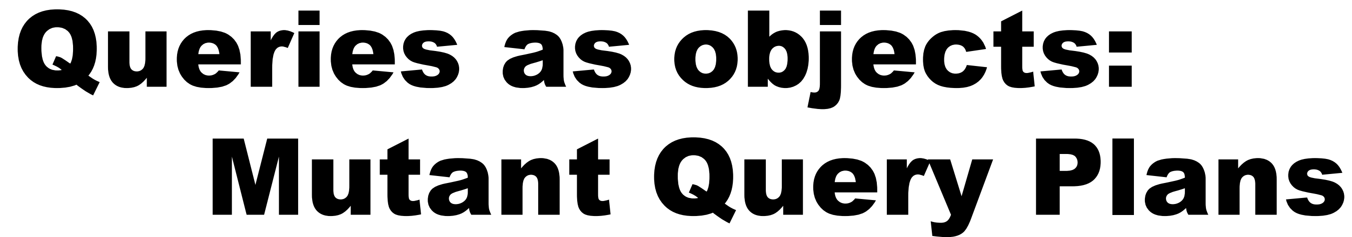
Queries as objects: Mutant Query Plans



David Maier

Vassilis Papadimos

OGI School of S. & E.

Oregon Health & Science University

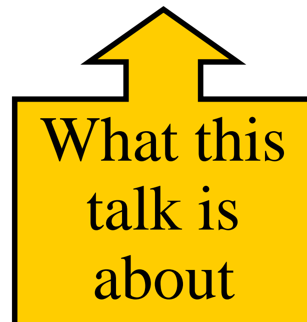
{maier,vpapad}@cse.ogi.edu

Why represent queries as objects

- Easier to create, manipulate, store under programmatic control
- Easier embedding in a programming language
  - *Database command ADT*
- Allows sharing of arbitrary subpieces
- Easier to include “data templates” for complex structures (e.g., cycles)

Arbitrary constants in queries

- Specific objects (by ID)
- Values of arbitrary shapes and sizes
- Supports partial evaluation
  - Don’t repeat part of query that doesn’t change
  - Supports distributed query evaluation



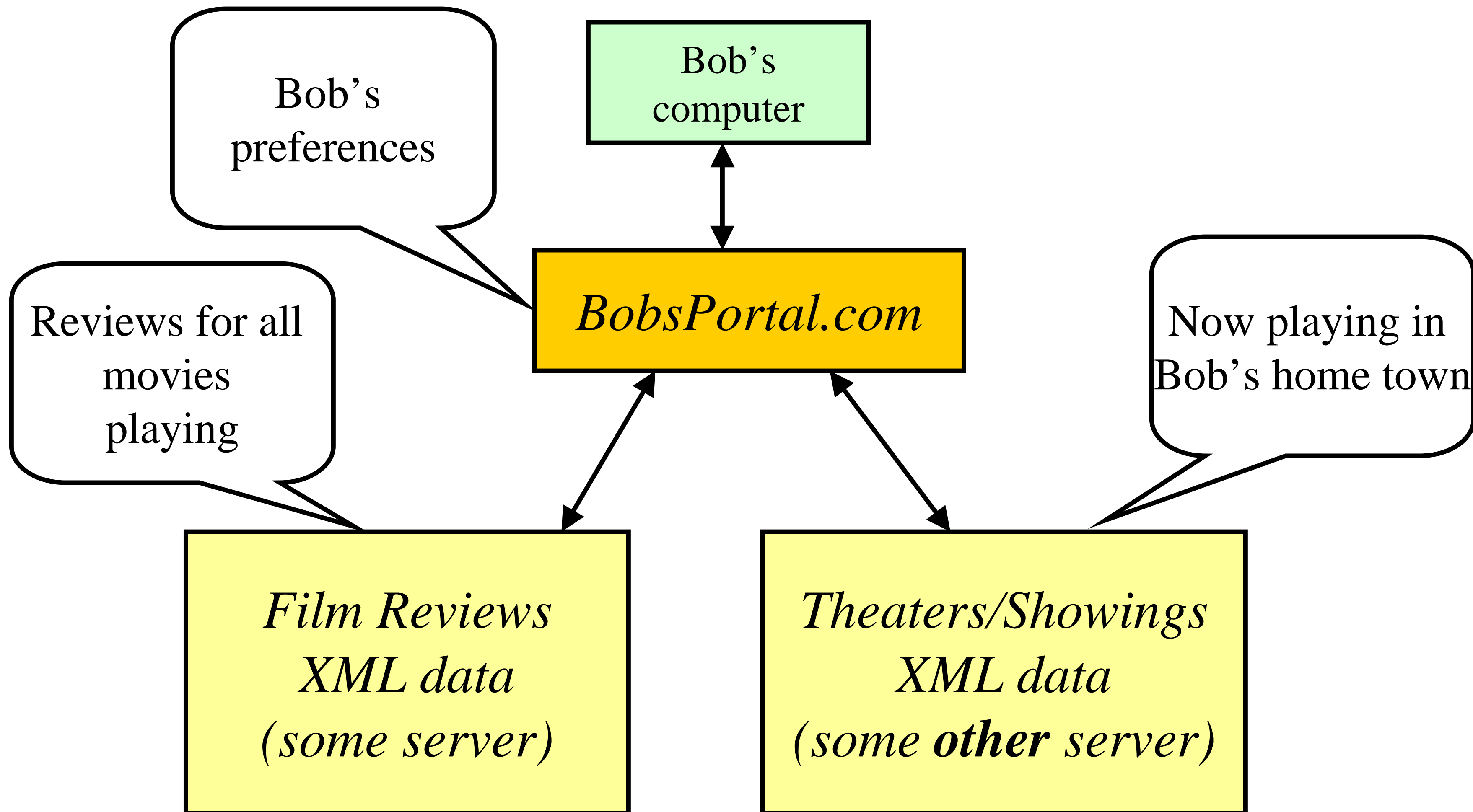
The Internet: Data rich, query poor

- Three major issues:
  - Content is often **stripped of its semantics**
    - **HTML** is just a page layout language
  - **Primitive** query capabilities
    - Can only query through form interfaces
      - **if** the data owner provides one
      - Usually simple I.R. based searches
  - **Localized** query capabilities
    - No queries involving multiple servers

Solutions

- **Now**: XML and semi-structured data will help with the lack of semantics
- **Real Soon Now**™: Query engines like *Niagara* will handle full-blown XML queries in web servers
- But, no distributed query framework in sight
  - *Why is this a problem?*

Bob's night at the movies…



..has to be distributed!

- We can’t ship all the data to one place and compute the query there…
  - **Technical** reasons
    - Waste of bandwidth
  - **Political** reasons
    - Data ownership and control

Traditional distributed querying in four easy steps

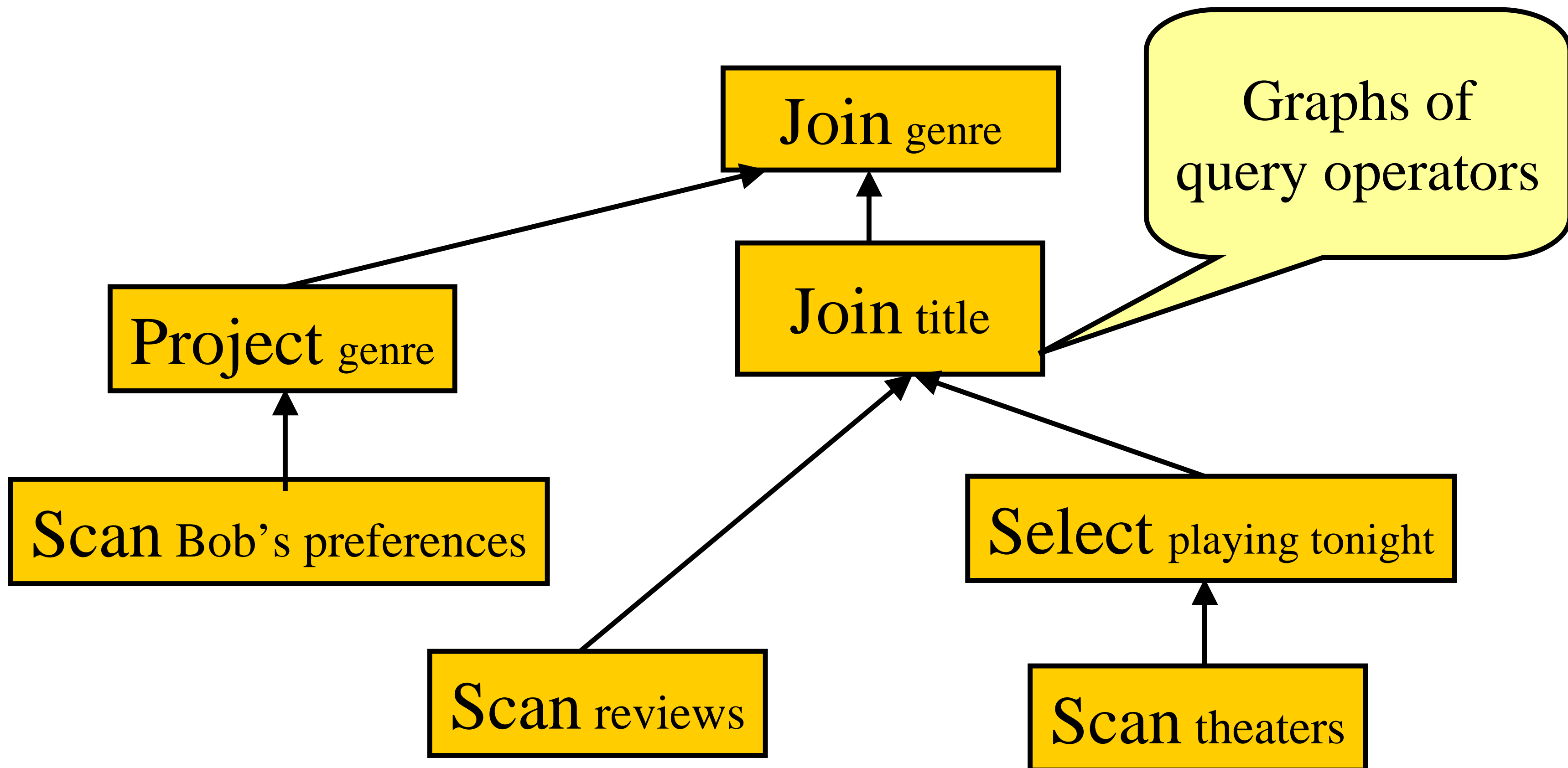
- The user (maybe using a GUI tool) comes up with a *textual* query. An **XQuery** example:

```
FOR $r IN document(“reviews”)//review,
    $g IN document(“preferences”)//genre,
    $s IN document(“showings”)/
                   showing/[date=“May 18”]
WHERE $r/genre = $g AND $r/title = $s/title
RETURN <film>
          $r/title $r/rating $r/theater
       </film>
```

2. Optimization

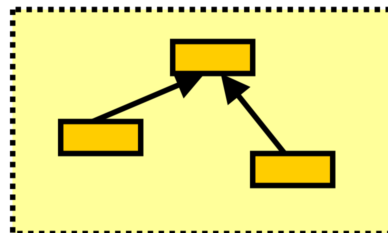
- The server receiving this query (in Bob's case, *BobsPortal.com)* is called the ***coordinator***
- Query execution may be distributed, but query ***optimization*** is centralized
- The coordinator optimizes the query by choosing the best logical (and subsequently physical) **query plan**

Query plans

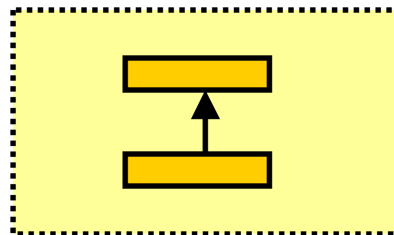


3. Deployment

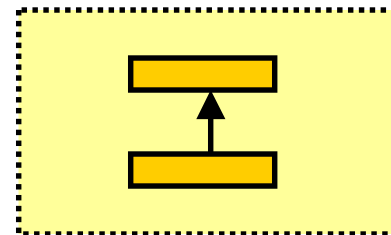
- The coordinator then decides **where** each operator is going to run and sends **subplans** to **apprentice sites**



BobsPortal.com



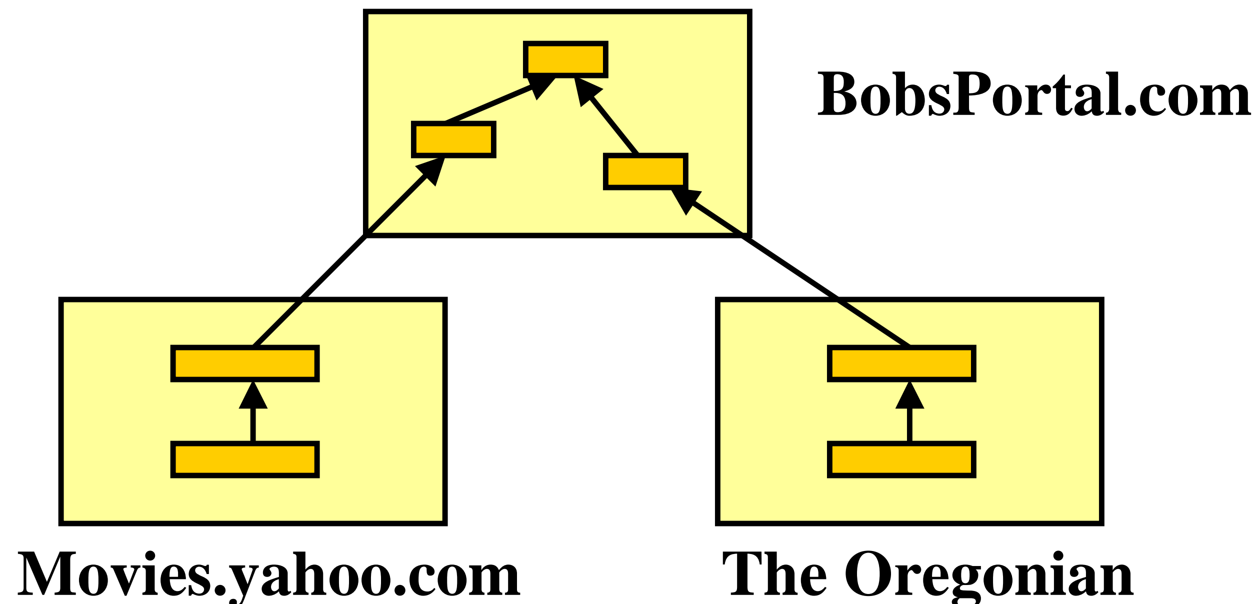
Movies.yahoo.com



The Oregonian

4. Execution

- Finally, the subplans start running **in parallel** on the various sites, and the coordinator oversees execution

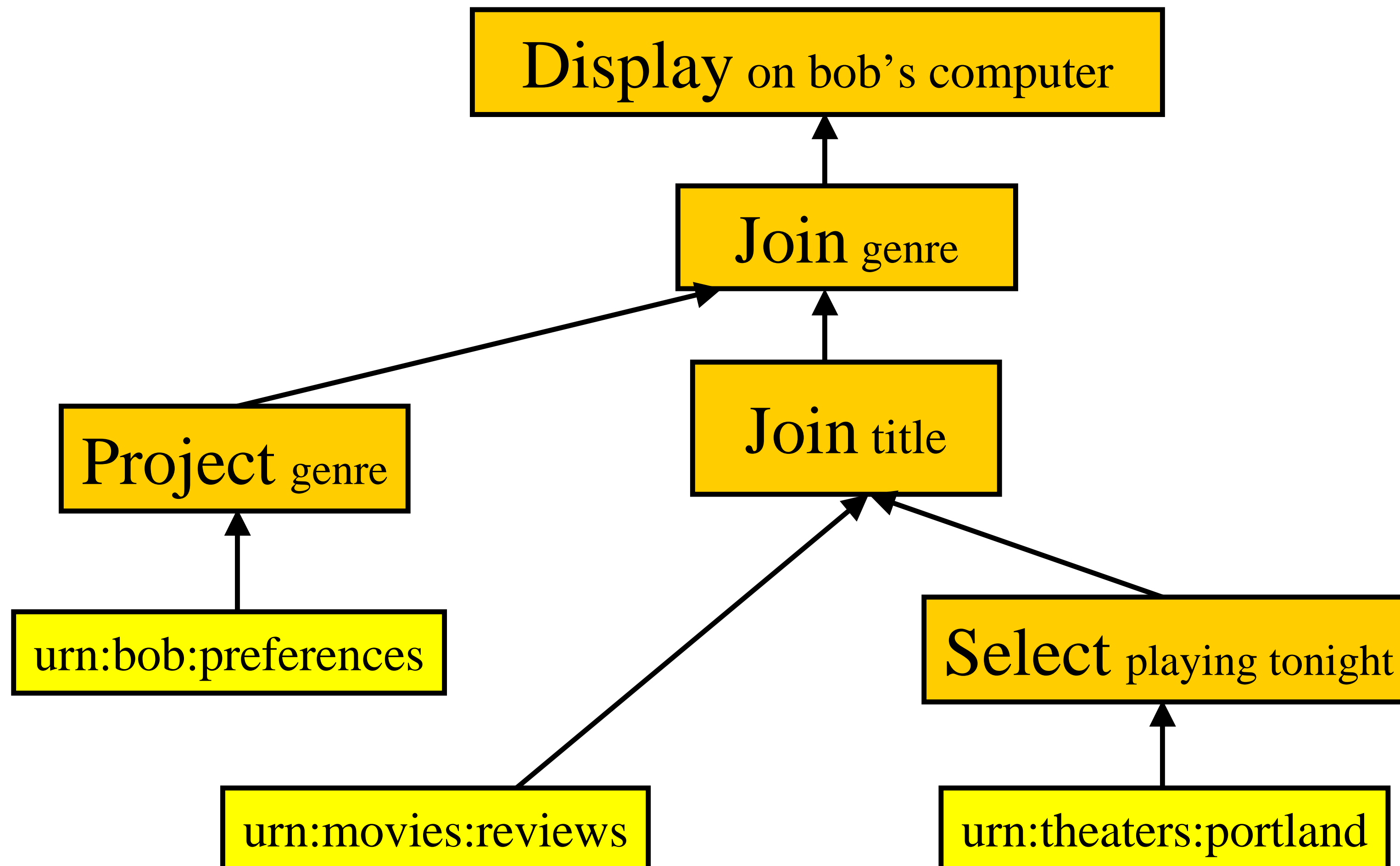


- Optimization depends on the coordinator having ***accurate, detailed statistics***
- But such statistics are hard to acquire and maintain:
  - (Replicated) Resource locations
  - Server load, Network bandwidth / latency
- Hard to orchestrate pipelined execution
  - All sites must be up and talking to begin execution

Mutant query plans

- Query as object: operators + XML data / URLs / URNs
- A server can ***mutate*** a Q.P. by:
  - resolving some URNs ➨ URLs ➨ data
  - evaluating subplans, and inserting results in their place
- Route to next server
  - count on best-effort at each site

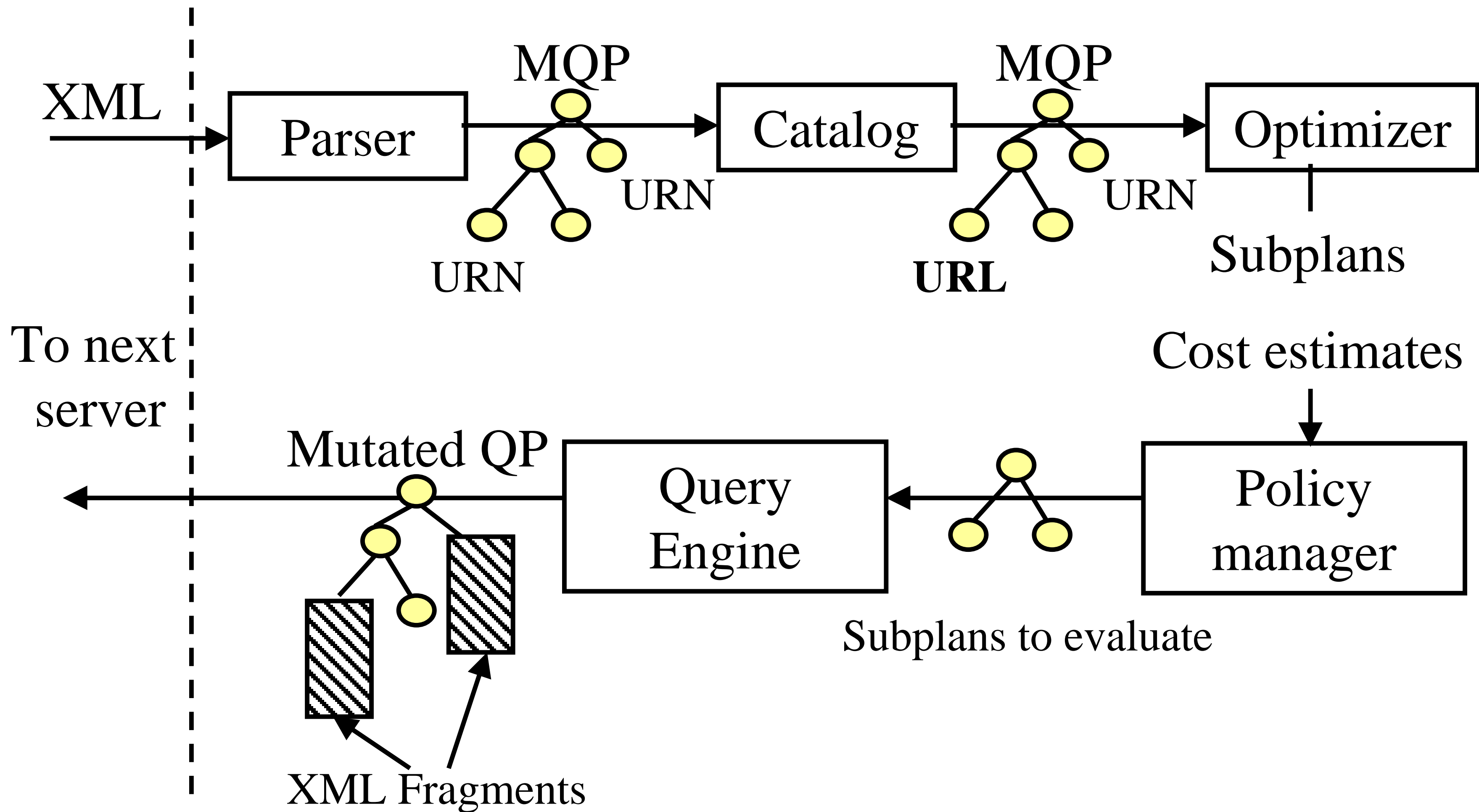
Bob’s own mutant query plan



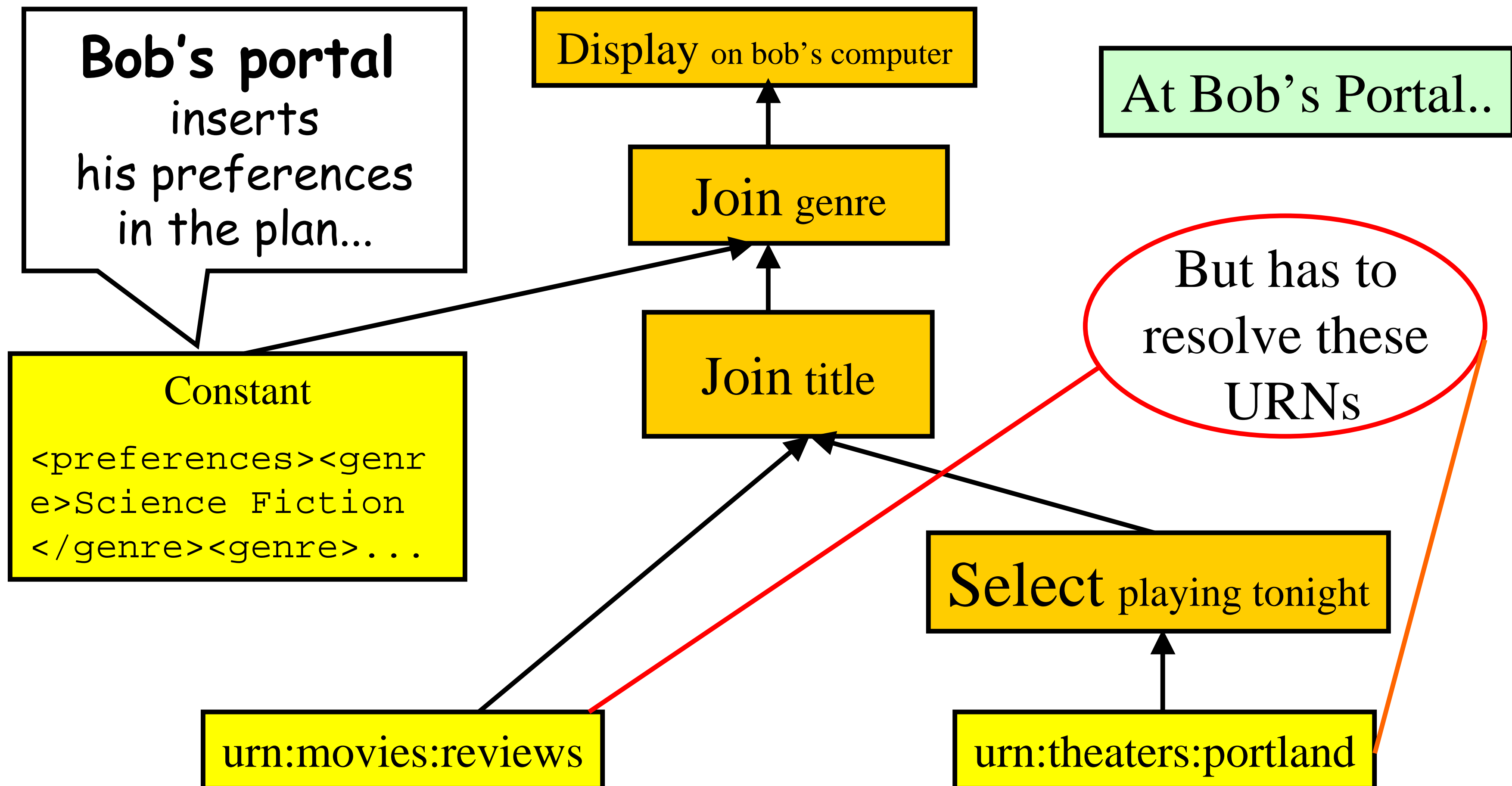
Decisions, decisions...

- A local **metadata catalog** maps abstract resources (URNs) to either:
  - concrete URLs
  - ***other servers*** that can resolve the URN
- A policy manager uses local metadata to decide **how much** of a query to evaluate, and **where** to send it next
- Current naive strategy:
  - evaluate all we can
  - send to server that can resolve the most remaining URNs

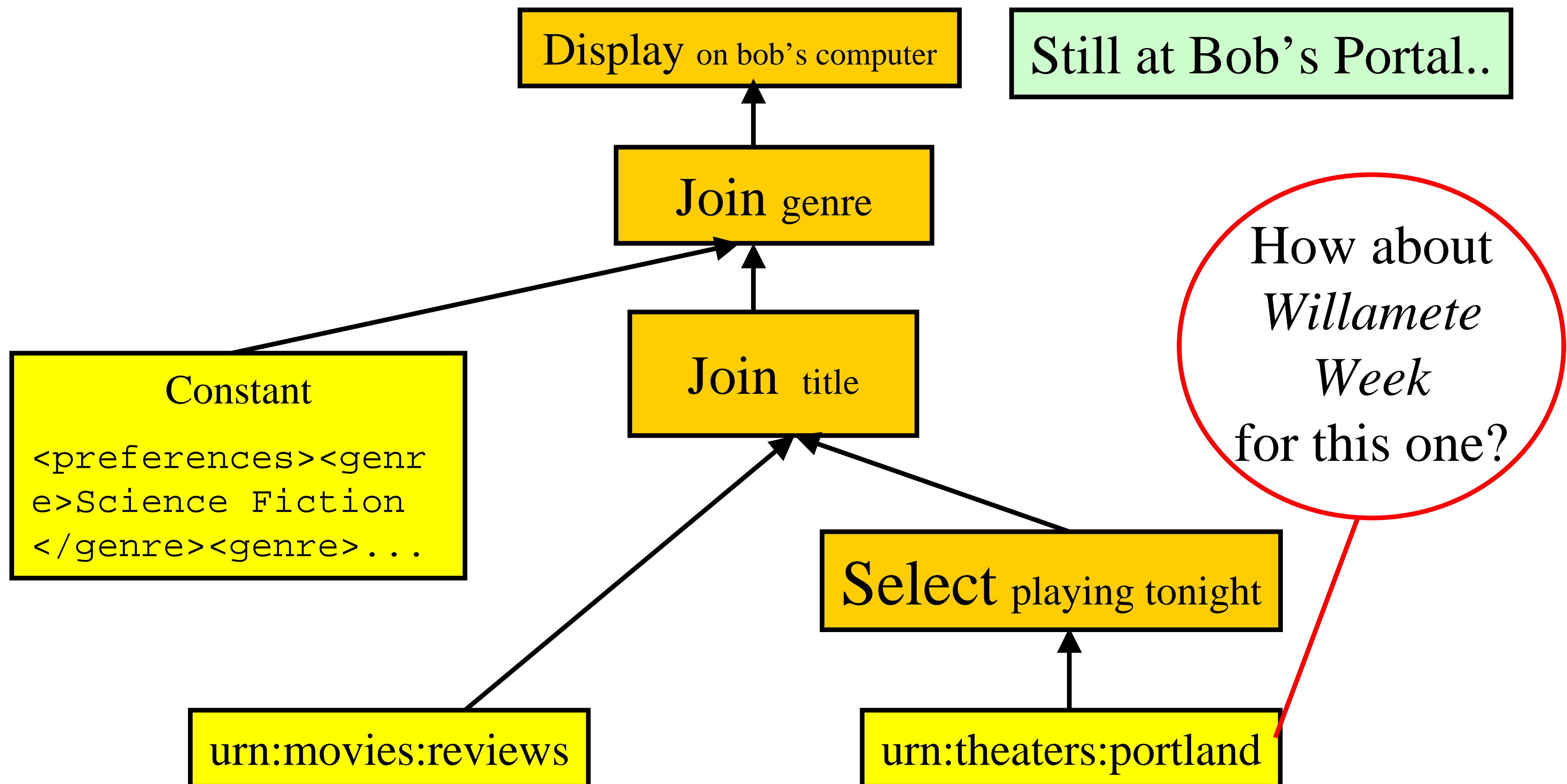
Architecture



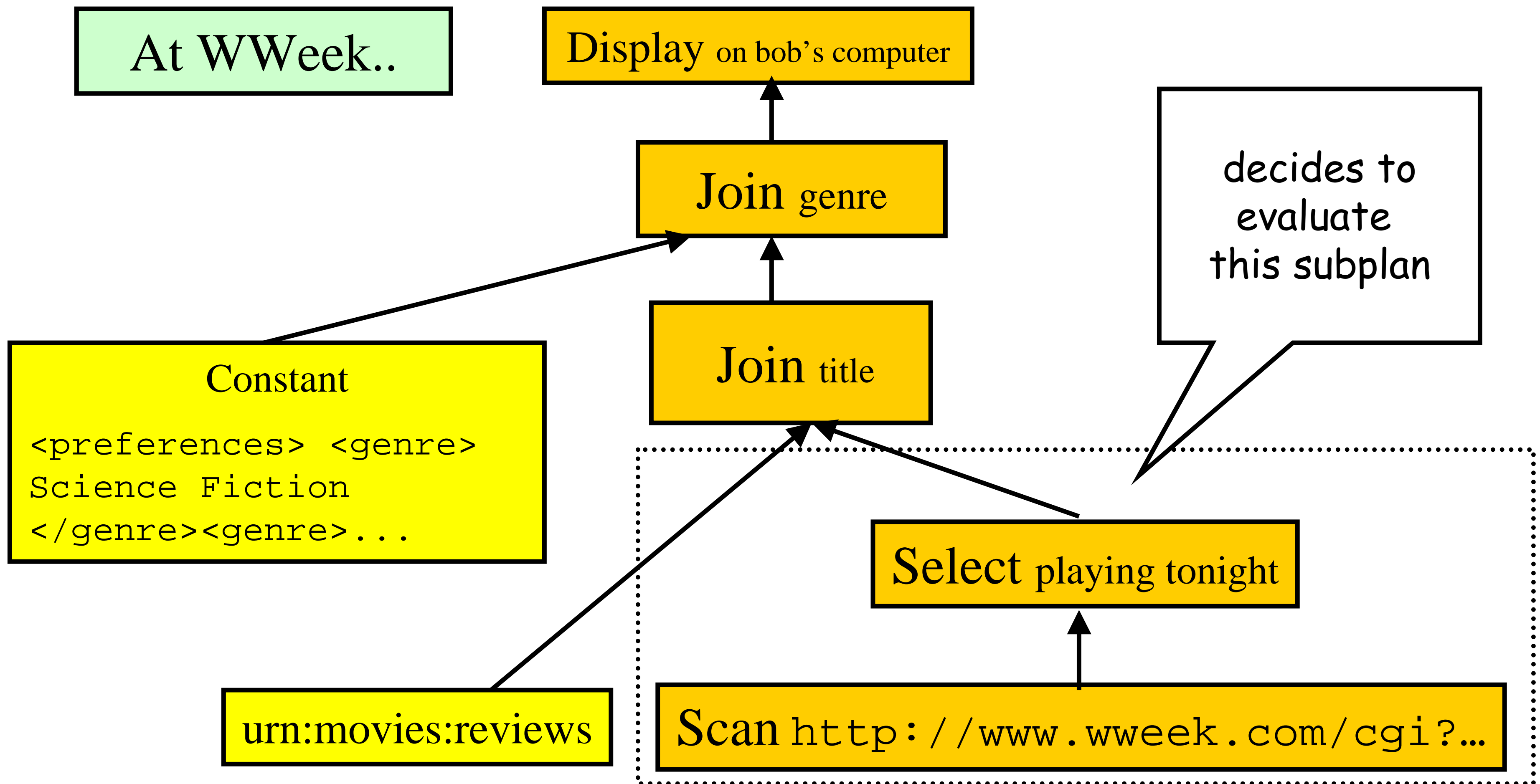
A tale of three servers



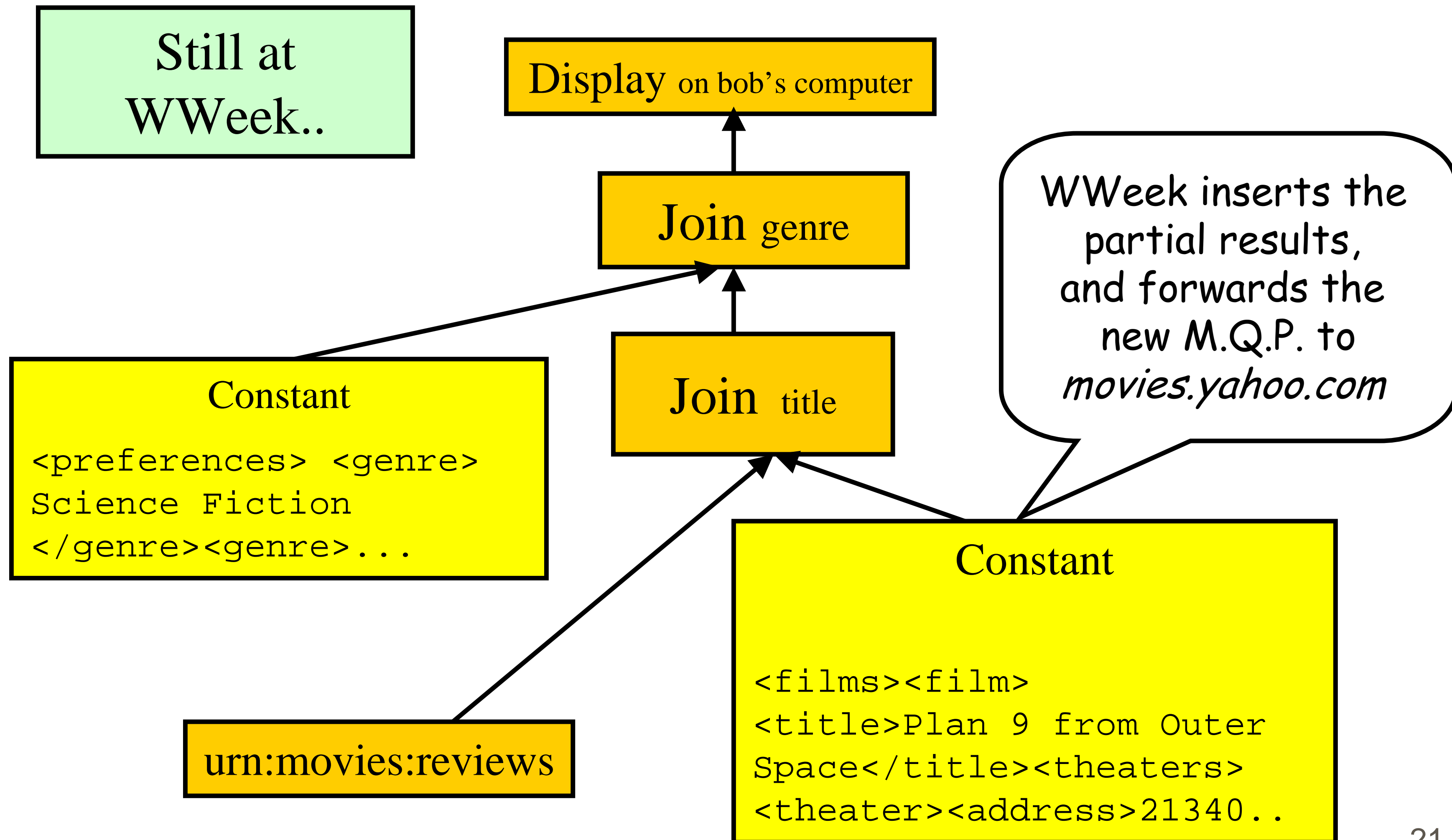
Resolving URNs



Mutating...



More mutating...



Yahoo can now evaluate the whole plan...

Display on bob’s computer

Constant
<films><film><title>
Plan 9 from outer space
</title> <rating>
10/10</rating><theater>
<address>21340..

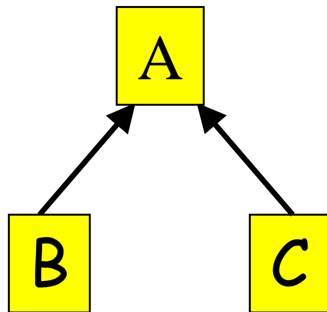
… and send the results to Bob!

Pipelined vs. Mutant execution: Tradeoffs

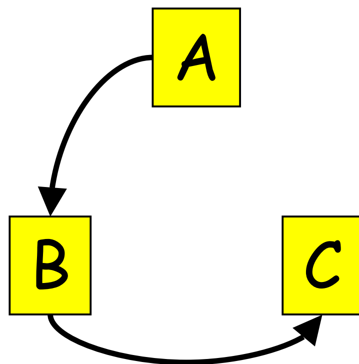
- We give up:
  - parallelism and pipelined execution (big)
- We gain:
  - robustness
  - site autonomy
  - ability to re-optimize on each server
  - reduced coordination overhead

Performance expectations

Pipelined plan



Mutant plan



A
B
C

Better latency

Resources are released earlier

A
B
C

Resources x Time "area" may be smaller

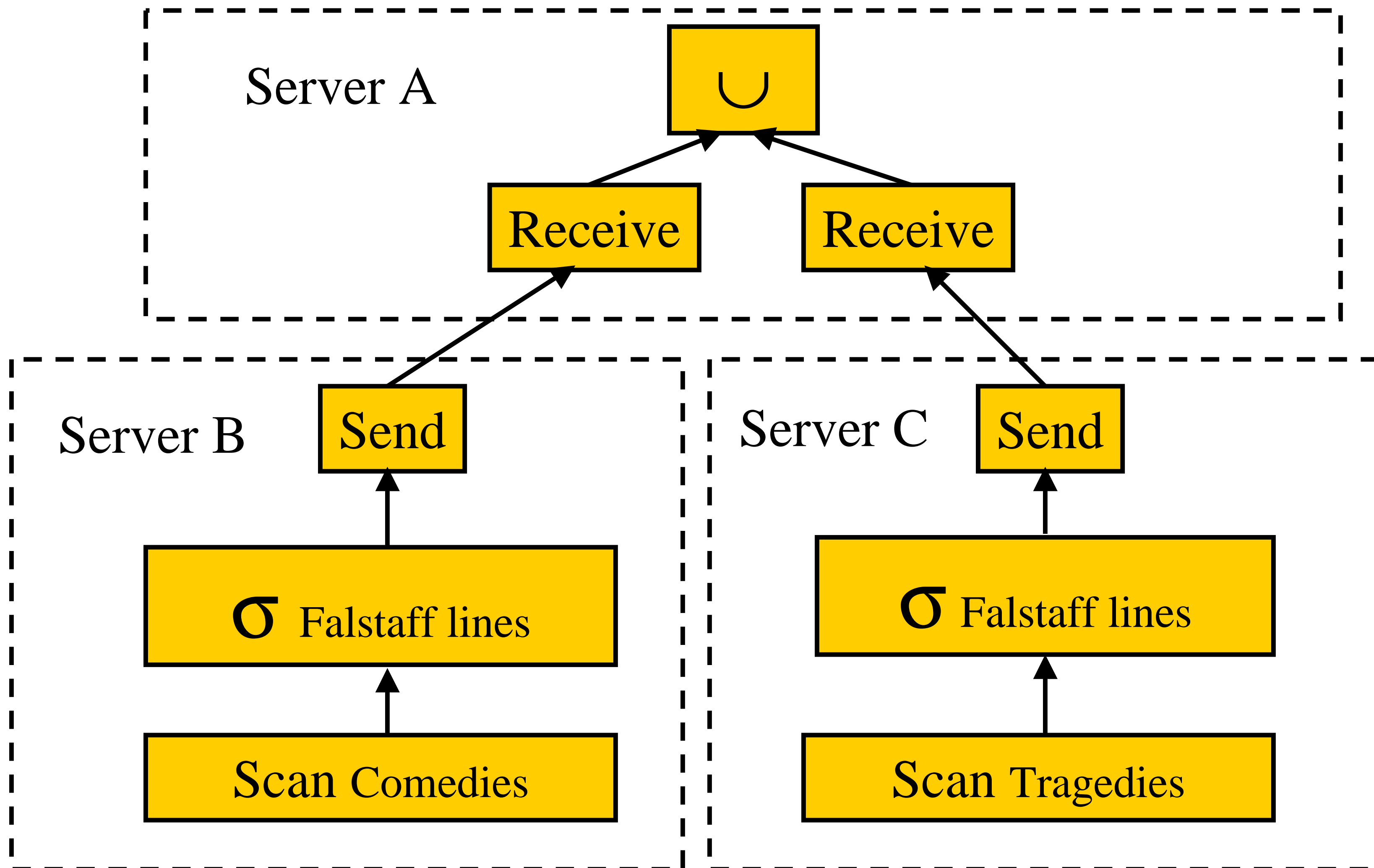
Time

Some initial results

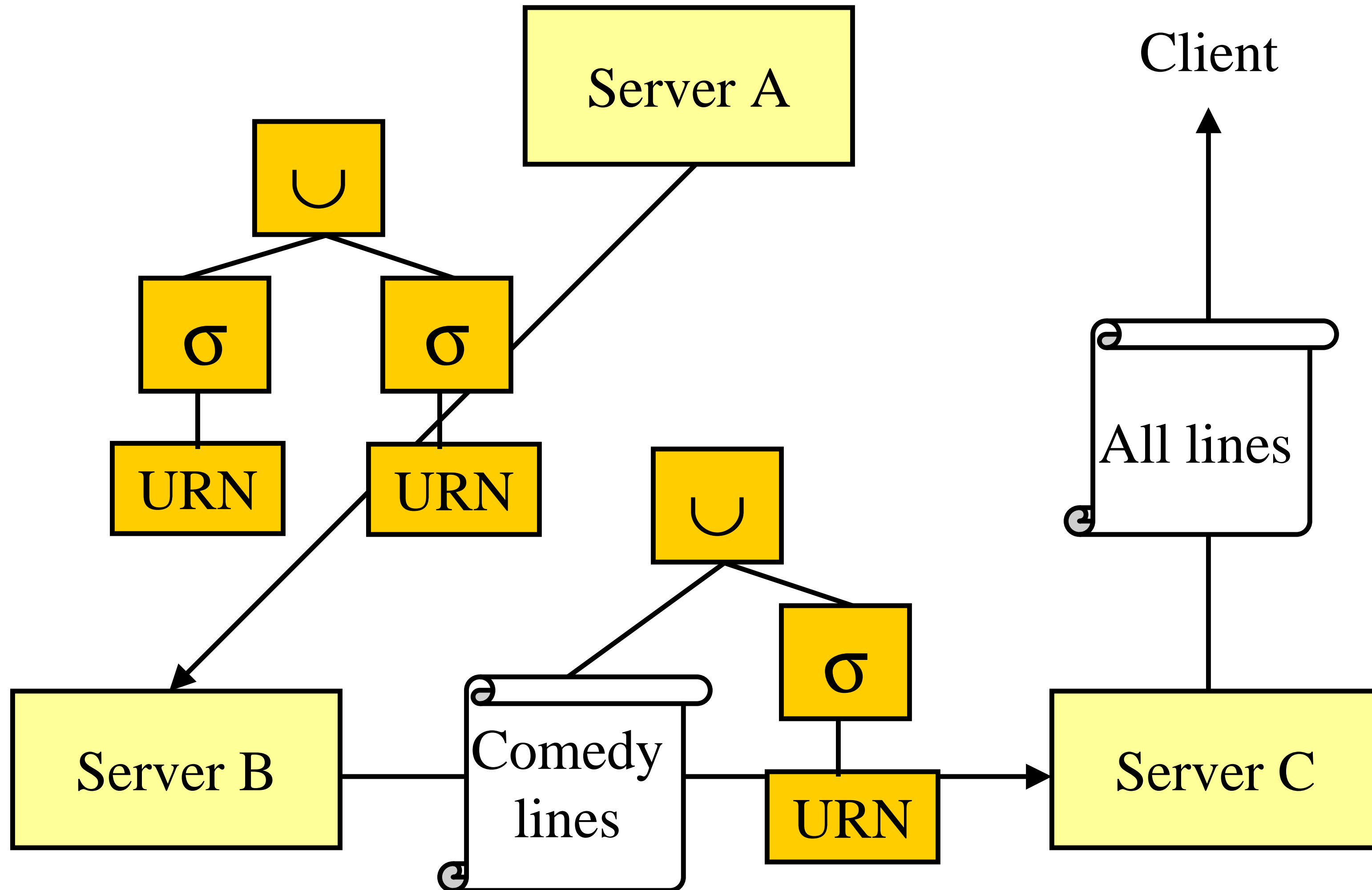
- We have extended ***Niagara*** with initial support for both mutant and pipelined distributed plans
- We tested manually optimized mutant and pipelined query plans

- The query finds all the lines of Sir John Falstaff in all the works of Shakespeare
- Three identical servers
  - server **A** is the coordinator
  - server **B** stores all the comedies
  - server **C** stores histories and tragedies
- Two scenarios, with server C having normal or high load

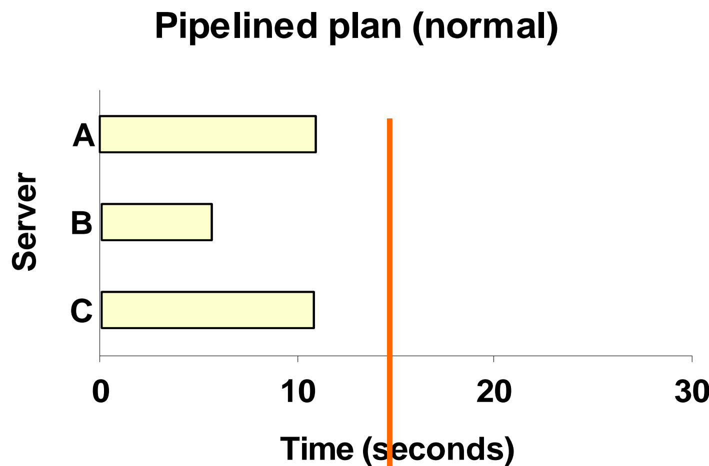
Pipelined plan

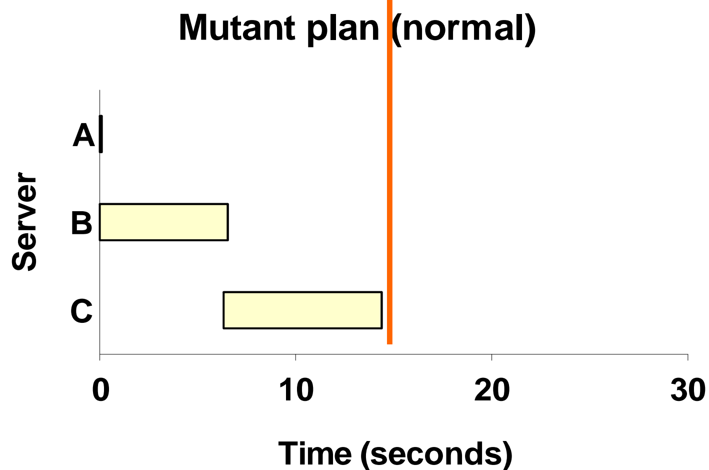


Mutants



Results (normal load)



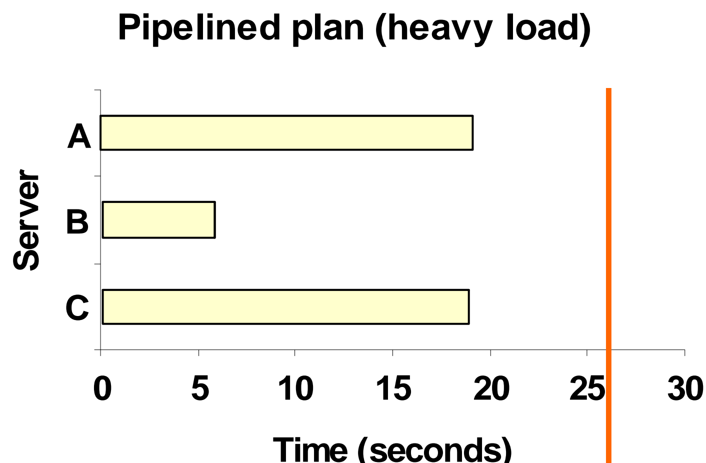


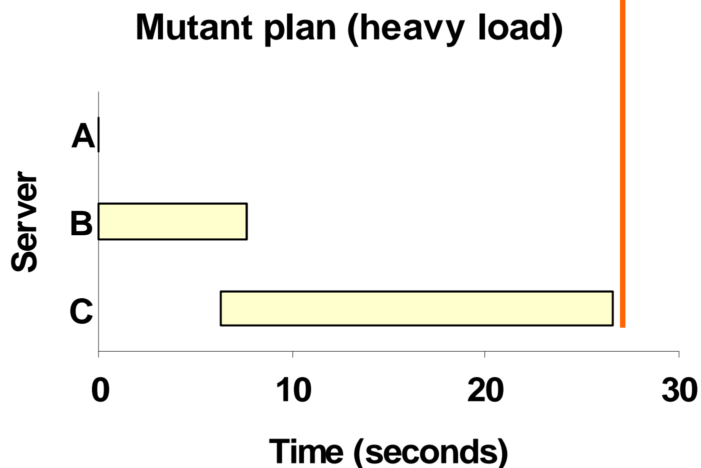
Latency	Footprint
10.95sec	27.28sec

Latency	Footprint
13.85sec	**14.56sec**

Less than
B + C
for pipelined !

Results (heavy load on C)





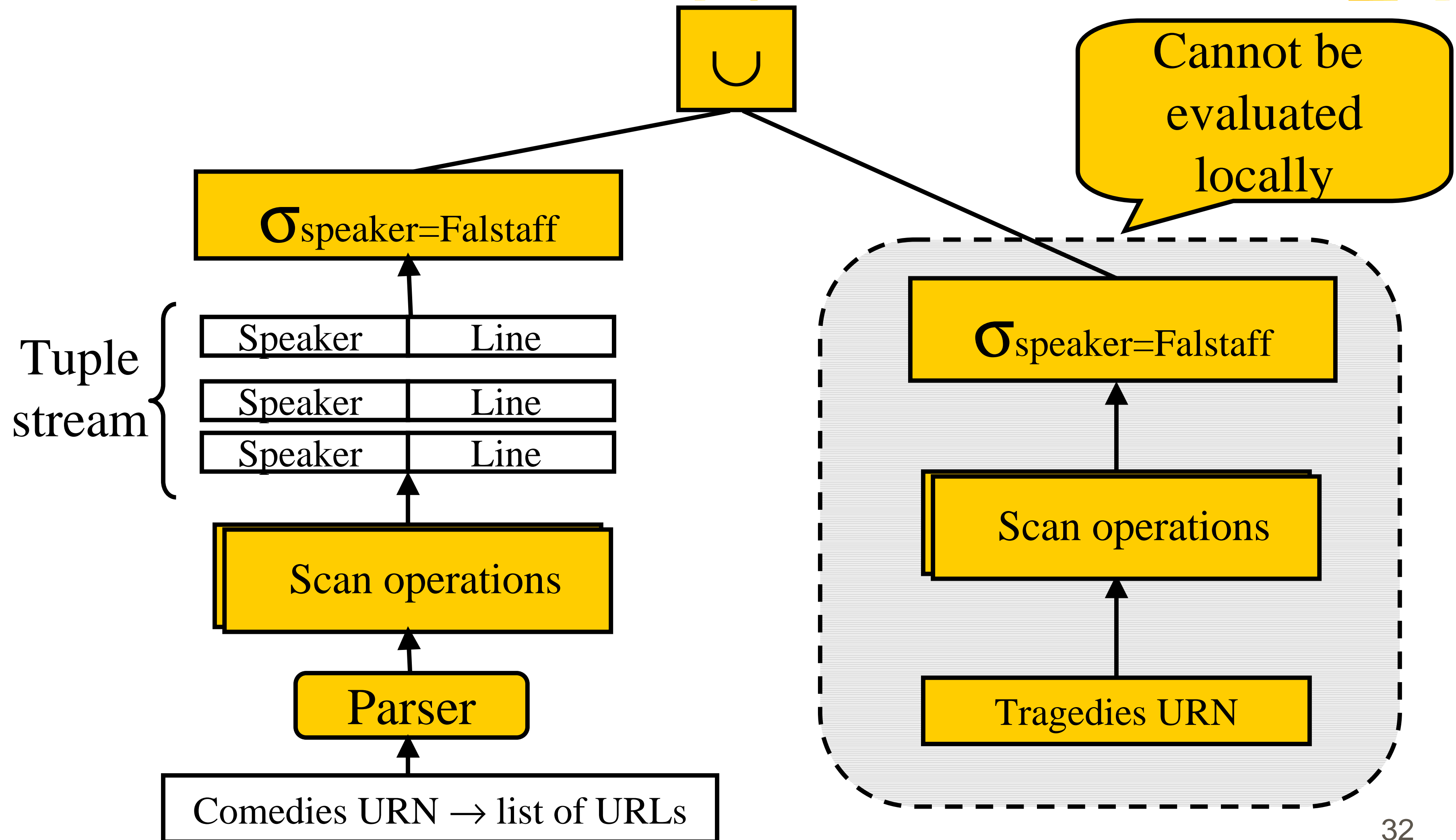
Latency	Footprint
19.11sec	43.59sec

A has to wait for C to finish!

Latency	Footprint
26.06sec	**28.03sec**

- Parachute queries (Bonnet and Tomasic)
  - Partially evaluate a query even when some sources are unreachable
- ObjectGlobe (Braumandl et al.)
  - Centralized metadata maintainance
  - distributed state vs. local state
- ubQL (Sahuguet)
  - MQP similar to “recruiting” strategy
  - separate deployment and execution phases
- Mobile agents

Partial evaluation of example plan



Mutated plan, in XML

```
<?xml version="1.0"?>
<!DOCTYPE plan SYSTEM "queryplan.dtd">
<plan top="union">
  <union id="union" input="from_comedies from_tragedies"/>
  <constant id="from_comedies">
    <stream>
      <tuple><elt><LINE>Now, Master Shallow, you'll complain of me to the king?</LINE></elt></tuple>
      <tuple><elt><LINE>But not kissed your keeper's daughter?</LINE></elt></tuple>
      <tuple><elt><LINE>I will answer it straight; I have done all this.</LINE></elt></tuple>
      <!-- . . . Rest of result omitted . . . -->
    </stream>
  </constant>
```

Mutated plan, in XML (cont.)

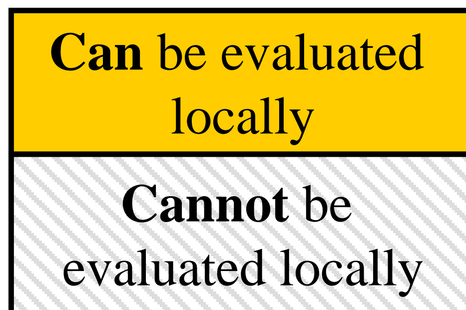
```
  <resource id="tragedies”  urn="urn:niagara:tragedies"/>
  <scan id="speech" regexp="PLAY.ACT.SCENE.SPEECH"
        input = "tragedies"/>
  <scan id="speaker" type="content”
        regexp="SPEAKER" input = "speech"/>
  <scan id="line" root="$speech"
        regexp="LINE" input = "speaker"/>
  <select id="from_tragedies" input="line">
    <pred op="eq">
      <var value="$speaker"/>
      <string value="FALSTAFF"/>
    </pred>
  </select>
</plan>
```

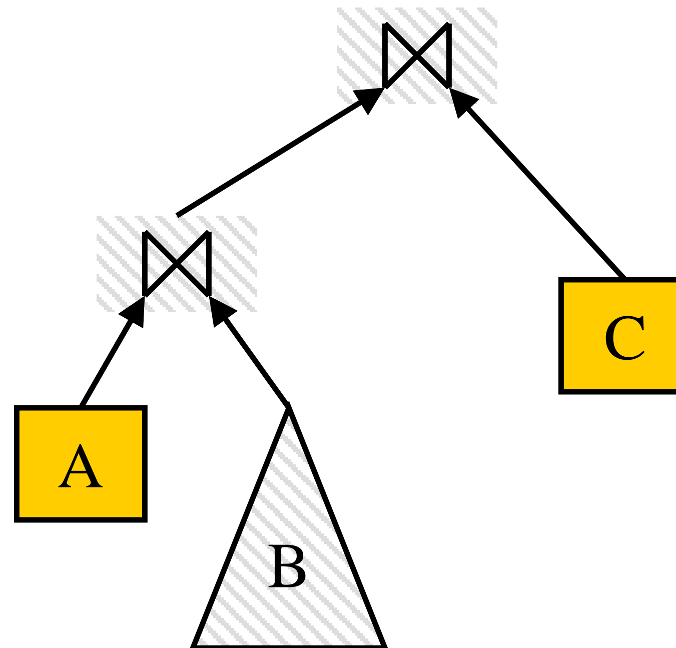
Mutant query optimization

- Mutant query plans bring up new optimization problems and opportunities
- Some ideas:
  - Consolidation
  - Deferment
  - Mutant strains
  - Result parking

Consolidation

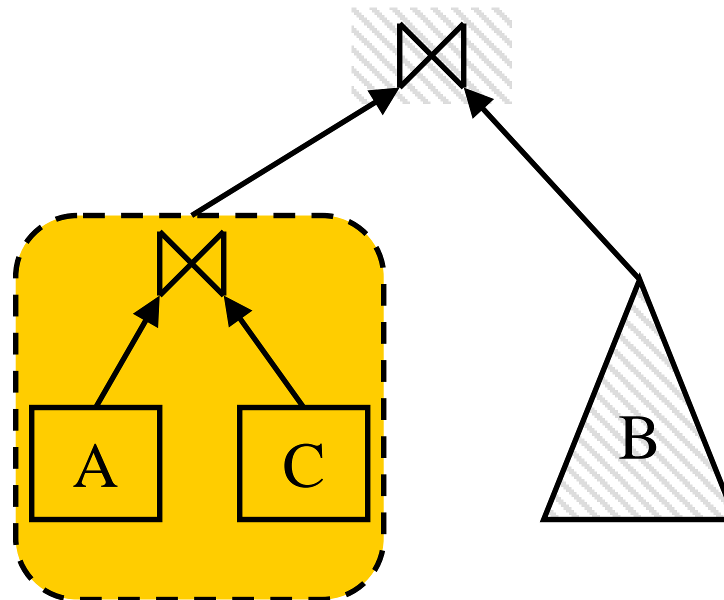
- A sub-plan can be evaluated locally only if **all** of its leaf nodes are available locally





Consolidation (cont.)

- Consolidation: reordering a plan so that more sub-plans become evaluable

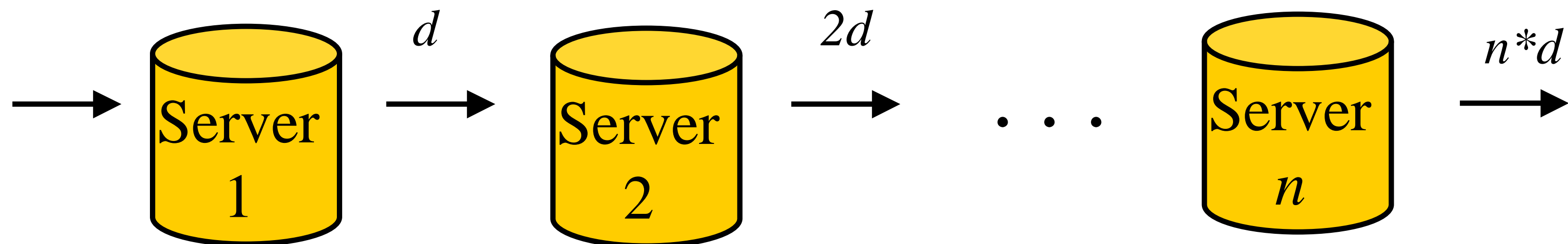


Can be evaluated locally
Cannot be evaluated locally

- Just because we *can* evaluate a sub-plan, does not mean we *should*
- If A $\bowtie$ C is a cartesian product (while A $\bowtie$ B $\bowtie$ C is not), evaluating it locally is the wrong thing to do, because it needlessly increases the size of the output plan.

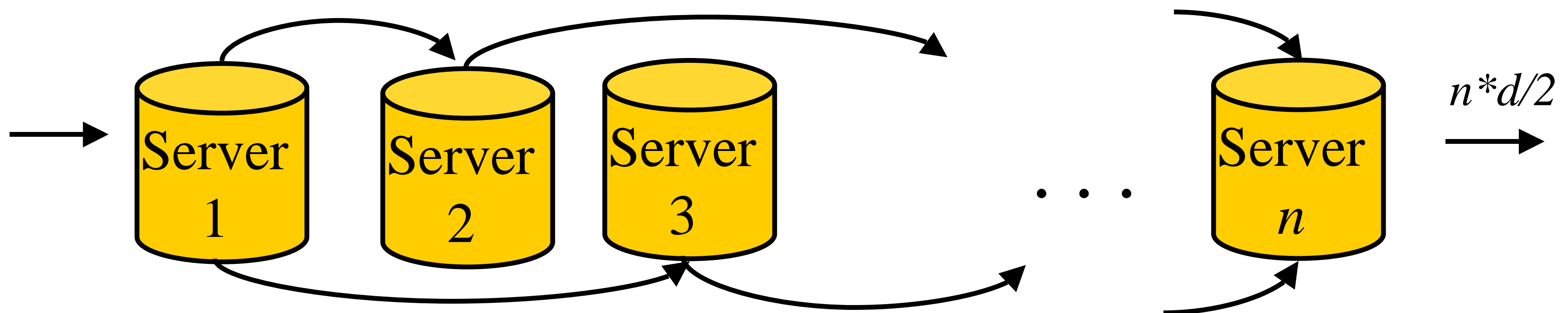
Mutant strains

- Reducing latency by fragmenting a plan into smaller plans, that are later joined together.
- Visiting *n* servers sequentially:



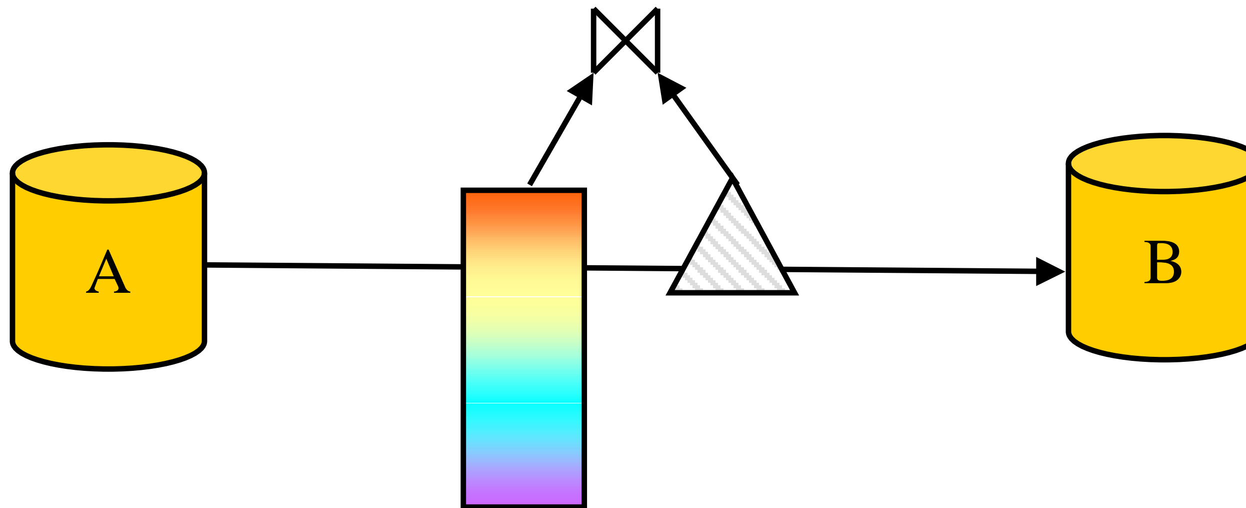
Mutant strains (cont.)

- Breaking the plan into two strains that visit alternate servers reduces the delay by half



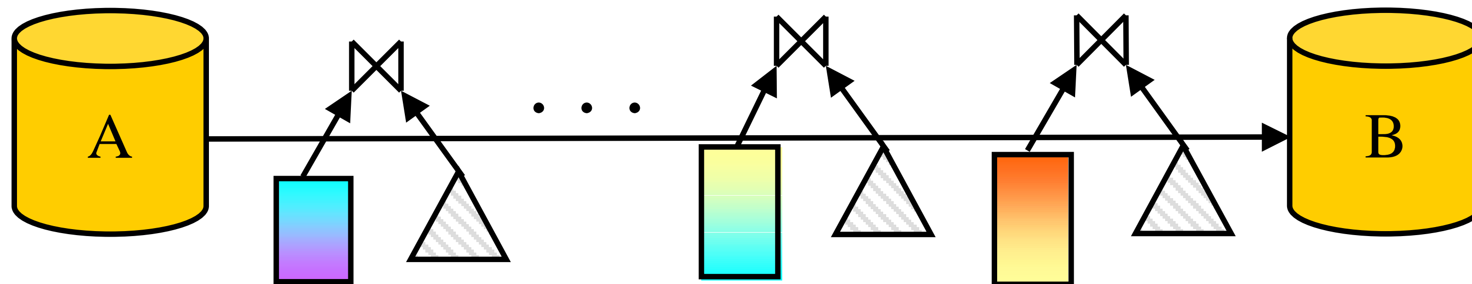
Getting some pipelining

- We can also use mutant strains to reduce delay by breaking the transmission of a large plan into a series of smaller plans. For example, instead of:



Partial results (cont.)

- We can partion the big join into an equivalent series of smaller joins:



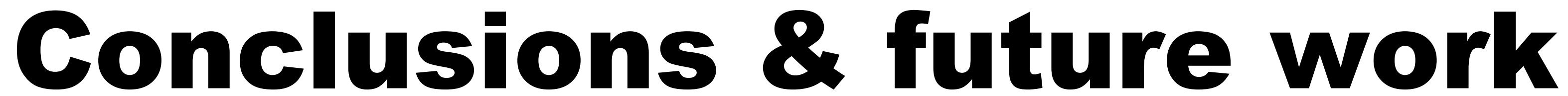
Result parking

- Suppose we know that the results of evaluating a sub-plan will not be used in the next few destinations of the MQP…
- We can “park” the results locally, and insert a temporary URL pointing to the current server in the mutated QP
- A subsequent server that can needs the results can retrieve them, or send the MQP back

There's more!

- Richer metadata
  - Collect and distribute metadata about server load and capabilities
- Controlled mutation
  - Give the user control over how MQPs are routed and processed (privacy and security concerns)
- Meta-mutant processing
  - triggering, load-balancing, caching

Conclusions & future work



- Mutant query plans part of a viable, scalable framework for distributed queries
  - no need for omnipotent, omniscient coordinators
- Future work
  - Implement our ideas for mutant query optimization in the Niagara framework
  - Connect with information discovery facilities