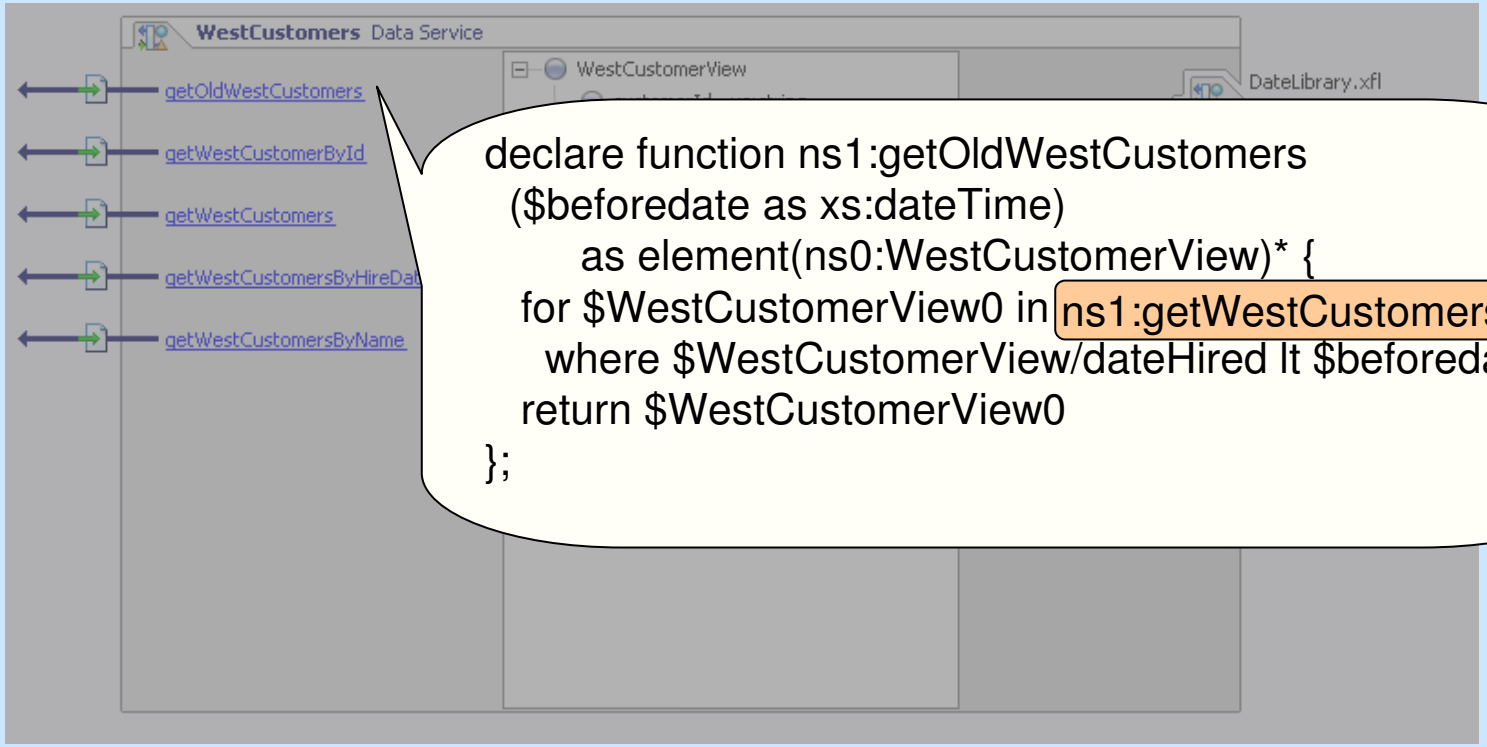
Read method example

- West Customers example
- Read method implemented as a selection over the view



View Unfolding

- Inline XQuery functions

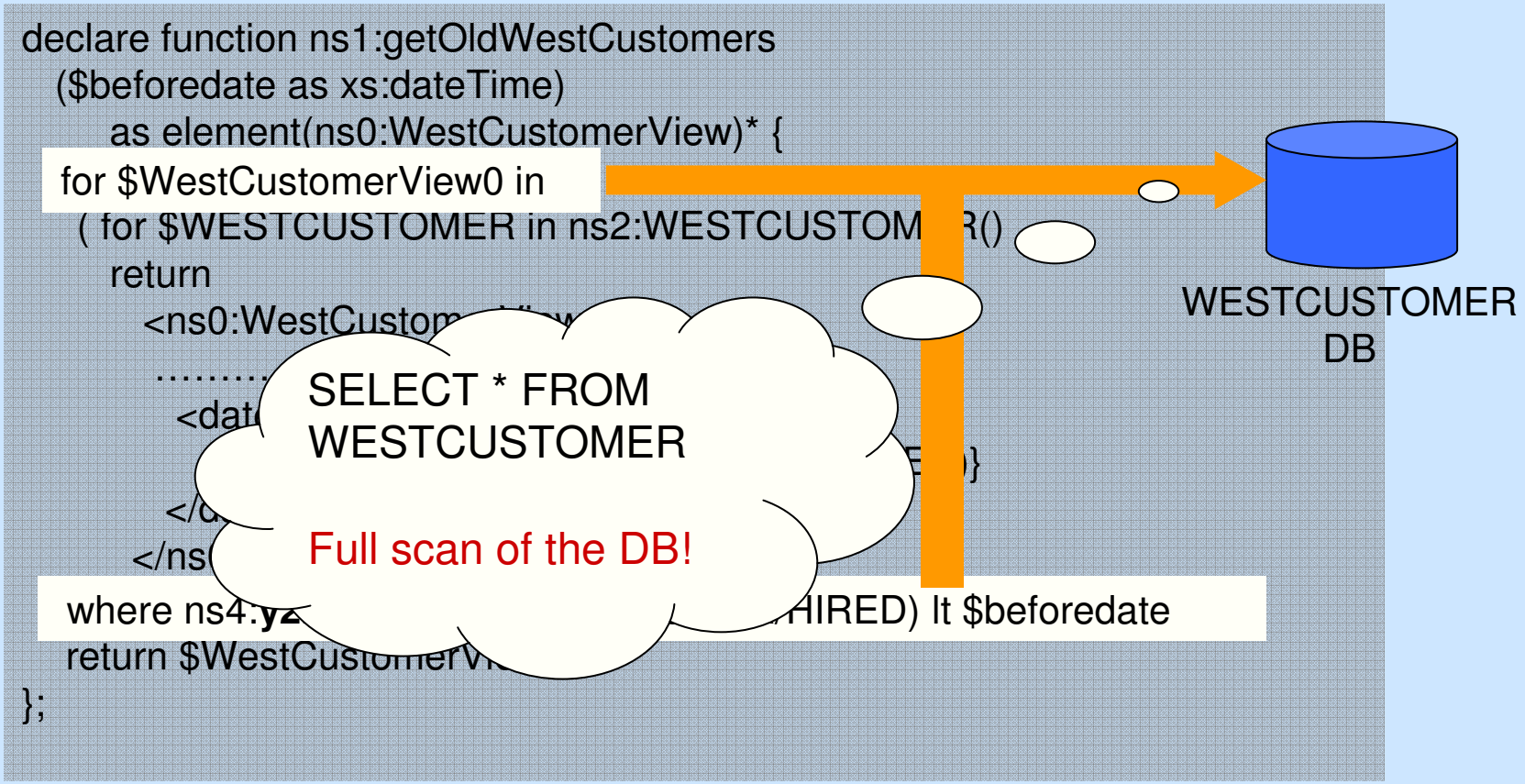


After View Unfolding

- a call to getOldWestCustomers expands into …

```
declare function ns1:getOldWestCustomers
   ($beforedate as xs:dateTime)
       as element(ns0:WestCustomerView)* {
    for $WestCustomerView0 in
     ( for $WESTCUSTOMER in ns2:WESTCUSTOMER()
            return
              <ns0:WestCustomerView>
               ……..
                 <dateHired>
                     {ns4:y2kdate($WESTCUSTOMER/HIRED)}
                 </dateHired>
              </ns0:WestCustomerView> )
    where $WestCustomerView0/dateHired lt $beforedate
    return $WestCustomerView0
};
```

External Function Calls Preclude Optimizations



The Problem

- Some transformations are implemented by external functions (e.g. written in Java)
- Consequence:
  - bottleneck in pushing the queries to the underlying sources
  - no declarative way of updating the views
- But if functions are invertible and their inverses are declared as such, it is often possible to rewrite into an equivalent condition that can be pushed to the source.
- e.g. y2kdate($WESTCUSTOMER/HIRED) lt $beforedate

  ≡

  $WESTCUSTOMER/HIRED lt y2kdays($beforedate)

Optimized Query Plan

Optimized Rewriting

- How are optimizations enabled?
- Solution idea: declare **which** functions can *invert* the transformations and **how** they can be used.

  $\Leftrightarrow$

  to f(x,y,..) associate inverses f_x^{-1}, f_y^{-1}, …

  and ways of rewriting $E_1(f(x,y,..)) \rightarrow E_2(f_x^{-1}(u), f_y^{-1}(u),\ldots)$

- Inverses and transforms are, in general, registered by the user, helped by the UI.
- Certain properties can be inferred, based on monotonicity.

Rewrites(1:1)

- To a given function, one can associate:
  - an *inverse*
    e.g.: y2kdays is the inverse of y2kdate
  - a set of *equivalent transforms*, describing how the inverses behave
    e.g.: y2kdate(h) lt x ≡ h lt y2kdays(x)
    (in this case, the inverse preserves monotonicity)

Rewrites(1:N)

- The same thing can be done for 1:N transformations

```
declare function ns1:getWestCustomersByName($fullname) {
  for $WestCustomerView in ns1:getWestCustomers()
   where $WestCustomerView/fullname = $fullname
   return
      $WestCustomerView
};
```

- several *inverses* (one inverse for each input parameter)

Rewrites(1:N)

- After inlining and simplifications

```
declare function ns1:getWestCustomersByName($fullname) {
  for $WESTCUSTOMER in ns2:WESTCUSTOMER()
   where ns3:fullname($WESTCUSTOMER/LNAME,
                      $WESTCUSTOMER/FNAME)
         eq $fullname
   return
         <ns0:WestC                erView>
          ..............
          </n
};
```

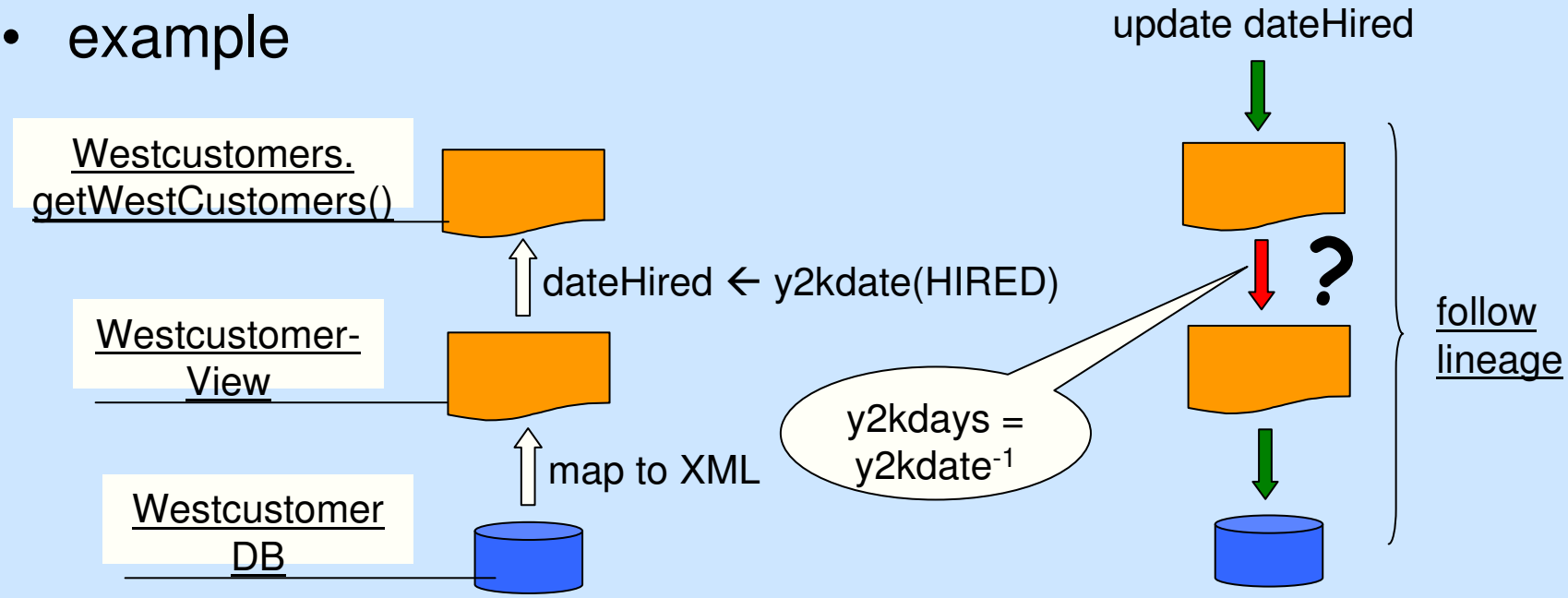
```
$WESTCUSTOMER/LNAME eq ns3:lname($fullname)
and
$WESTCUSTOMER/FNAME eq ns3:fname($fullname)
```

- We know that

  fullname() has lname(), fname() as *inverses* and
  fullname($l,$f) eq $n ≡ $l eq lname($n) and $f eq fname($n)

Updates

- Inverse functions not only permit pushing selections and joins, but also allow updating views.
- example



- more details in [V.Borkar, M.Carey, D.Lychagin, T.Westmann, D.Engovatov, N.Onose VLDB2006]

Implementation Challenges

- Rewrites declared as equivalences of expressions containing free variables
- Termination of the rewriting process is undecidable
- Simple restrictions (such as acyclicity) are unsatisfactory as they disallow certain usecases.
- See paper for details.

Solution

- Consider each transform as a directed rule:
  e.g. y2kdate(h) lt x → h lt y2kdays(x)
- Analyze graph of dependencies between pairs of invertible functions and boolean operators
  examples of nodes: (lt, y2kdate), (eq, fullname) etc
  examples of edges: (lt, y2kdate) → (lt, y2kdays)
- Compute a heuristic bound on the number of rule applications, based on
  - the graph
  - the total number of invertible functions
- Incomplete strategy (the problem is undecidable), but captures a significant number of test cases.

Experiments

Inverses	Indexes	100K customers	10K customers	1K customers	compile time
No	No	14400 ms	1500 ms	125 ms	125 ms
Yes	No	2600 ms	250 ms	15 ms	125 ms
Yes	Yes	8 ms	7 ms	5 ms	125 ms

Times for calling getWestCustomersByName(), in various settings

- for each table size, the call returned one XML element corresponding to one tuple in the DB
- indexed case: a composite index on (LNAME, FNAME)
- even when no index $\Rightarrow$ visible improvement
  (DB engine performs a scan) (less materialization, less work on mediator)

Related Work

- very little (surprisingly)
- OpenLink: a system that allows registering inverses for (monotonic) SQL functions
- ADT-Ingres, Postgres: enable indexes based on abstract data types

Conclusions

- External function/service calls are usually opaque to DB optimizations
- Inverse functions together with transforms enable optimizations in a declarative way
- Implementation
  - carried out as part of a summer project
  - part of the ALDSP product since version 2.5
- Future work:
  - better strategies for rewriting
  - a formal study of optimality of using inverse function

Questions?

→ Download: http://www.bea.com,
follow links to Products/ALDSP
→ Docs. for using the feature (in dev2dev):
http://edocs.bea.com/aldsp/docs25/