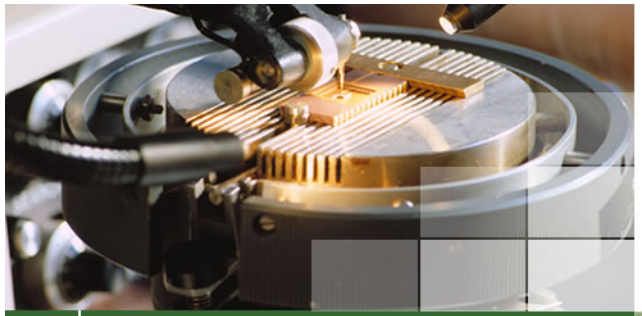


POLITECNICO DI MILANO

Dipartimento di
Elettronica e Informazione

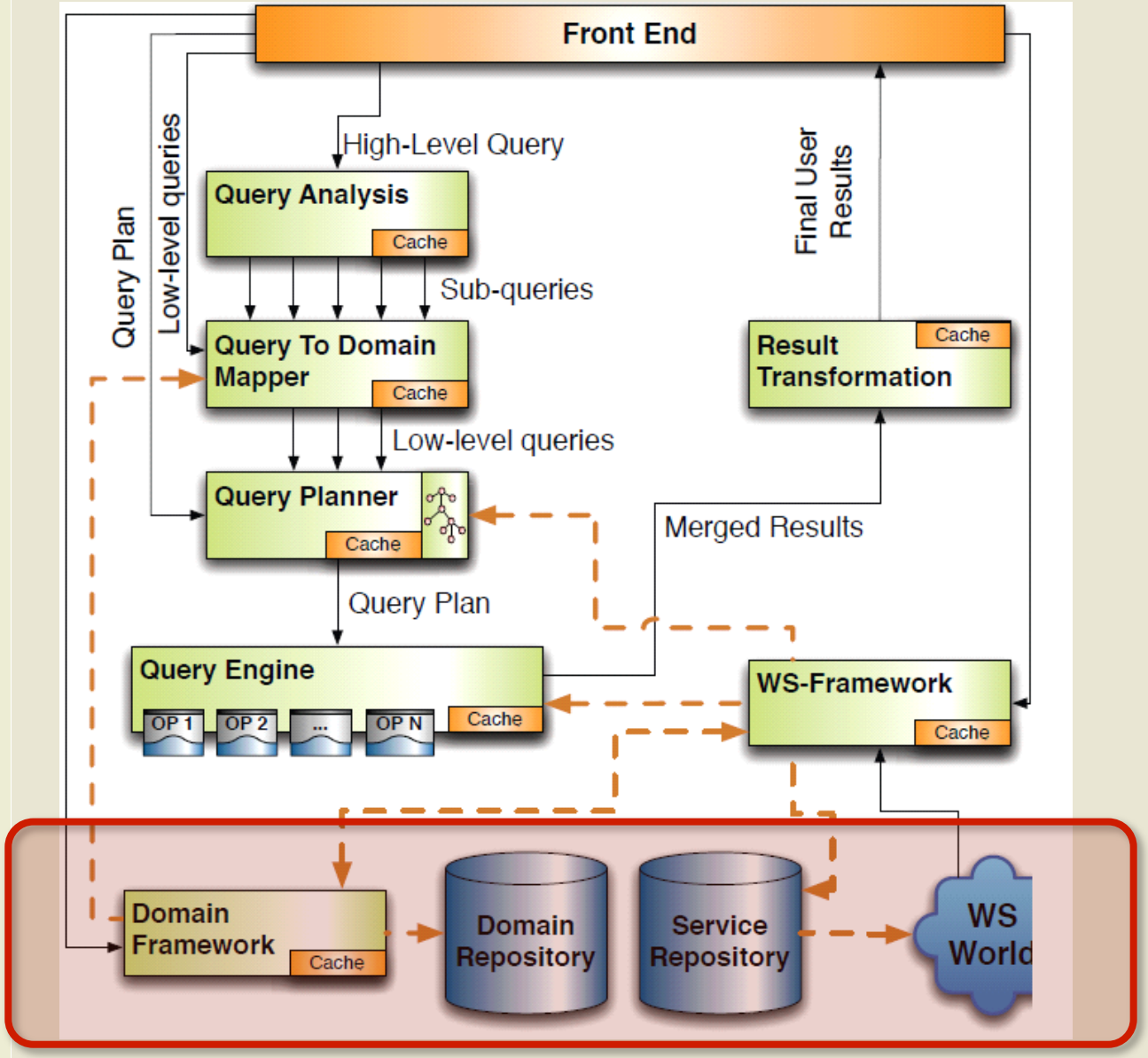
Search Computing: Query Execution Framework

Stefano Ceri

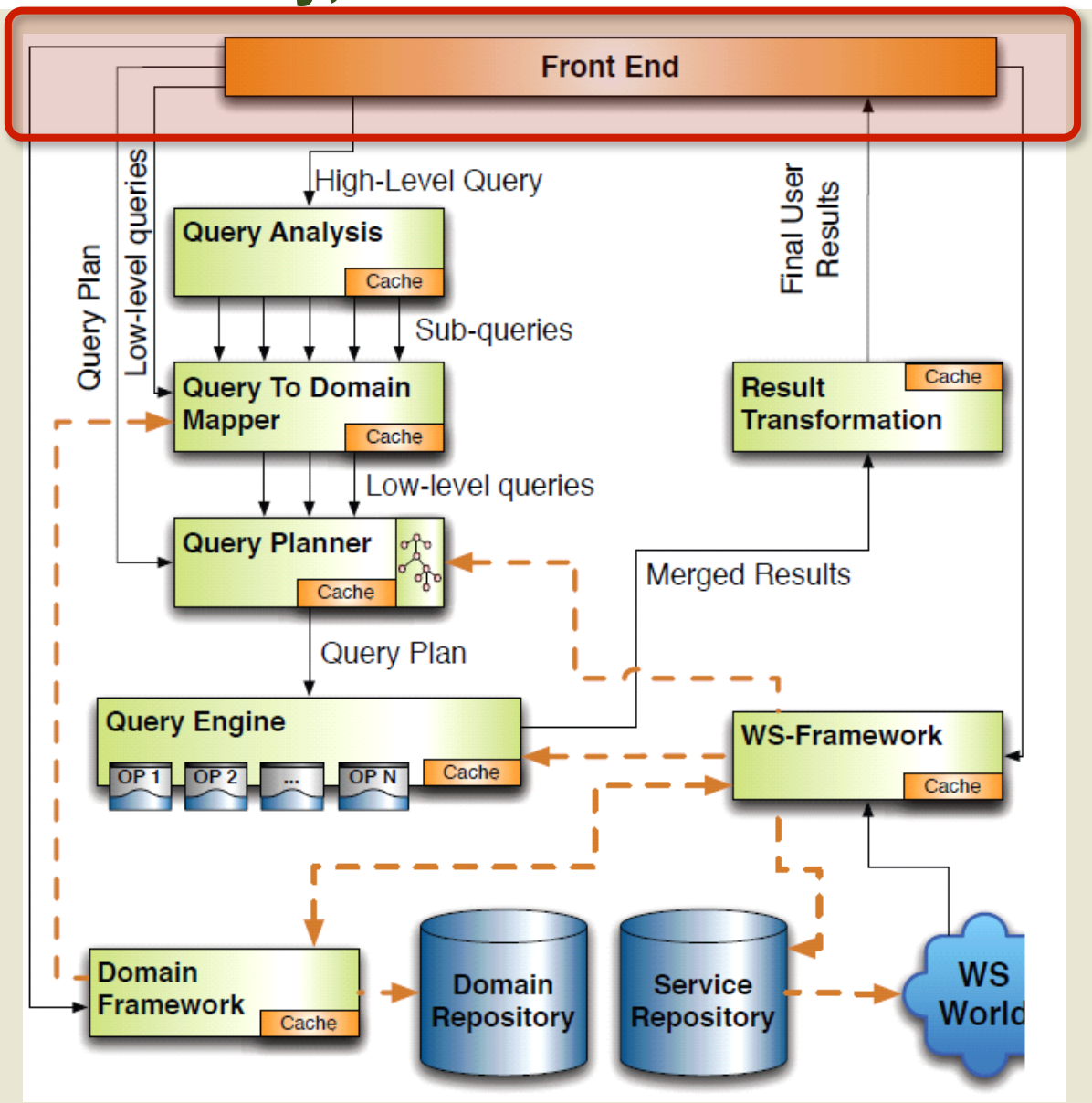
... and the SeCo Project Team

Adnan Abid, Mamoun Abu Helu, Davide Barbieri, Daniele Braga, Marco Brambilla, Alessandro Bozzon, Alessandro Campi, Davide Chicco, Emanuele Della Valle, Piero Fraternali, Nicola Gatti, Giorgio Ghisalberti, Davide Martinenghi, Marco Masseroli, Maristella Matera, Chiara Pasini, Silvia Quarteroni, Marco Tagliasacchi, Luca Tettamanti, Salvatore Vadacca, Serge Zagorac

That was the Service Framework



On Monday, focus on the user Interface



Liquid Queries

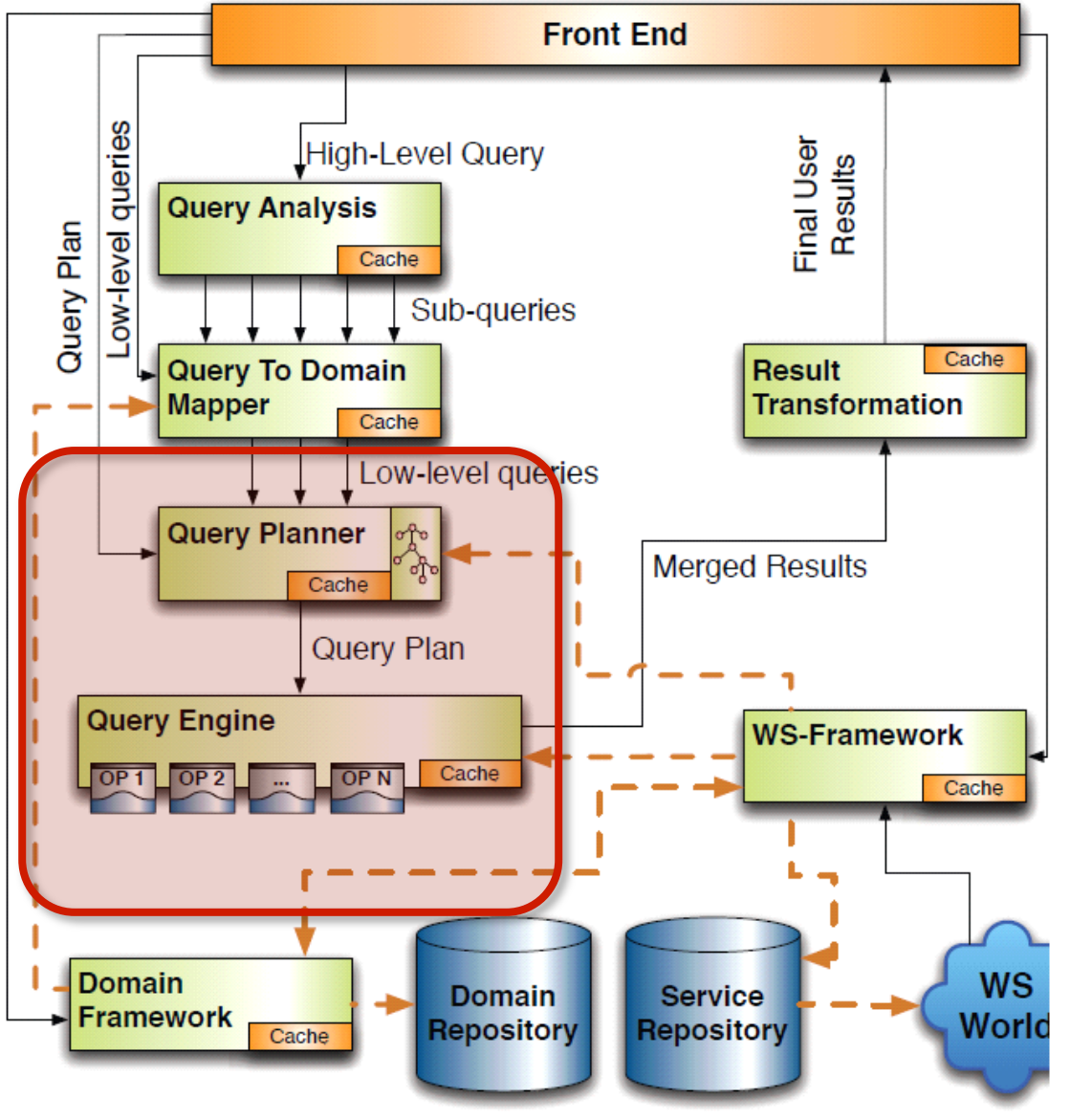
- **Intuitive formulation**
- **Exploratory search**:

  search as a **process**

  incremental formulation
- **Multi-dimensional visualization**:

  multiple views for results

Now, focus on: SeCo queries and the Query Engine



Query execution

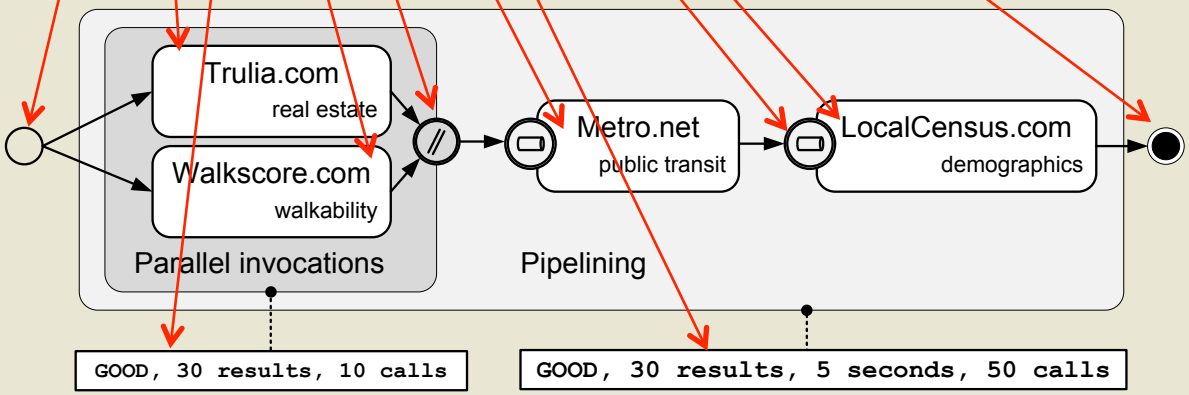
- **Query planner**:
  translates queries into query plans
- **Query engine**:
  executes query plans according to strategies

  Top-k
  Optimistic (good-k)
  Cost-driven

Search Computing = service composition "on demand"

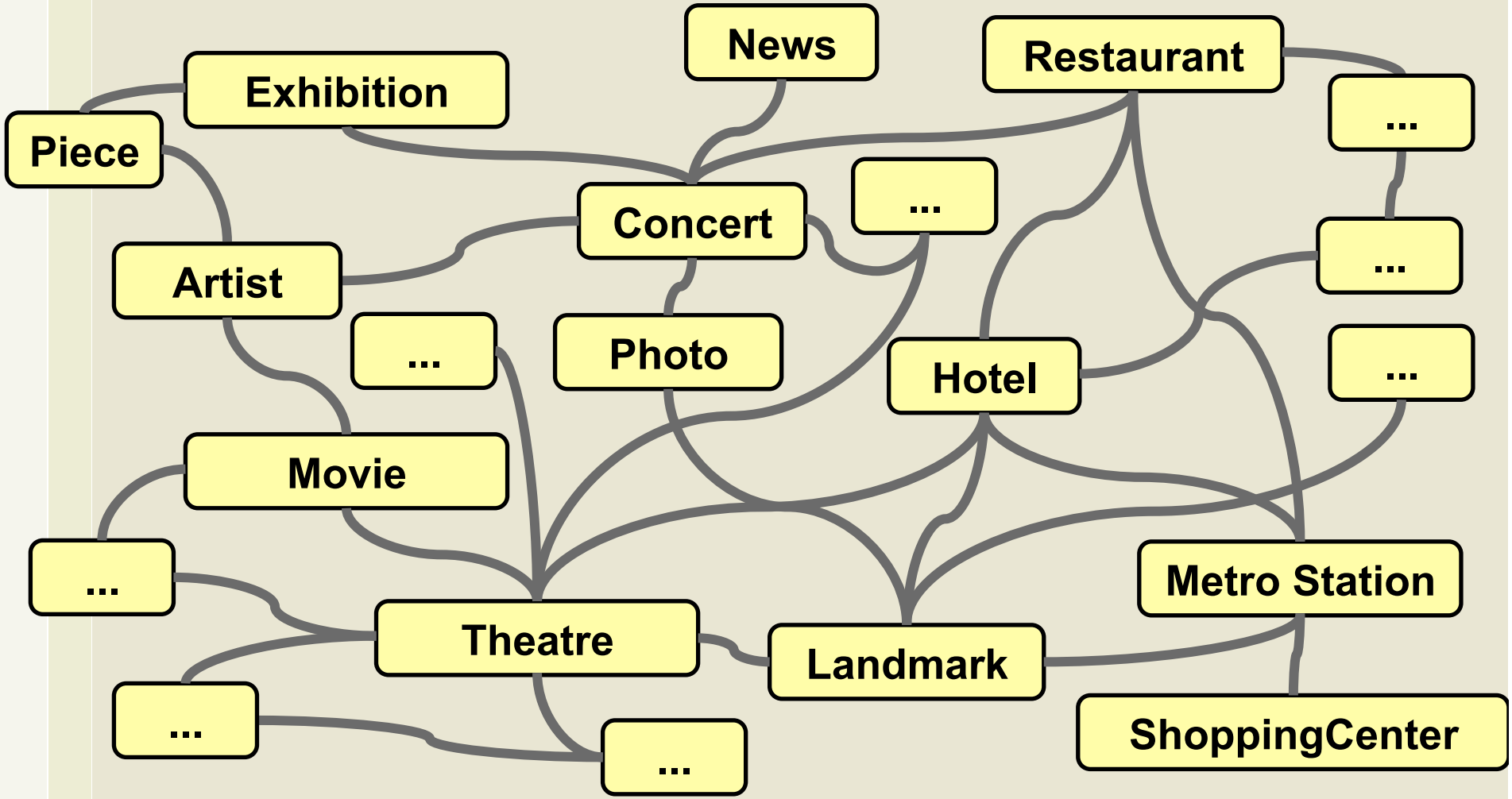
- Composition **abstractions** should emphasize few aspects:
  - service invocations
  - fundamental operations (parallel invocations, joins, pipelining, …)
  - global constraints on execution
- Data composition should be search-driven
  - aimed at producing **few top results** very fast

A house in a walkable area, close to public transportation and located in a pleasant neighborhood



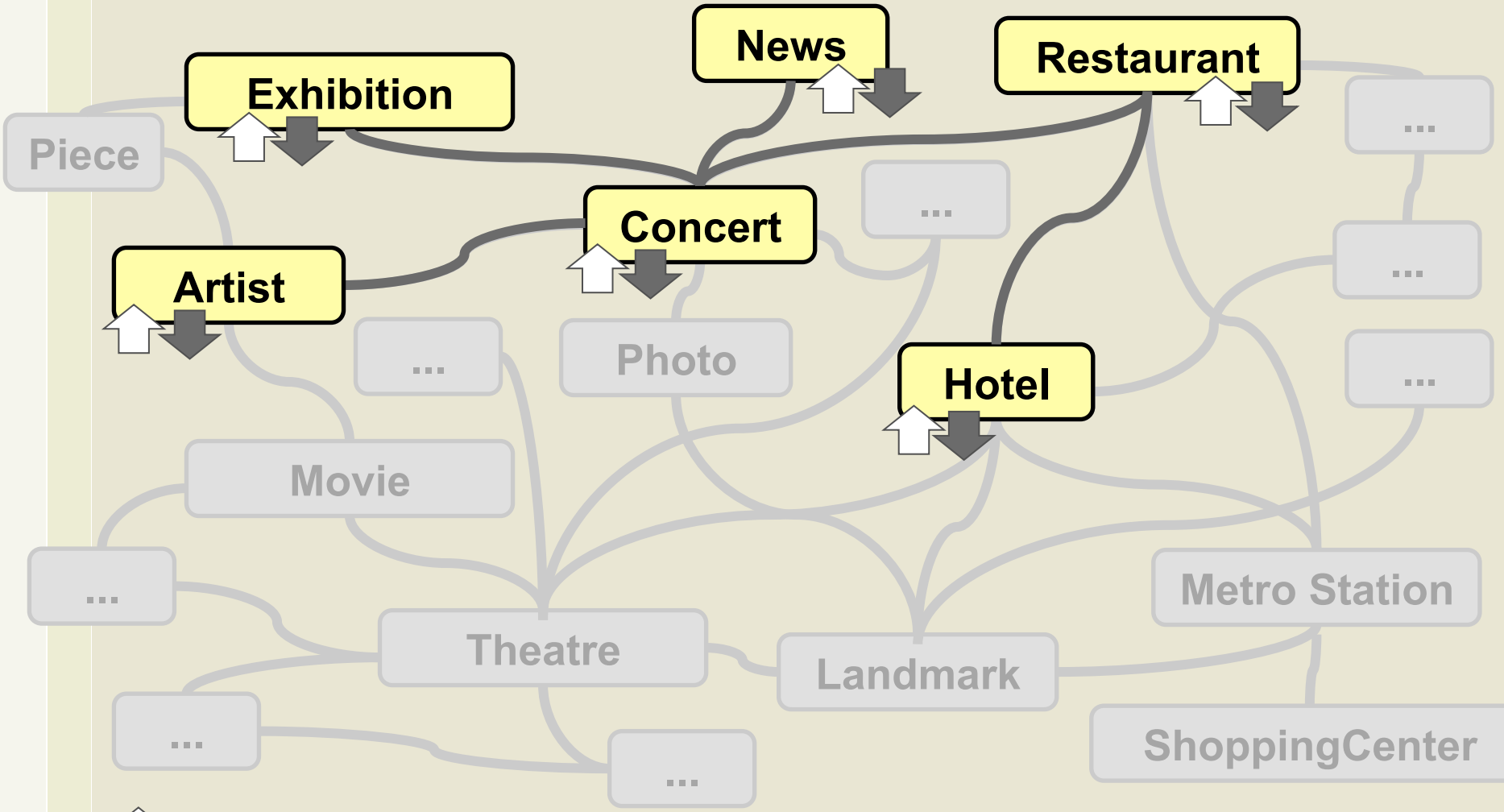
SeCo queries

- Navigation of a graph of connected data sources



Query formulation

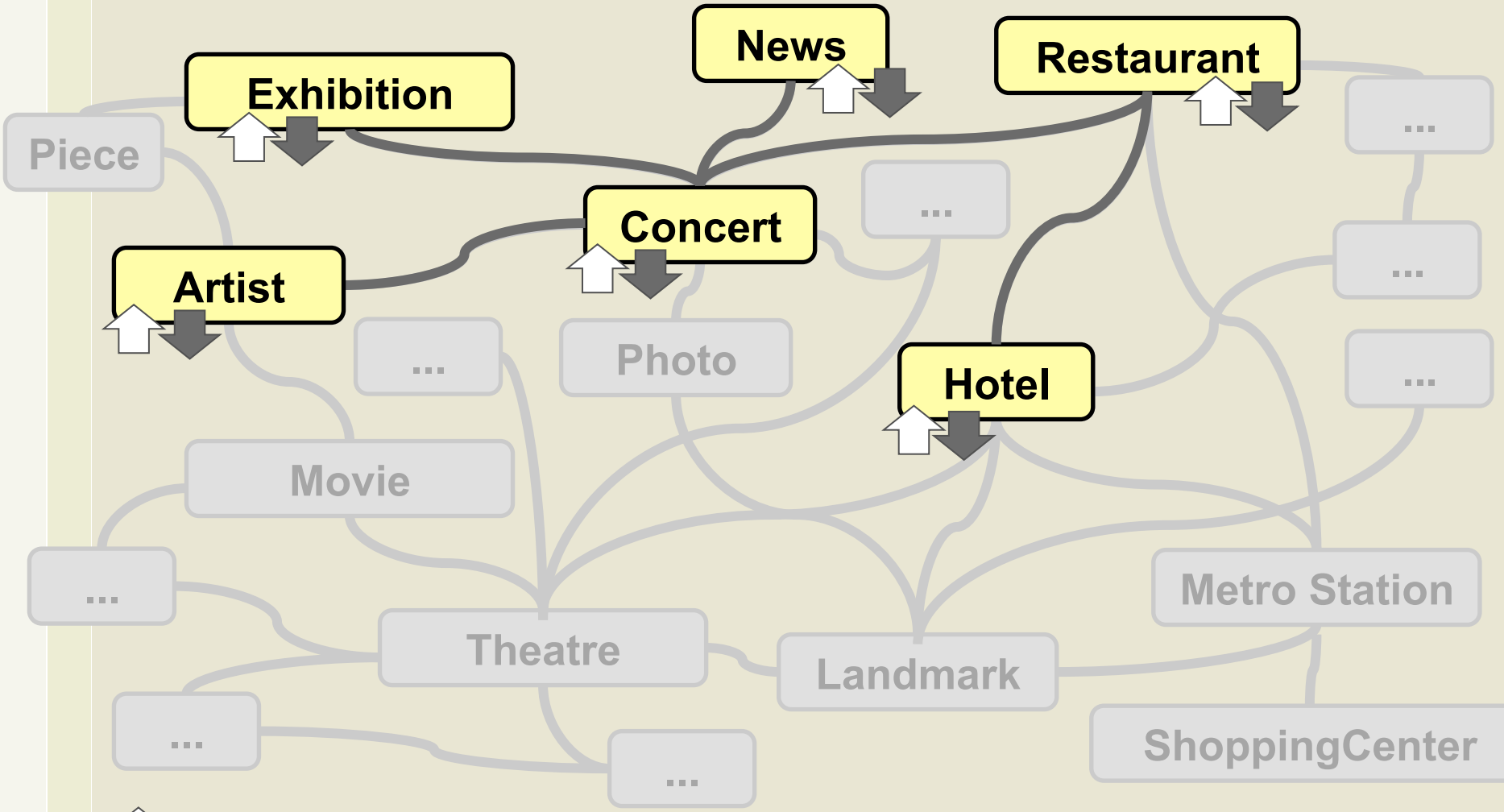
Consists in subsetting and parametrizing the graph



= inputs, outputs + GR = global ranking

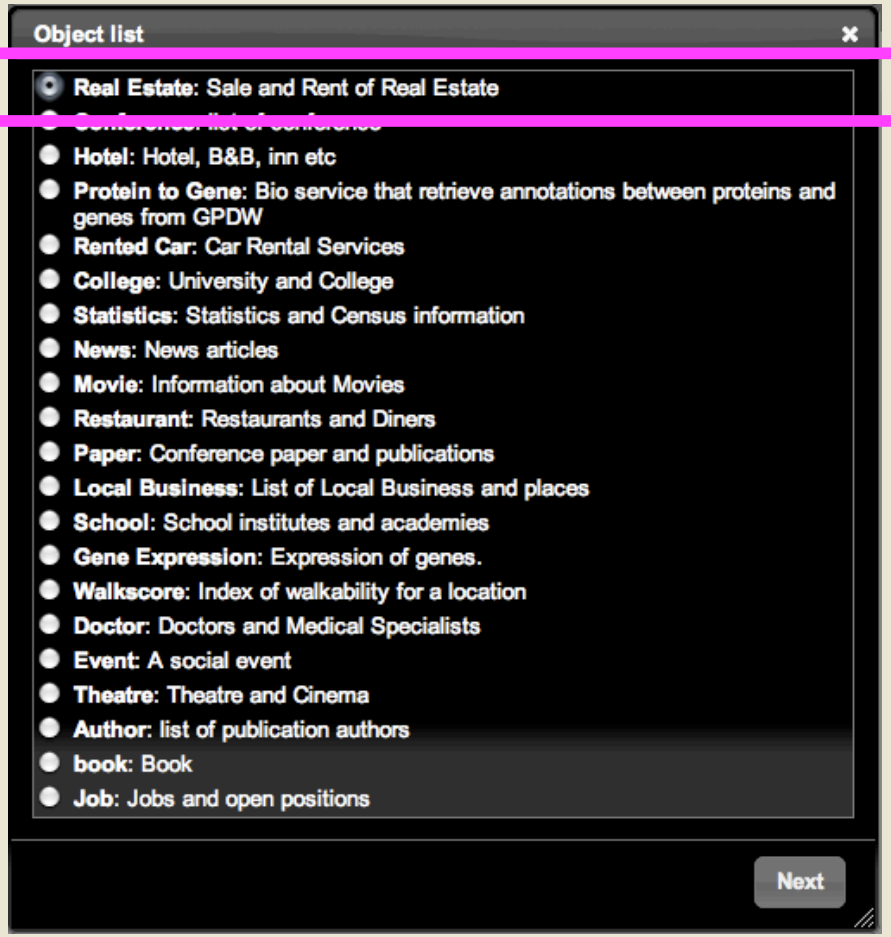
Query formulation

Consists in subsetting and parametrizing the graph...



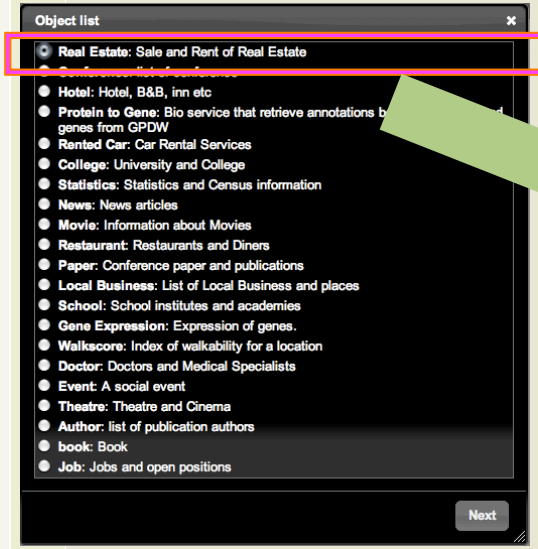
= inputs, outputs + GR = global ranking

Exploration of the Service Space



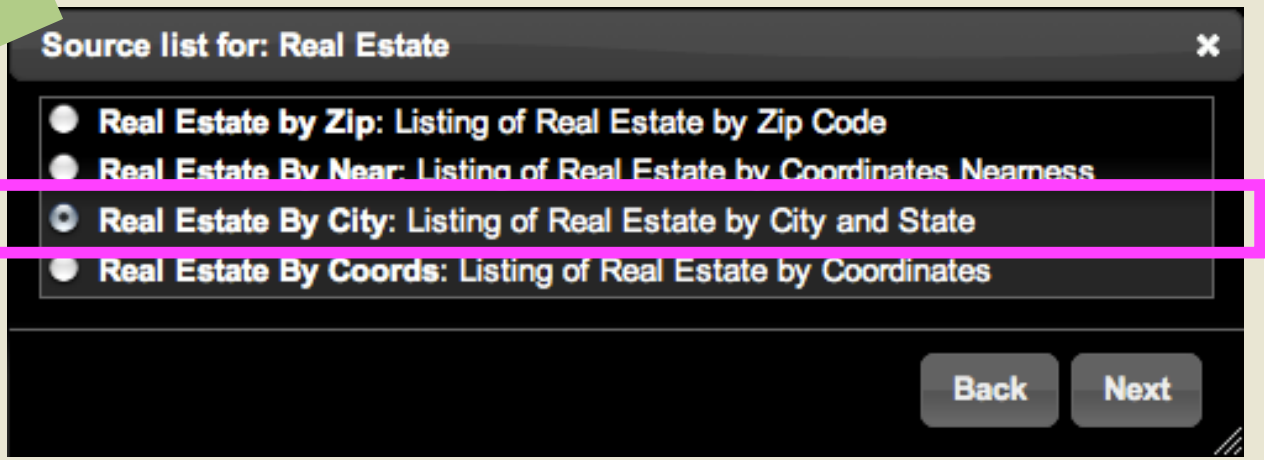
Entity Selection
(service mart)

Exploration of the Service Space

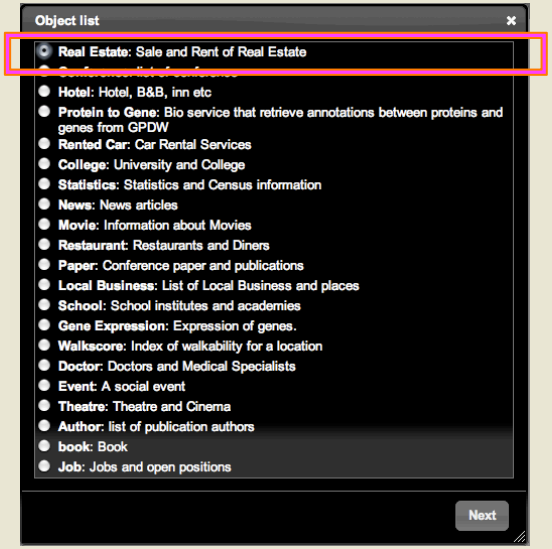


Entity Selection

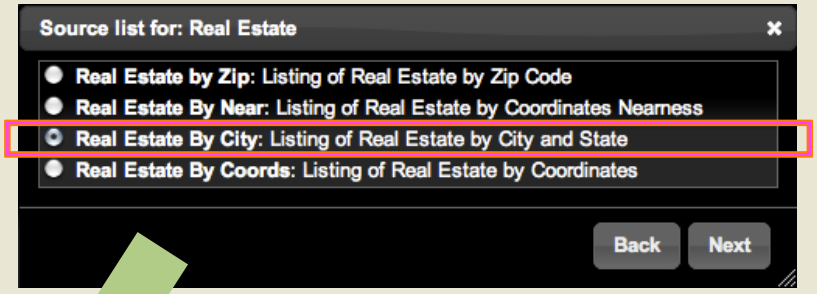
Service Selection
(access pattern)



Exploration of the Service Space



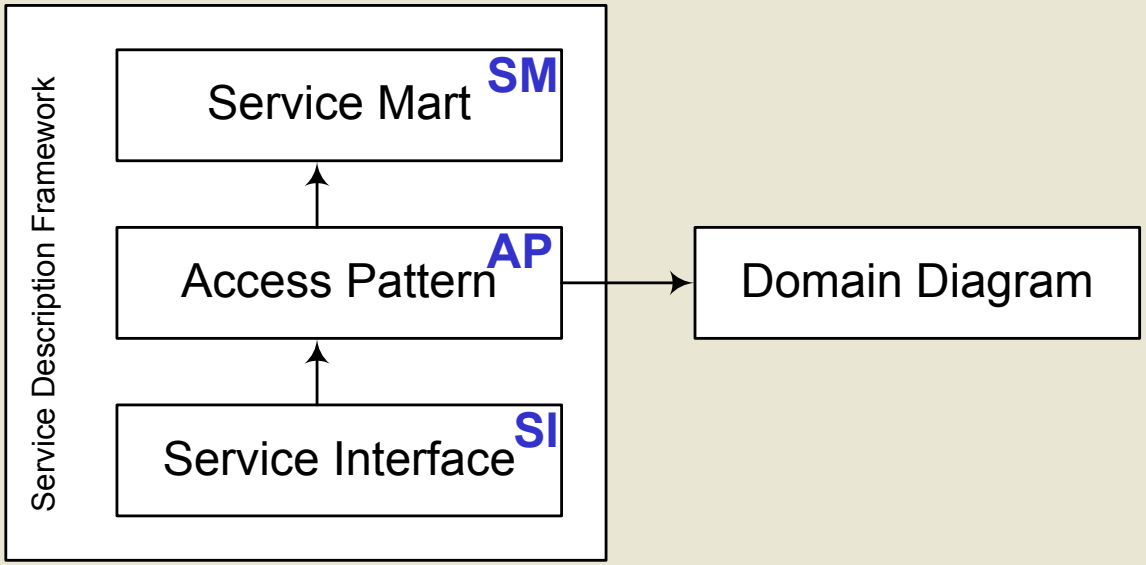
Entity Selection



Service Selection

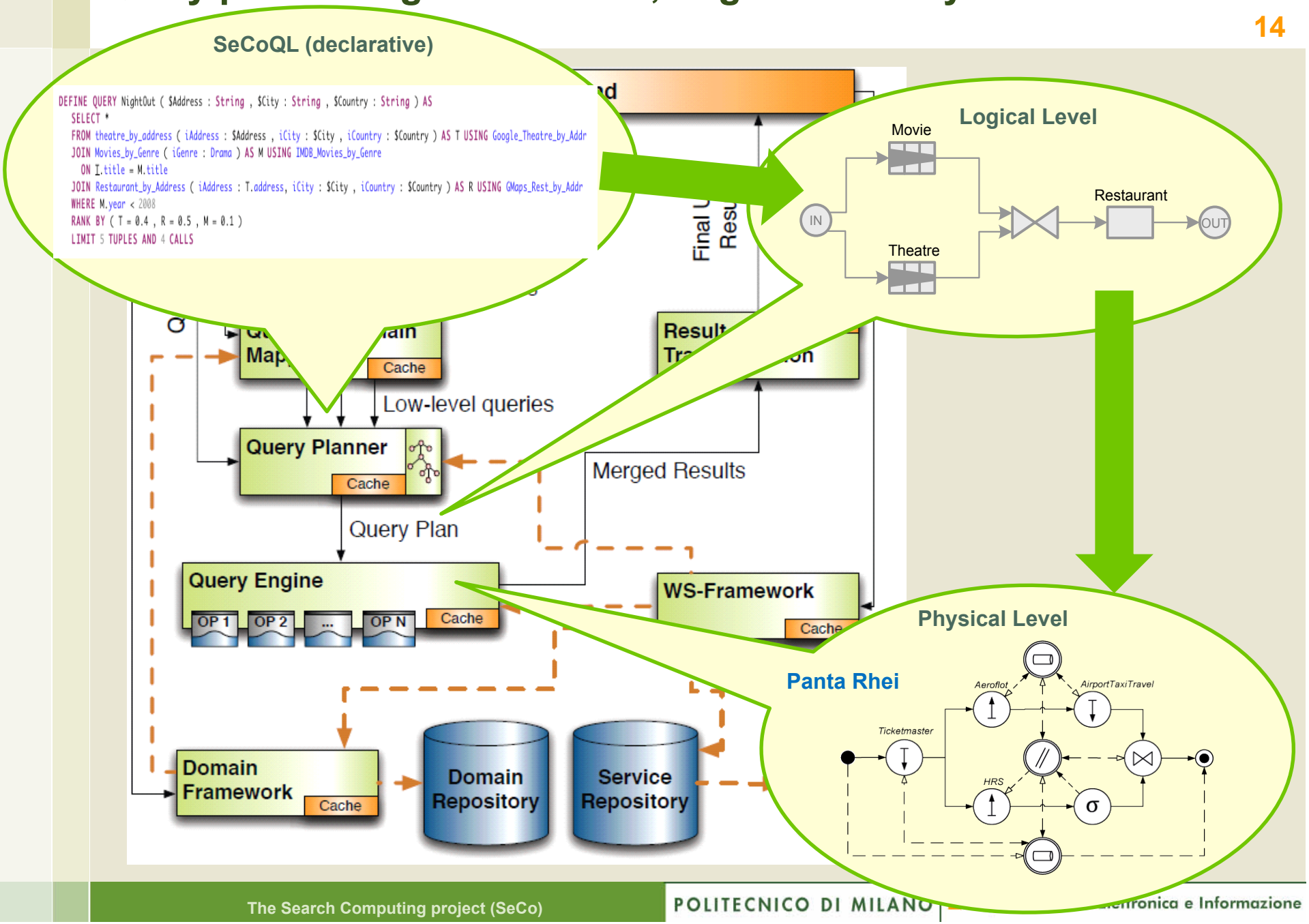
Submit Query

Service Framework in SeCo



Formulation of SeCo Queries
Logical and Physical Query plans
The Panta Rhei execution framework

Query processing: Declarative, Logical and Physical formulation

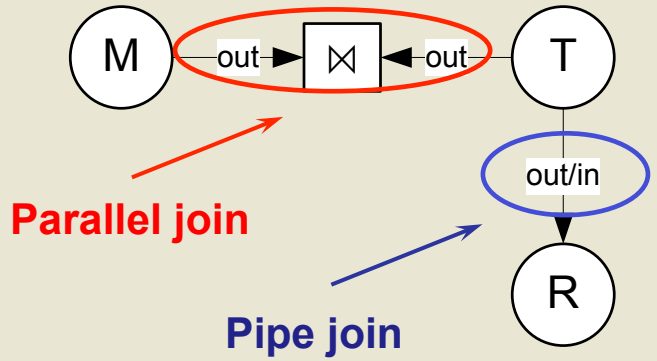


SeCoQL

A good old drama movie showing tonight in a theatre close to a good restaurant

NightOut ("via Golgi", "Milan", "Italy")

```
DEFINE QUERY NightOut ( $Address : String , $City : String , $Country : String ) AS
   SELECT *
   FROM theatre_by_address ( iAddress : $Address , iCity : $City , iCountry : $Country ) AS T USING Google_Theatre_by_Addr
   JOIN Movies_by_Genre ( iGenre : Drama ) AS M USING IMDB_Movies_by_Genre
     ON T.title = M.title
   JOIN Restaurant_by_Address ( iAddress : T.address, iCity : $City , iCountry : $Country ) AS R USING GMaps_Rest_by_Addr
   WHERE M.year < 2008
   RANK BY ( T = 0.4 , R = 0.5 , M = 0.1 )
   LIMIT 5 TUPLES AND 4 CALLS
```



First step: from conjunctive queries to logical plans

A good American action movie showing tonight in a theatre in Milan, close to a good Mexican restaurant

results(Title, TAddress, RName,RPhone, RAddress, ShowTimes.st) :–

Movie(Genre^{I}, MCountry^{I}, Title^{O}, Score^{O}, Genres.Genre^{O}, Actors.Act^{O})

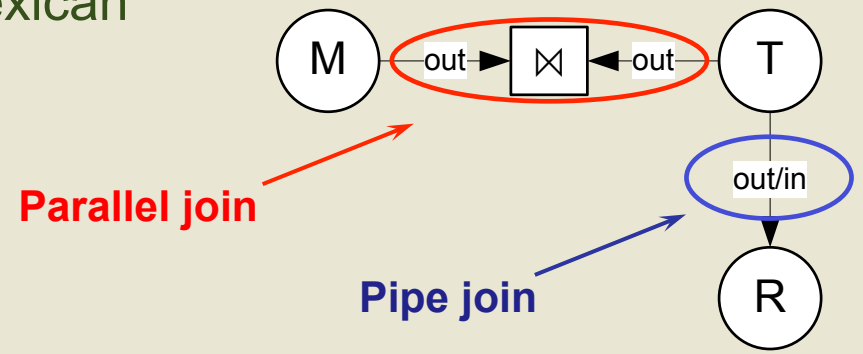
Theatre(Country^{I}, City^{I}, TName^{O}, TAddress^{O}, TPhone^{O}, Titles.Title^{O}, RunningTimes.rt^{O}, ShowTimes.st^{O})

Restaurant(Country^{I}, City^{I}, Category^{I}, TAddress^{I}, RName^{O}, RAddress^{O}, RPhone^{O}, RScore^{O}, Categories.cat^{O})

Genre="Action", MCountry="USA", City="Milan"

Country="Italy", Category="Mexican"

User inputs



Query Processor Overview: Two-Step Approach

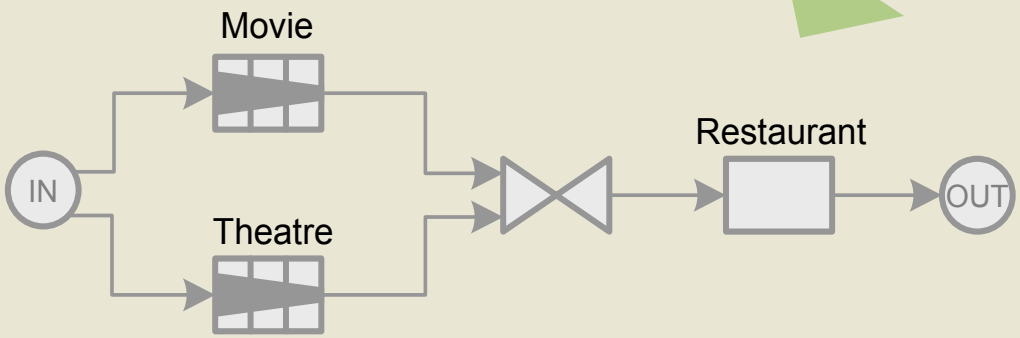
- Logical level
  - **Input**: a **declarative conjunctive** query
    - Access Patterns and Service Interfaces have already been chosen in higher layers, and join types are therefore known
  - **Output**: a **logical plan**, a specification of a workflow with quantitative estimates of the size of partial results
    - The planner exploits the available degrees of freedom to fix the topology, the number and sequence of service invocations, the join strategies, ...
- Physical level
  - From the **query planner** level to **run-time enactment**
  - **Output**: the **physical plan** in the **Panta Rhei** language
    - Directly executable by the engine
    - **Distribution**, **parallelization** and **replication** are considered

First step: from conjunctive queries to logical plans

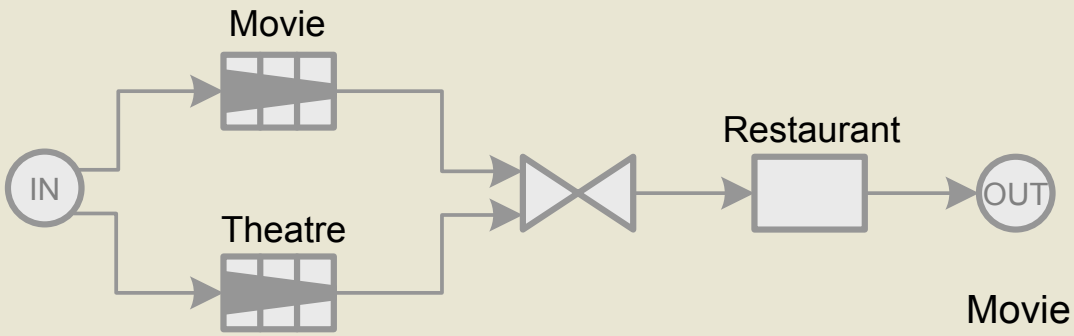

```
DEFINE QUERY NightOut ( $Address : String , $City : String , $Country : String ) AS
   SELECT *
   FROM theatre_by_address ( iAddress : $Address , iCity : $City , iCountry : $Country ) AS T USING Google_Theatre_by_Addr
   JOIN Movies_by_Genre ( iGenre : Drama ) AS M USING IMDB_Movies_by_Genre
     ON T.title = M.title
   JOIN Restaurant_by_Address ( iAddress : T.address, iCity : $City , iCountry : $Country ) AS R USING GMaps_Rest_by_Addr
   WHERE M.year < 2008
   RANK BY ( T = 0.4 , R = 0.5 , M = 0.1 )
   LIMIT 5 TUPLES AND 4 CALLS
```

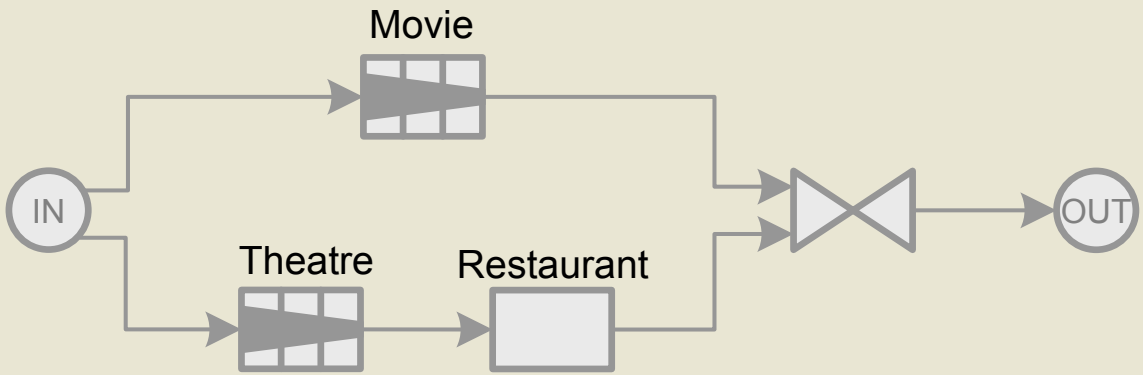
Generation of a logical plan

Estimated number of invocations and cardinality of inputs and outputs



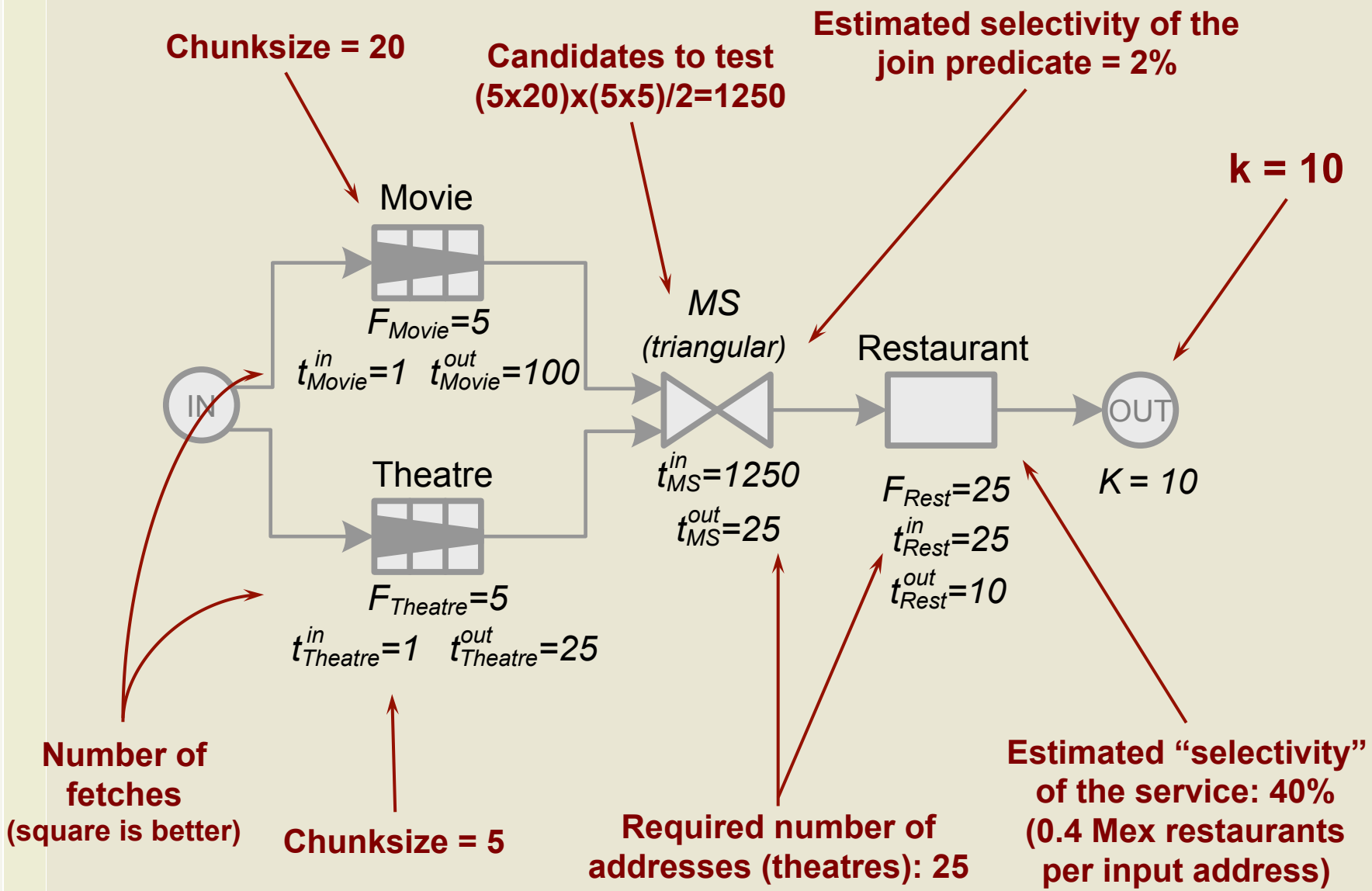
Choice of topology for the logical plan





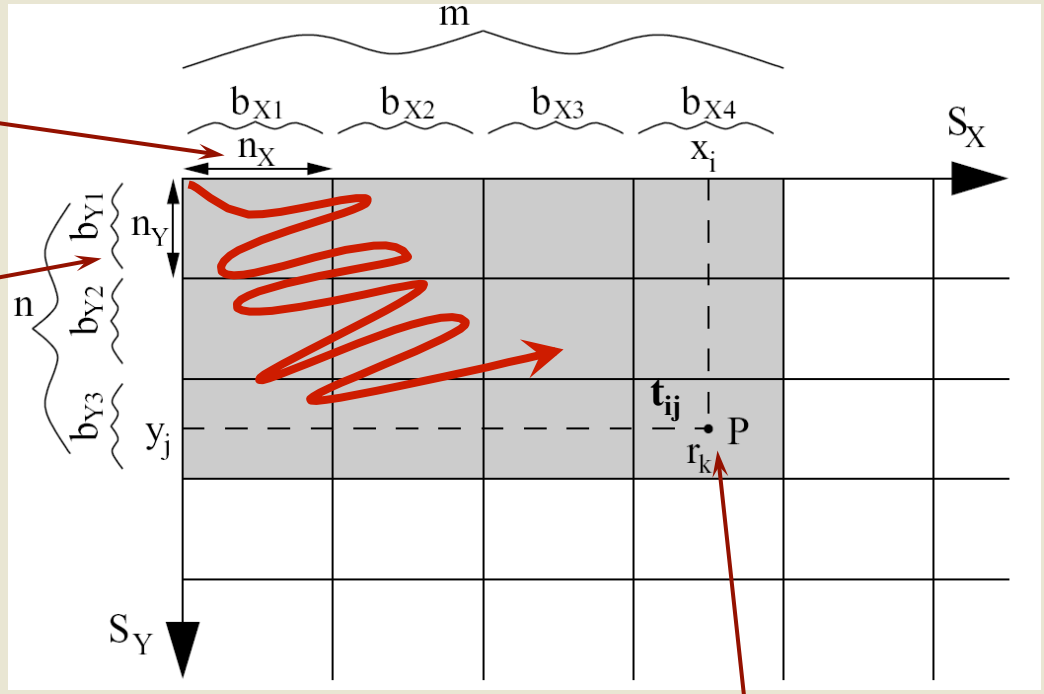
- Heuristics:
  - **Selective first**
    - Anticipating selective invocations minimizes intermediate results
    - Expensive services can be invoked later, with qualified input only
  - **Parallel is better**
    - Invocations with no i/o dependencies should always be performed in parallel, to minimize latency

Estimating the number of fetches



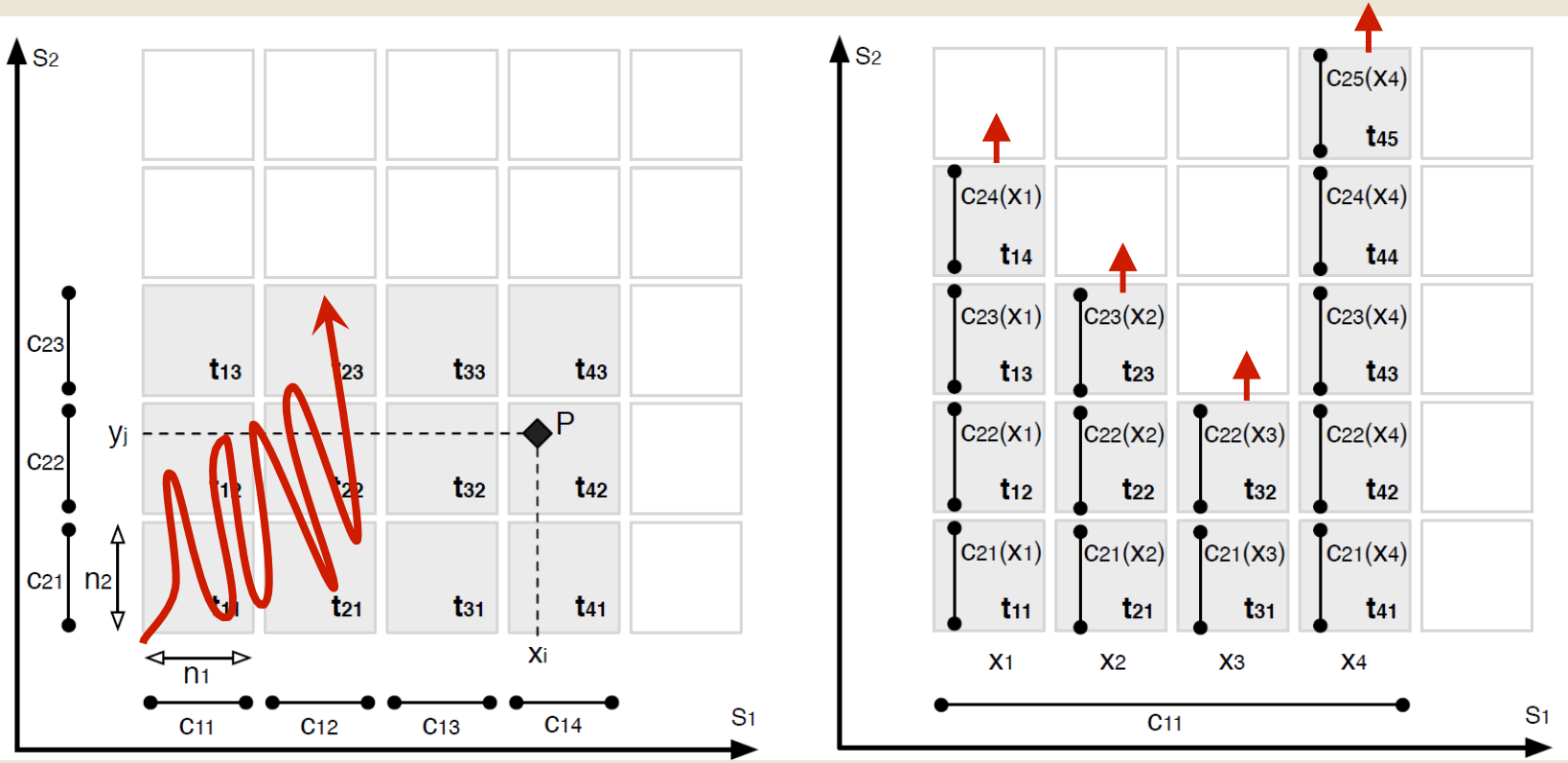
Choice of the join strategies (search space)

- The **join search space**
  - Different explorations for different joins methods under different assumptions and with different guarantees



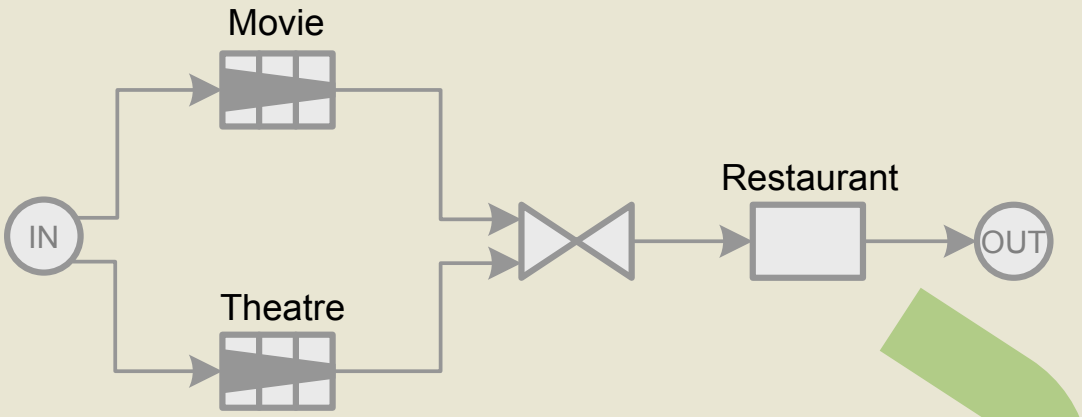
Any exploration trajectory for this space is a parallel join strategy

Search space: parallel join vs pipe join

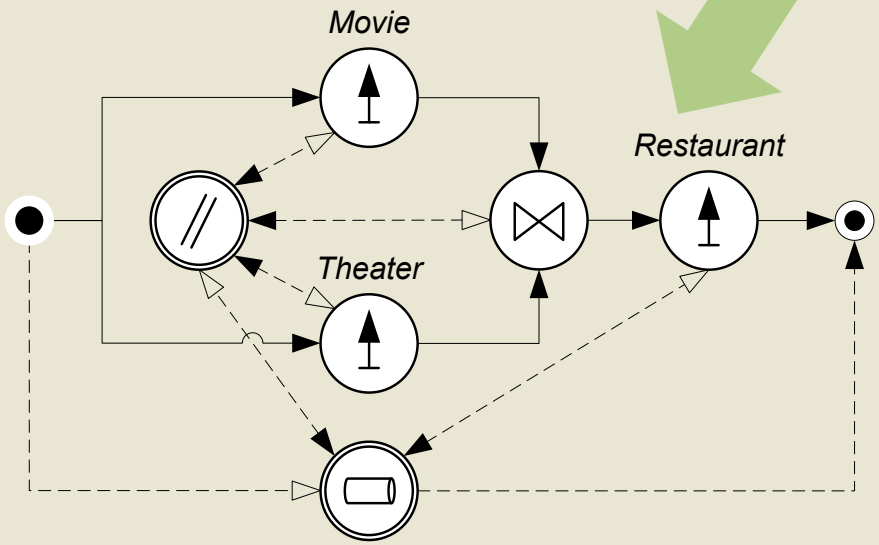


In the case of pipe joins, a join strategy is a pulling strategy deciding at each step from which invocation the next chunk is to be fetched

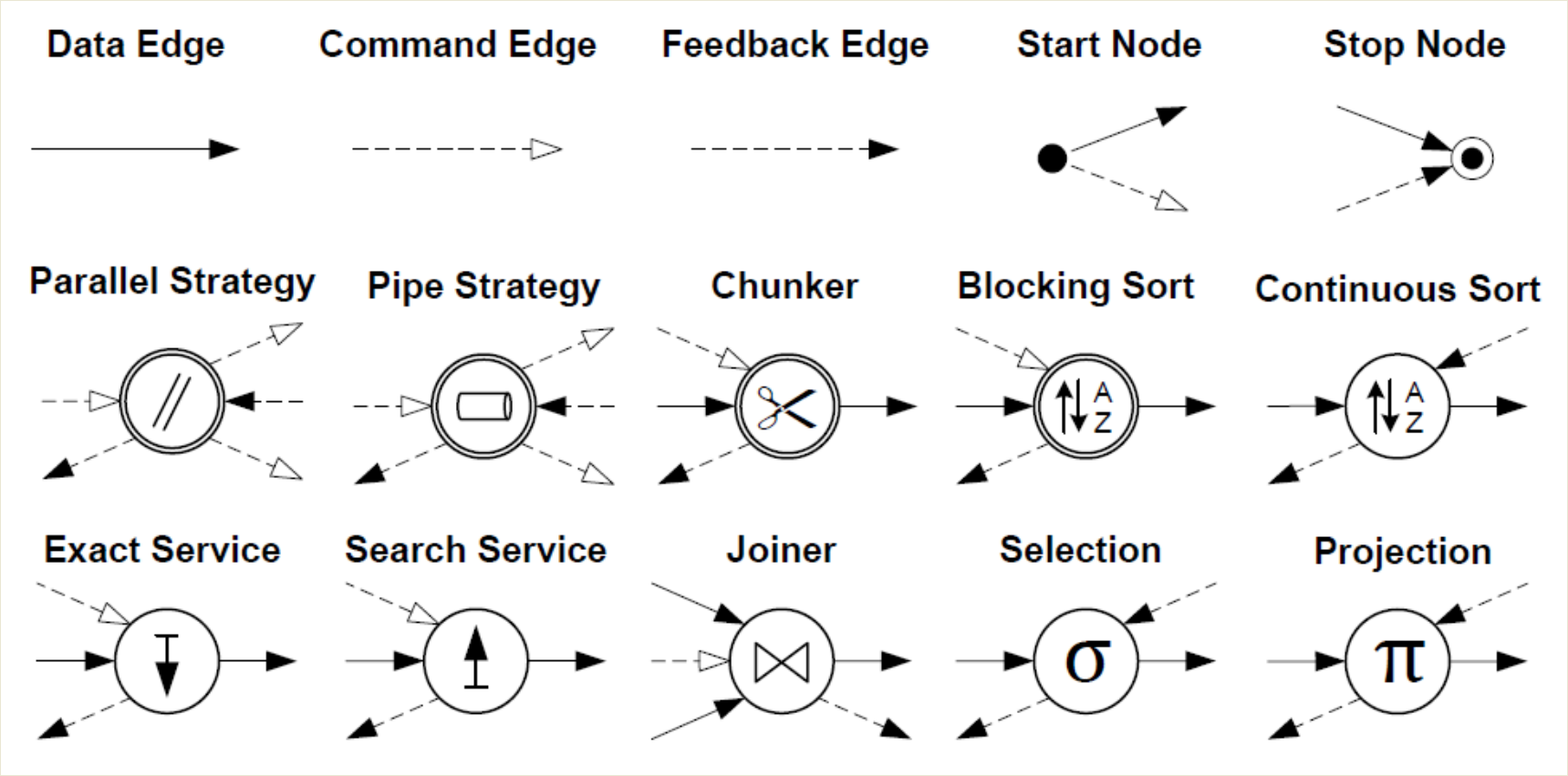
Second step: from logical to physical query plans



Then the planner generates a physical, executable query plan, expressed in *Panta Rhei*



Units


Data Edge
Command Edge
Feedback Edge
Start Node
Stop Node
Parallel Strategy
Pipe Strategy
Chunker
Blocking Sort
Continuous Sort
A
Z
Exact Service
Search Service
Joiner
Selection
Projection
σ
π

Query Execution Plans

- A query execution plan (QEP) accepts an incoming **data and control flow** edge and has a data and control flow edge in output
  - The incoming data edge buffers chunks of tuples in input, while the outgoing data edge transmits chunks of tuples in output to a downstream query execution plan or to the stop node.
  - The commands that are received through the incoming control edge express how the tuples in input relate to chunks in output. The outgoing control edge transmits both new commands generated by the query execution plan and feedback data about the execution of the plan.

Composition Rules (QEPs as basic building blocks)

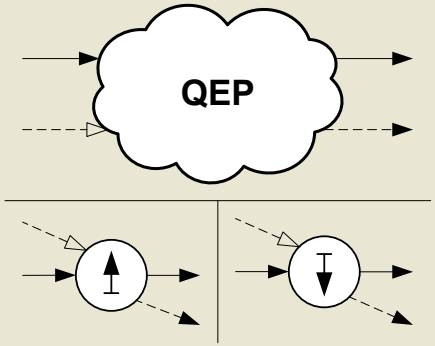
(1) $QEP := \{$  $\}$

(2) $QEP := QEP \cup \{$  $\}$

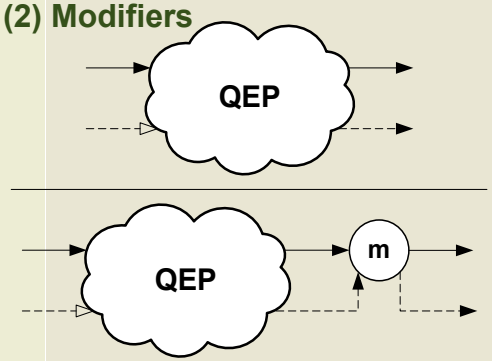
(3) $QEP := QEP \bowtie_{pipe} QEP$

(4) $QEP := QEP \bowtie_{parallel} QEP$

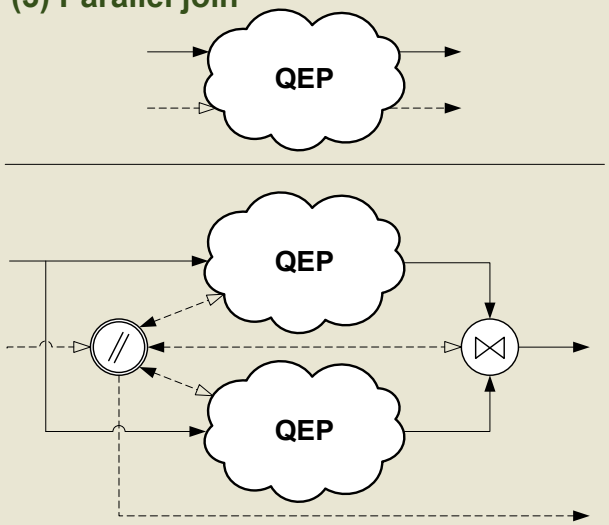
(1) Service invocations



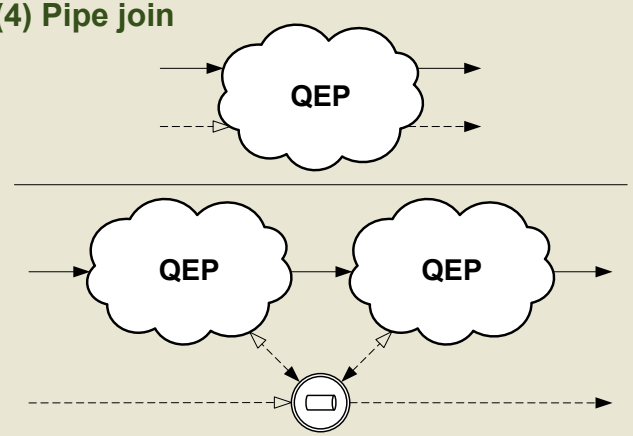
(2) Modifiers



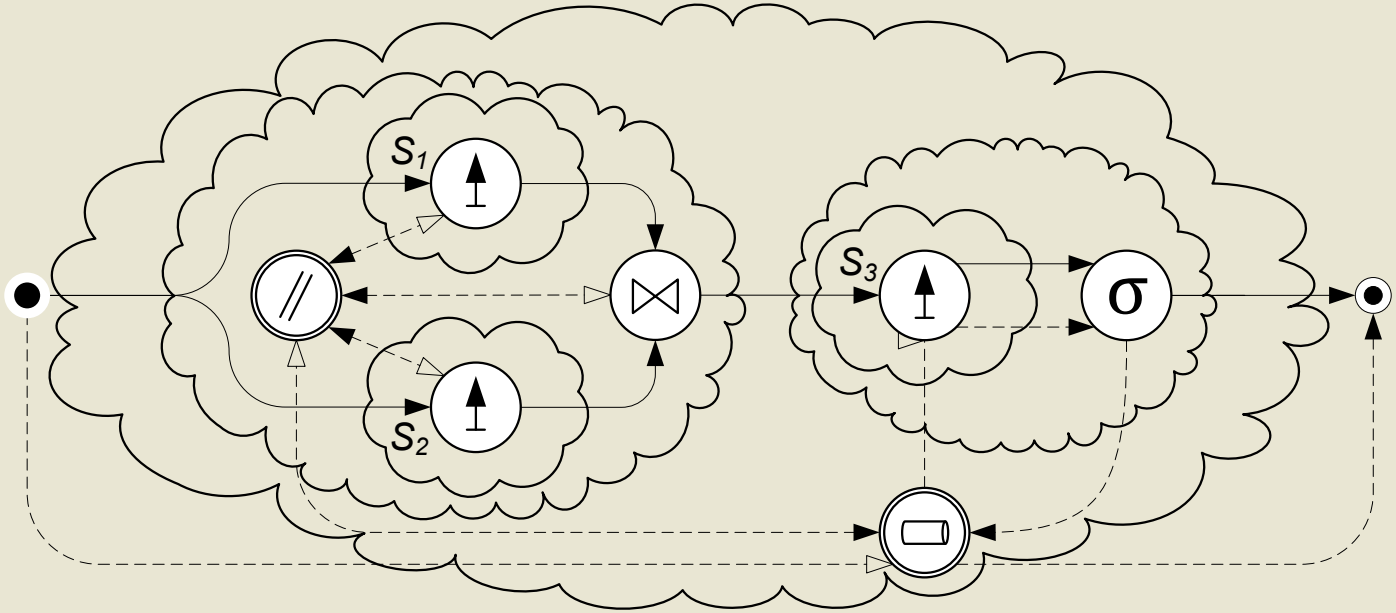
(3) Parallel join



(4) Pipe join



Query Plan Compilation: Topology



- Traversal of logical plan to identify query execution plans
  - logical plan is processed “backwards”
  - returns an ordered list of query execution plans
- Processing of query execution plans
  - insertion of unit configurations and parameters
  - consecutive query execution plans are combined with pipe joins
- Right associativity
  - $QEP \bowtie_{pipe} QEP \bowtie_{pipe} QEP = QEP \bowtie_{pipe} (QEP \bowtie_{pipe} QEP)$

Support for Forward and Backward Control

- Strategy depends on whether execution plan is scheduled in forward or backward manner
  - **Forward**: the strategy is entirely determined by scheduler at compile-time and configured in strategy units. The strategy defines which input tuple should be used in the next execution of a query execution plan.
  - **Backward**: there is no predefined strategy, but an algorithm that is encapsulated in the strategy units controls the sending of commands to the dependent query execution plan
  - **"Middle Way" (adaptive)**: the strategy unit has a predefined strategy but takes feedback into consideration to allow a bounded deviation from the plan or react to user input.

Panta Rhei: Main Stateless Units

- Operation is entirely defined by the inputs and can be distributed, parallelized and replicated

Search Service Invocation: Completes the tuples in input by invoking a search service. Per service invocation, it returns a chunk of completed tuples in output.

Joiner: Receives chunks of tuples on two different data edges and joins them into chunks of tuples according to a predicate.

Selection: Filters chunks of tuples according to a selection predicate. Since the selection unit does not re-chunk the tuples, the chunk size can decrease in the order of the selectivity of the given predicate.

Panta Rhei Main Stateful Units: Controllers

- Operation depends on the inputs and the internal state of the unit.

Parallel Strategy: Controls two query execution plans that are composed in a parallel join. It breaks down the commands it receives in input and distributes them to the two sub-plans and, optionally receives feedback from the query execution plans.

Pipe Strategy: Controls two query execution plans that are composed in a pipe join. It breaks down the commands it receives in input and distributes them to the two sub-plans and, optionally receives feedback from the query execution plans.

Good vs Top-k joins
Pluggable join strategies
Demo
System architecture

Guarantee of top-k-ness

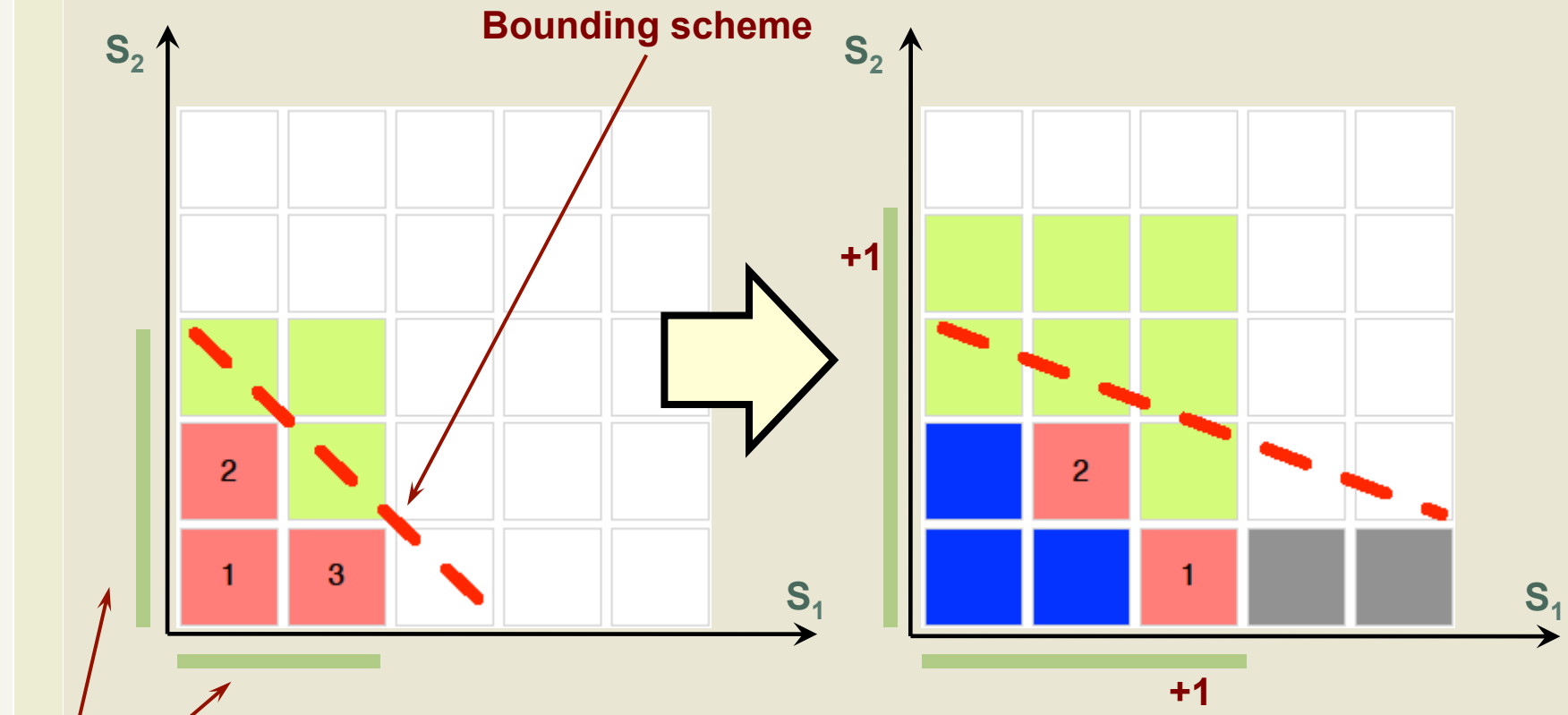
- Good-k strategies
  - Rely on the sorting of data sources and produce query output as soon as it is available

- Top-k strategies
  - The output is ordered according to a score function
  - Results are sent to the user when there is a guarantee that no result with higher score can be produced
  - Rely on the sorting of data sources
  - Slower strategies
  - Higher time-to-screen

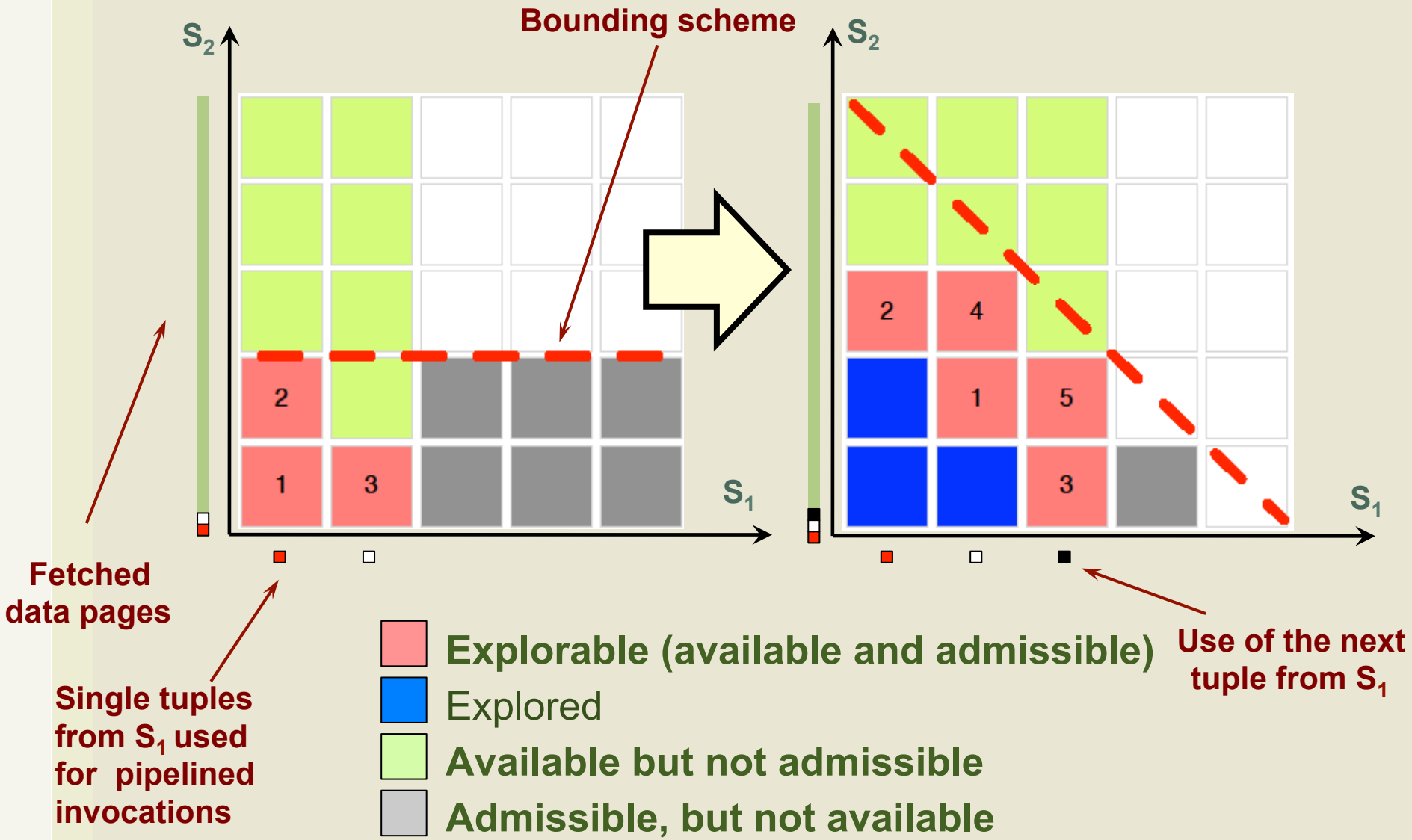
Offline good strategies

- Sequences of steps
- Pulling strategy
  - The number of invocations to be sent to each service
- Bounding scheme
  - Determines the tiles which will be joined
- No adaptation
  - Pulling strategy and bounding scheme are defined at compile-time

Parallel join: a strategy with a bounding scheme



Pipe join: a strategy with a bounding scheme



Offline good parallel join strategy

ALGORITHM 1: Offline good parallel join strategy

Input: $\mathcal{S} = (S_1, S_2, \ldots, S_n)$, set of n data sources to be joined
Input: $\mathcal{J} = (J_1, J_2, \ldots, J_m)$, strategy; $J_i = \{P_i, b_i\}$
Input: $P_i = \{P_{1,i}, P_{2,i}, \ldots, P_{n,i}\}$ pulling strategy at step i
Input: b_i bounding scheme at step i
Data: $\mathcal{A}$, set of admissible tiles of the join space
Data: $\mathcal{R}$, set of explored tiles of the join space
Data: $\mathcal{E}$, list of explorable tiles of the join space

```
1  R ← ∅;
2  for j = 1 to |J| do
3  |   E ← ∅;
   |   // Pulling strategy
4  |   for i = 1 to |S| do
5  |   |   for k = 1 to P_{i,j} do
6  |   |   |   sendFetch(S_i);
   |   // Bounding scheme
7  |   A ← admissible(b_j);
8  |   foreach X ∈ A do
9  |   |   if isAvailable(X) ∧ X ∉ R then
10 |   |   |   E ← E ∪ X;
11 |   E ← sortByDistance(E, b_j);
12 |   foreach X ∈ E do
13 |   |   sendJoinCommand(X);
14 |   |   R ← R ∪ X;
```

Online top strategies

- Invocations are driven by scores
- The next service to be invoked is the most promising one
  - i.e., the one that could lead to the unseen combination with the highest score
- Rely on the feedback received by services and joiner to decide the next invocation to be performed

HRJN

ALGORITHM 2: HRJN Parallel Strategy

Input: $\mathcal{S} = \{S_1, \ldots, S_n\}$, set of n data sources to be joined
Input: $C_{total}(s)$, total number of fetches assigned by the exploration strategy
Data: $F_{aggr}(\mathcal{S}, r)$, function modeling a join combination $r \in \mathbb{N}_0^n$ in terms of its score (weighted by each source $s \in \mathcal{S}$ and normalized to $[1 \ldots 0]$)
Data: $C_{sent}(s)$, count of sent fetches for data source $s \in \mathcal{S}$
Data: $t(s)$, threshold for data source $s \in \mathcal{S}$, computed as $F_{aggr}(\mathcal{S}, r)$
Data: $eof(s)$, end of file flag for data source $s \in \mathcal{S}$
Data: $R = \emptyset$, set of join space points $(x_1, \ldots, x_n)$ for which a join command has been sent
Data: $Q = \emptyset$, set of join space points $(x_1, \ldots, x_n)$ for which a join combination has not been emitted
Data: $t = 1.0$, current score threshold (normalized to $[1 \ldots 0]$)

```
1  begin
2      foreach s ∈ S do
3          C_sent(s) ← 0;
4          t(s) ← 1.0;
5          eof(s) ← false;
       // invoke every service at least once
6      foreach s ∈ S do
7          if C_total > 0 then
8              eof(s) ← SendFetch(s);
9              UpdateSource(s, ReceiveFeedback(s));
10             C_total ← C_total − 1;
11             C_sent(s) ← C_sent(s) + 1;
```

HRJN


```
12  while C_total > 0 do
        // choose next service to invoke and update its metadata
13      hScore ← 0;
14      foreach s ∈ S do
15          if ¬eof(s) and t(s) > hScore then
16              hScore ← t(s);
17              z ← s;
18      eof(z) ← SendFetch(z);
19      UpdateSource(z, ReceiveFeedback(z));
20      C_total ← C_total − 1;
21      C_sent(z) ← C_sent(z) + 1;
22      t ← UpdateThreshold();
    // create join combinations
23  foreach r ← (r_1, ..., r_n) ∈ N_0^n and r_i ∈ {1, ..., C_sent(s_i)} do
24      if r ∉ R then
25          SendJoinCommand(r);
26          R ← R ∪ r;
27          Q ← Q ∪ r;
    // emit top-k tuples
28  foreach r ∈ Q do
29      if F_aggr(S, r) >= t then
30          Q ← Q \ r;
31          Emit(r);
```

Adaptive online strategies

- The feedback allows handling unexpected situations
  - Unexpected score values
  - Unexpected predicate selectivity and result cardinality
  - Unexpected service response time
  - Data source failures
  - Result diversification

Diversification parallel join strategy

ALGORITHM 3: Diversification parallel join strategy

Input: $\mathcal{S} = (S_1, S_2, \ldots, S_n)$, set of n data sources to be joined
Input: $\mathcal{J} = (J_1, J_2, \ldots, J_m)$, strategy; $J_i = \{P_i, b_i\}$
Input: $P_i = \{P_{1,i}, P_{2,i}, \ldots, P_{n,i}\}$ pulling strategy at step i
Input: b_i bounding scheme at step i
Data: $\mathcal{A}$, set of admissible tiles of the join space
Data: $\mathcal{R}$, set of explored tiles of the join space
Data: $\mathcal{E}$, list of explorable tiles of the join space

```
1  R ← ∅;
2  for j = 1 to |J| do
3  |   E ← ∅;
   |   // Pulling strategy
4  |   for i = 1 to |S| do
5  |   |   for k = 1 to P_{i,j} do
6  |   |   |   sendFetch(S_i);
   |   // Bounding scheme
7  |   A ← admissible(b_j);
8  |   foreach X ∈ A do
9  |   |   if isAvailable(X) ∧ X ∉ R then
10 |   |   |   E ← E ∪ X;
11 |   for i = 1 to |E| do
12 |   |   E ← sortByDiversification(E, R);
13 |   |   X ← peek(E);
14 |   |   sendJoinCommand(X);
15 |   |   R ← R ∪ X;
16 |   |   E ← E \ X;
```

System architecture