Distributed Query Optimization

Distribution as a Means to Achieve Parallelism

Motivation

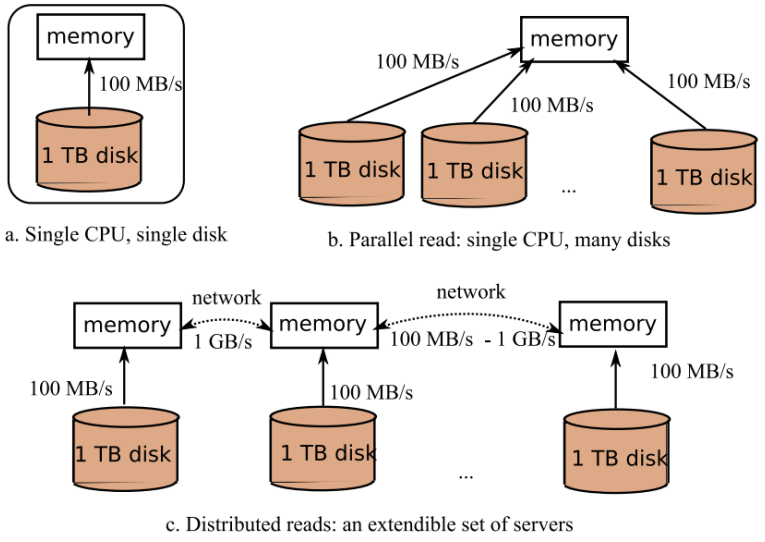
- In NOSQL efficiency is achieved by means of parallelism
  - Query processing must be able to exploit parallelism
    - Divide-and-conquer philosophy

- Reminder:

Type	Latency	Bandwidth
Disk	$\approx 5 \times 10^{-3}$s (5 millisec.);	At best 100 MB/s
LAN	$\approx 1 - 2 \times 10^{-3}$s (1-2 millisec.);	$\approx$ 1GB/s (single rack); $\approx$ 100MB/s (switched);
Internet	Highly variable. Typ. 10-100 ms.;	Highly variable. Typ. a few MB/s.;

Bottom line (1): it is approx. one order of magnitude faster to exchange main memory data between 2 machines in a data center, that to read on the disk.

Bottom line (2): exchanging through the Internet is slow and unreliable with respect to LANs.



Distributed Query Processing

Overall Architecture

Reminder: Challenges in Distributed Databases

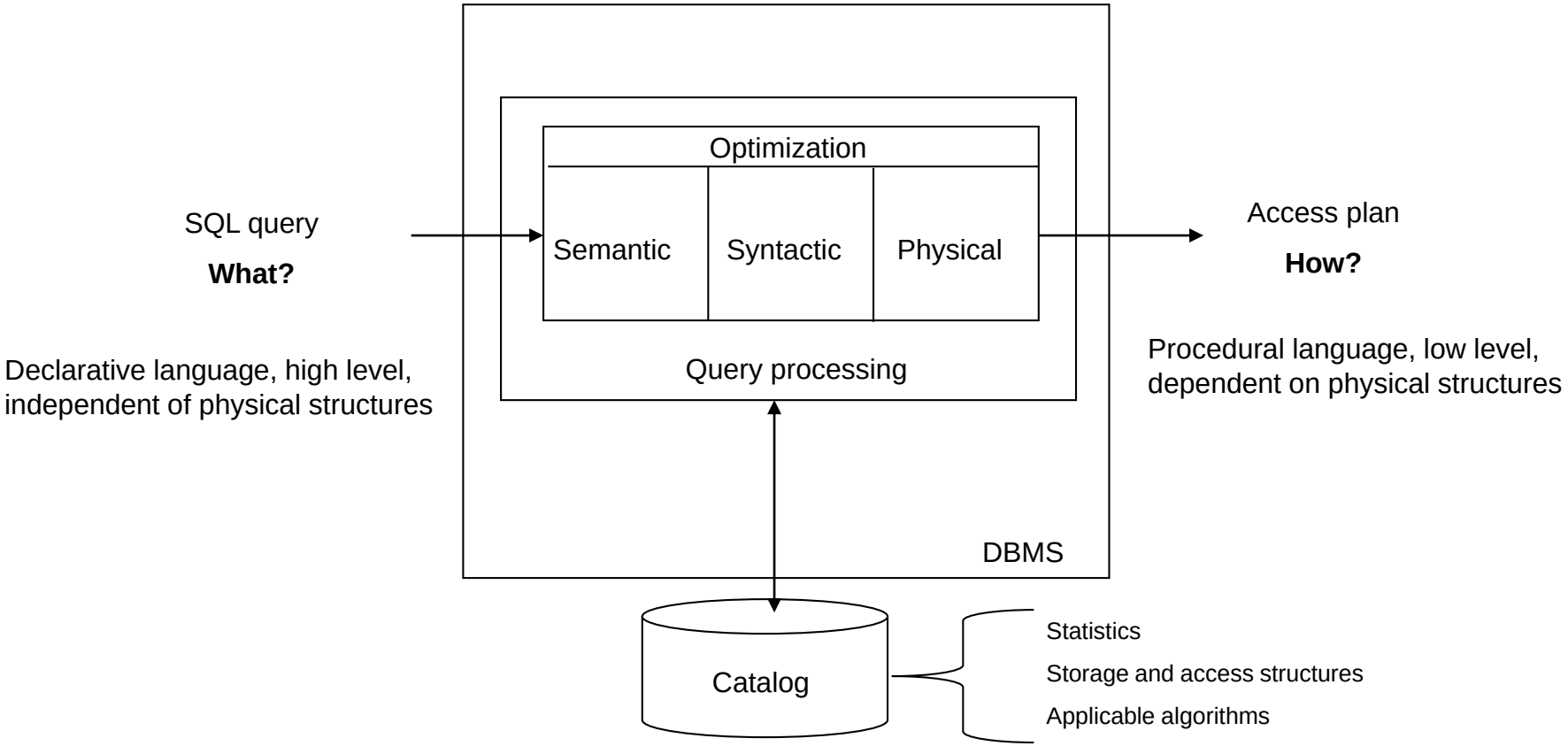
I. Distributed DB design
- Node distribution
- Data fragments
- Data allocation (replication)

II. Distributed DB catalog
- Fragmentation trade-off: Where to place the DB catalog
  - Global or local for each node
  - Centralized in a single node or distributed
  - Single-copy vs. Multi-copy

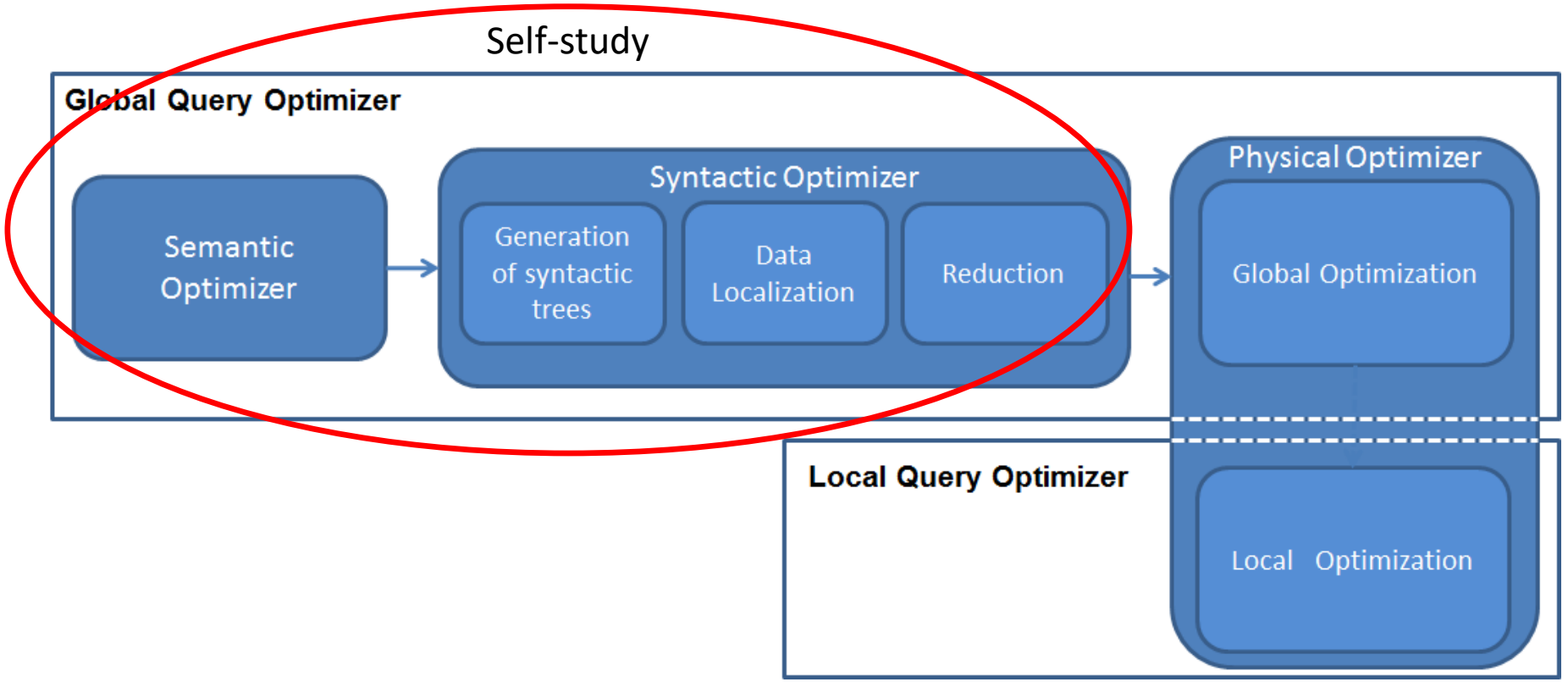
III. Distributed query processing
- Data distribution / replication
- Communication overhead

IV. Distributed transaction management
- How to enforce the ACID properties
  - Replication trade-off: Queries vs. Data consistency between replicas (updates)
  - Distributed recovery system
  - Distributed concurrency control system

Architecture of the Query Optimizer



Phases of Distributed Query Processing



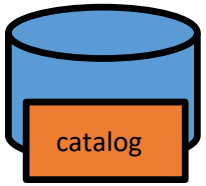
Challenge III: Distributed Query Processing

T

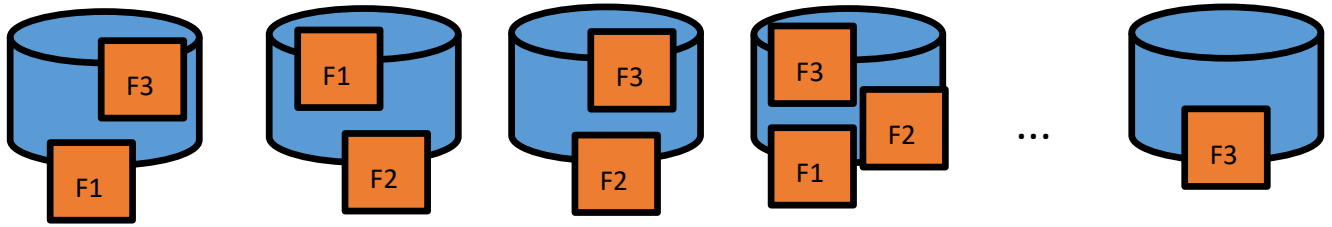
A	B	C	D	E

Conceptual View

Primary server



Catalog:
T <<frag. strategy>>
F1: @S1, @S2, @S4
F2: @S2, @S3, @S4
F3: @S1, @S3, @S4, @Sn

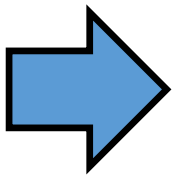


Secondary servers

Physical View

Challenge III: Distributed Query Processing

SELECT * FROM T
WHERE A > 5

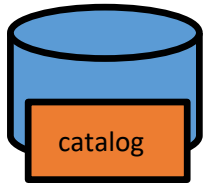


T

A	B	C	D	E

Conceptual View

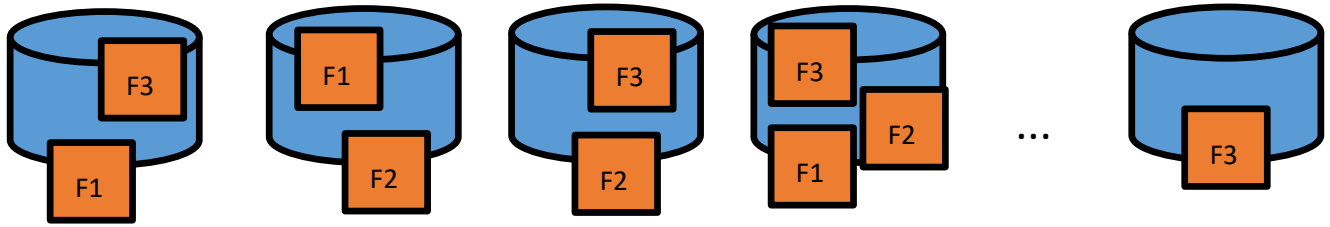
Primary server



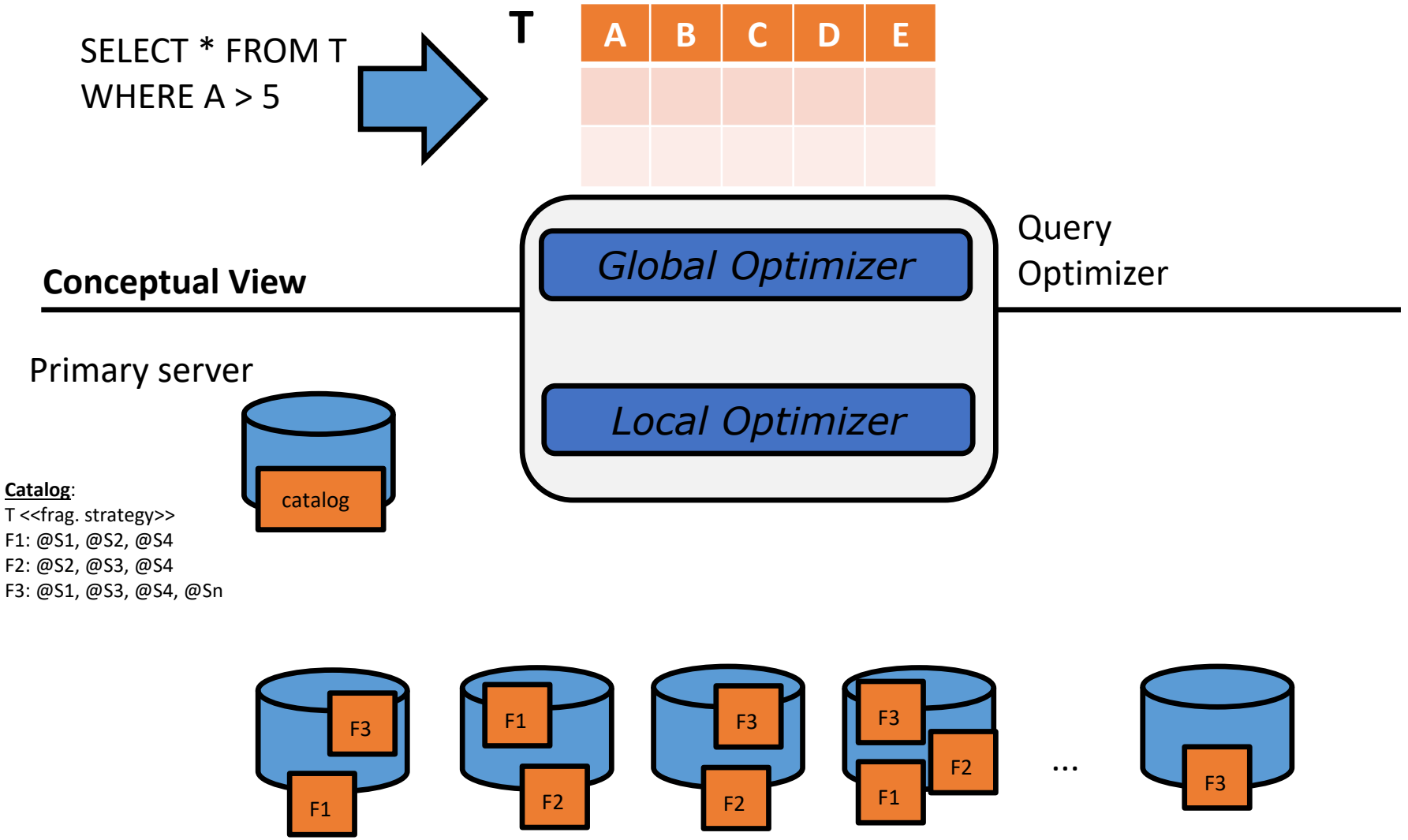
Catalog:
T <<frag. strategy>>
F1: @S1, @S2, @S4
F2: @S2, @S3, @S4
F3: @S1, @S3, @S4, @Sn

Physical View

Secondary servers

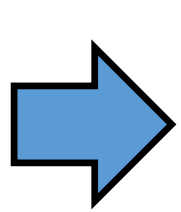


Challenge III: Distributed Query Processing



Challenge III: Distributed Query Processing

SELECT * FROM T
WHERE A > 5



T

A	B	C	D	E

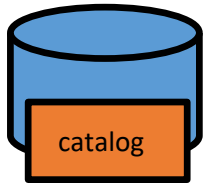
Conceptual View

Query Optimizer

Global Optimizer

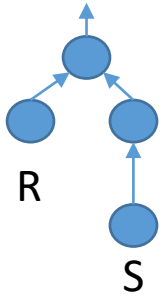
Local Optimizer

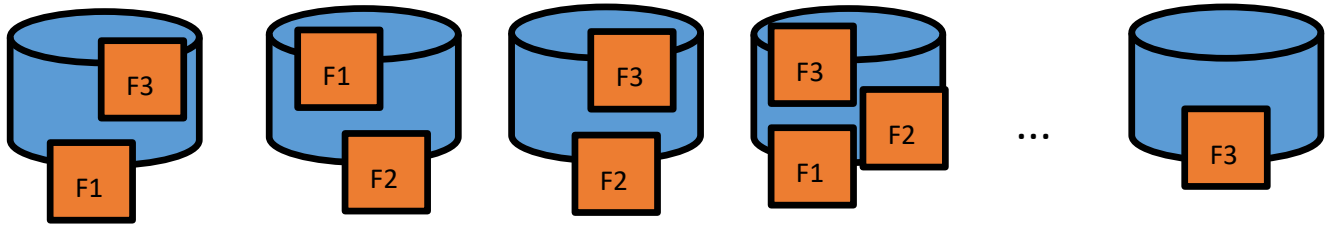
Primary server



Catalog:
T <<frag. strategy>>
F1: @S1, @S2, @S4
F2: @S2, @S3, @S4
F3: @S1, @S3, @S4, @Sn

1) Generate the **syntactic representation** of the query

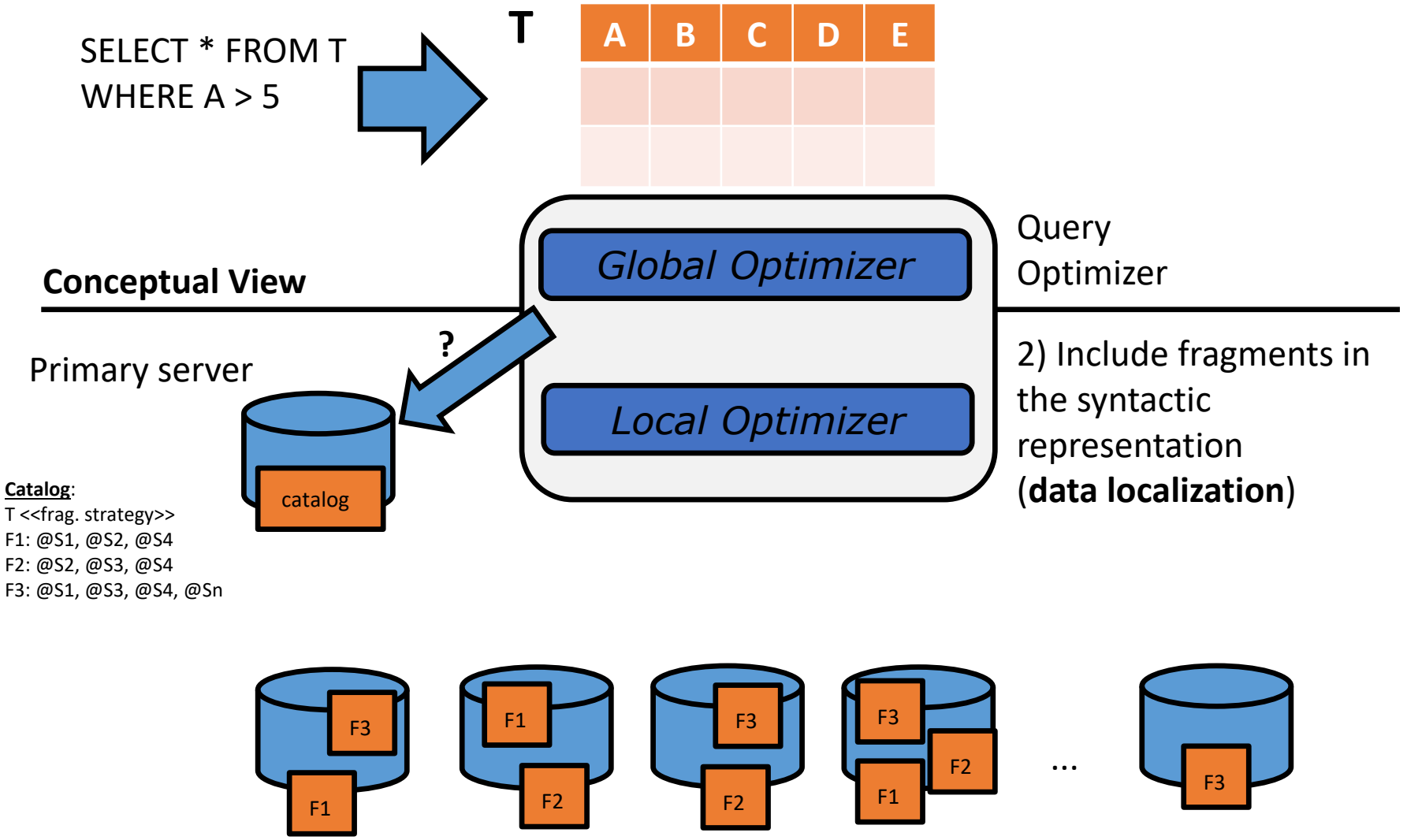




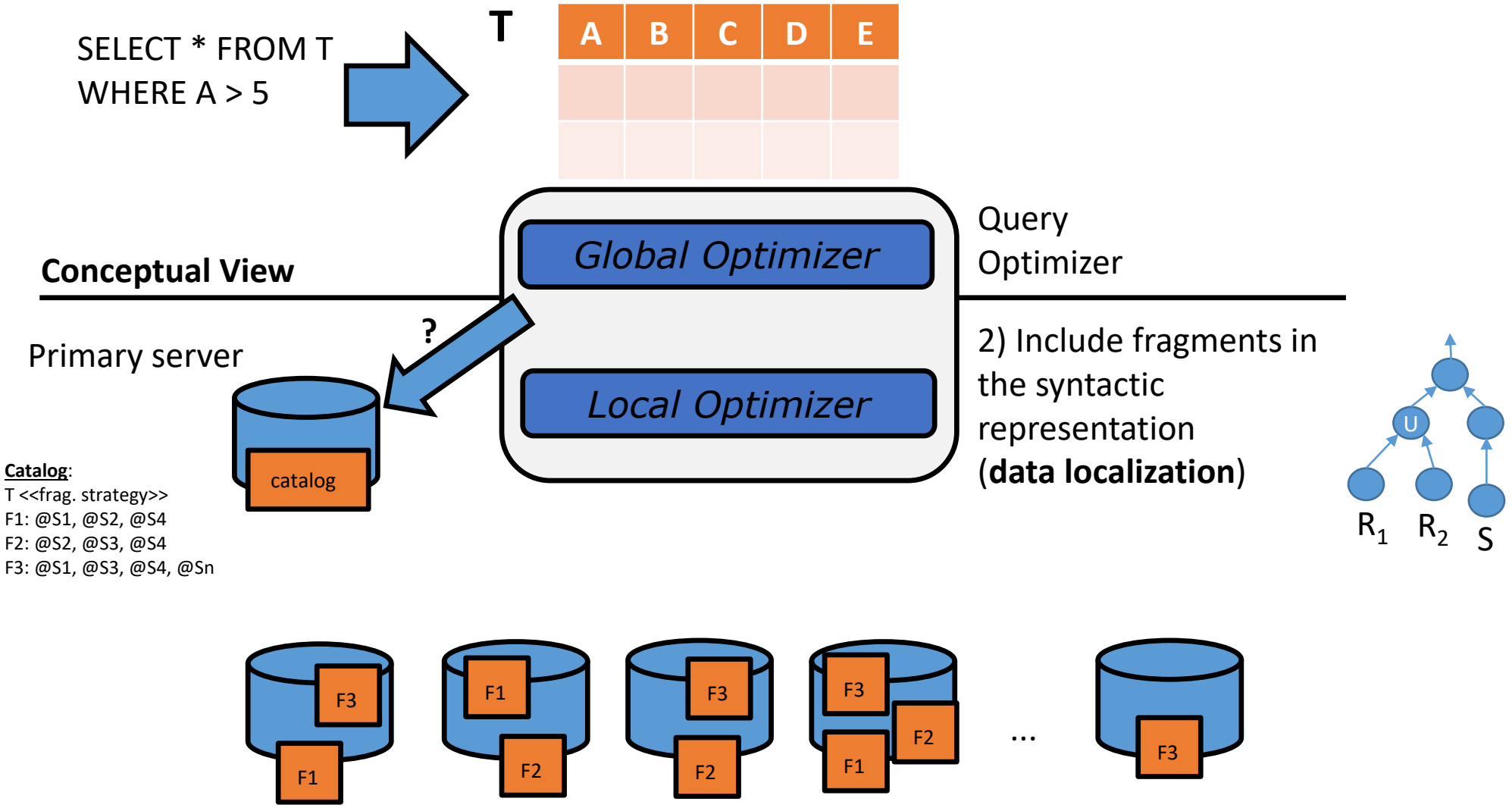
Physical View

Secondary servers

Challenge III: Distributed Query Processing



Challenge III: Distributed Query Processing



Challenge III: Distributed Query Processing

SELECT * FROM T
WHERE A > 5

T

A	B	C	D	E

Conceptual View

Primary server

?

catalog

Catalog:
T <<frag. strategy>>
F1: @S1, @S2, @S4
F2: @S2, @S3, @S4
F3: @S1, @S3, @S4, @Sn

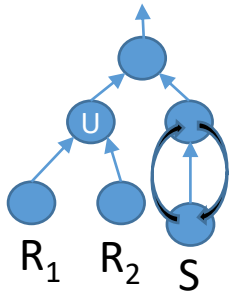
Global Optimizer

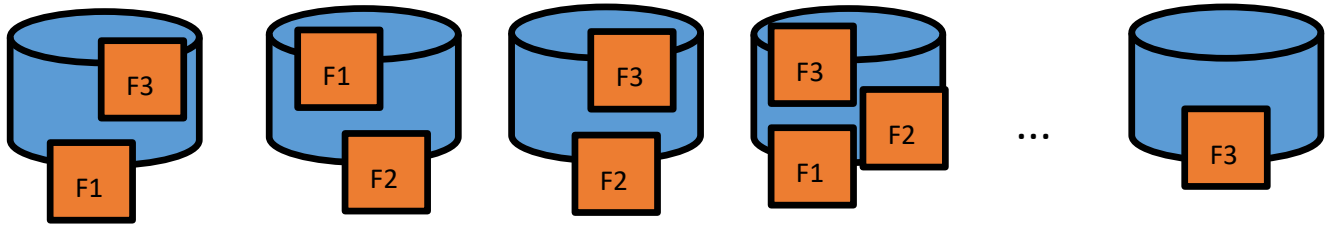
Local Optimizer

Query Optimizer

2) Include fragments in the syntactic representation (**data localization**)

2.1) Optimize the resulting tree



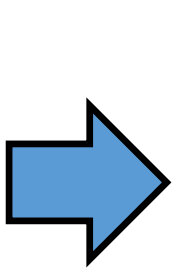


Physical View

Secondary servers

Challenge III: Distributed Query Processing

SELECT * FROM T
WHERE A > 5



T

A	B	C	D	E

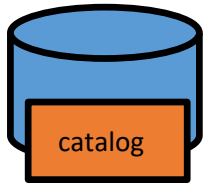
Conceptual View

Global Optimizer

Local Optimizer

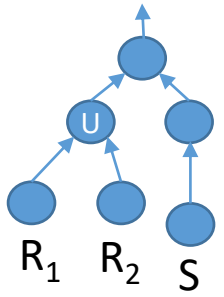
Query Optimizer

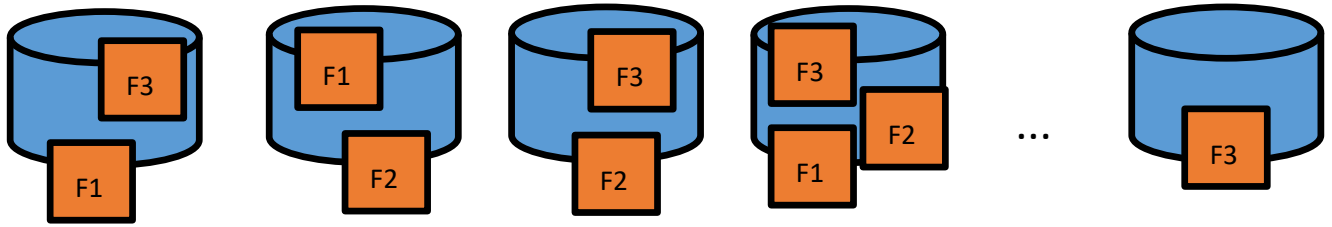
Primary server



Catalog:
T <<frag. strategy>>
F1: @S1, @S2, @S4
F2: @S2, @S3, @S4
F3: @S1, @S3, @S4, @Sn

3) Identify empty results (**reduction**)



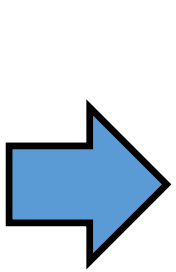


Physical View

Secondary servers

Challenge III: Distributed Query Processing

SELECT * FROM T
WHERE A > 5



T

A	B	C	D	E

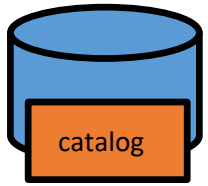
Conceptual View

Query Optimizer

Global Optimizer

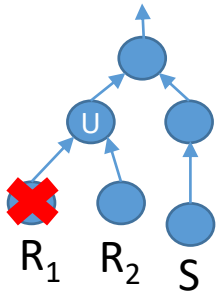
Local Optimizer

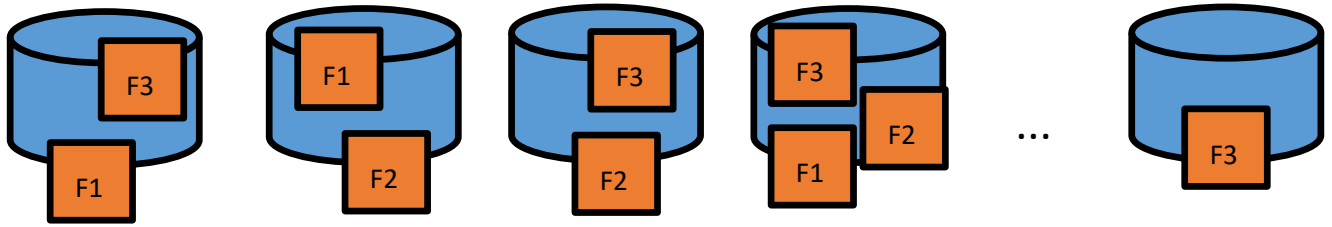
Primary server



Catalog:
T <<frag. strategy>>
F1: @S1, @S2, @S4
F2: @S2, @S3, @S4
F3: @S1, @S3, @S4, @Sn

3) Identify empty results (**reduction**)

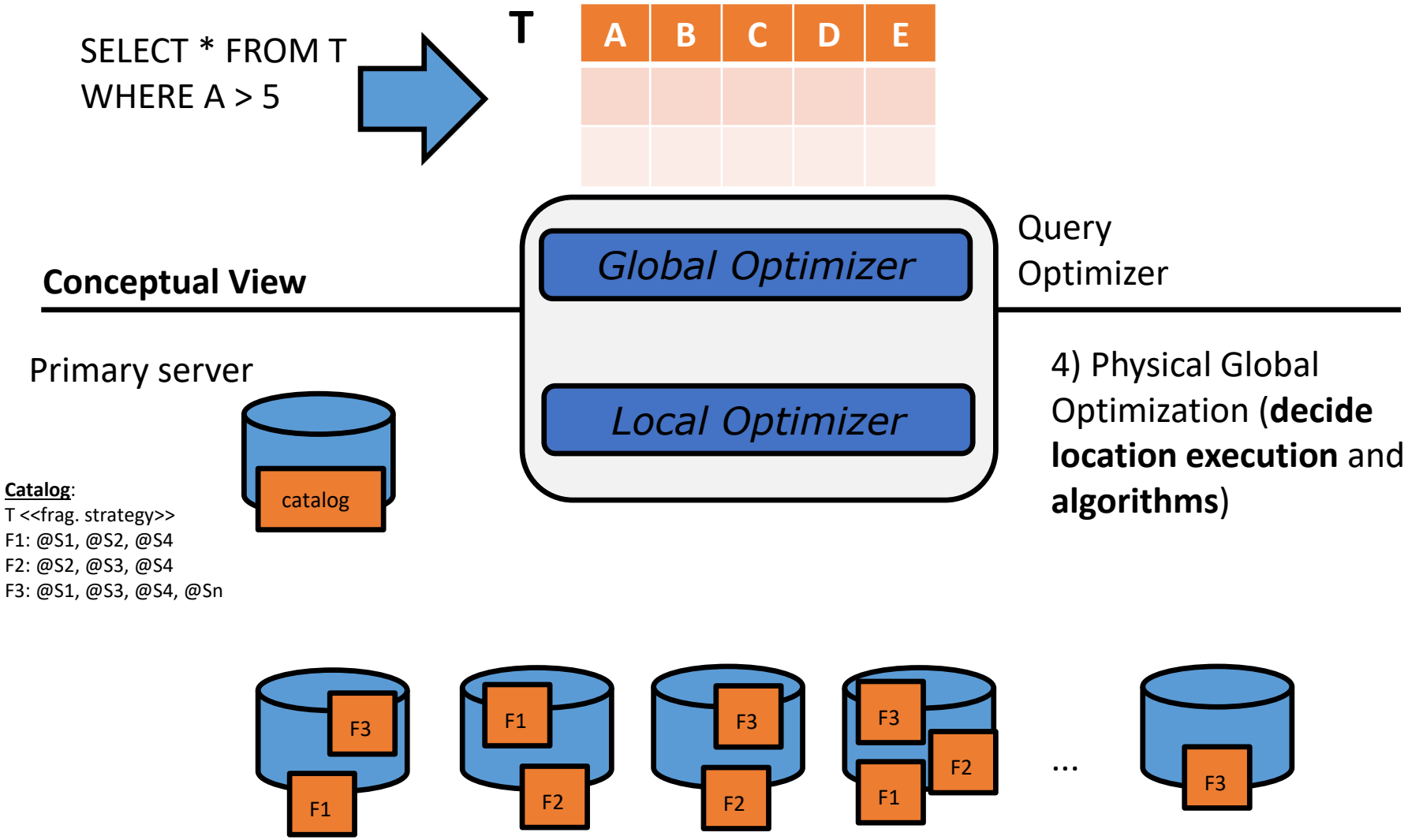




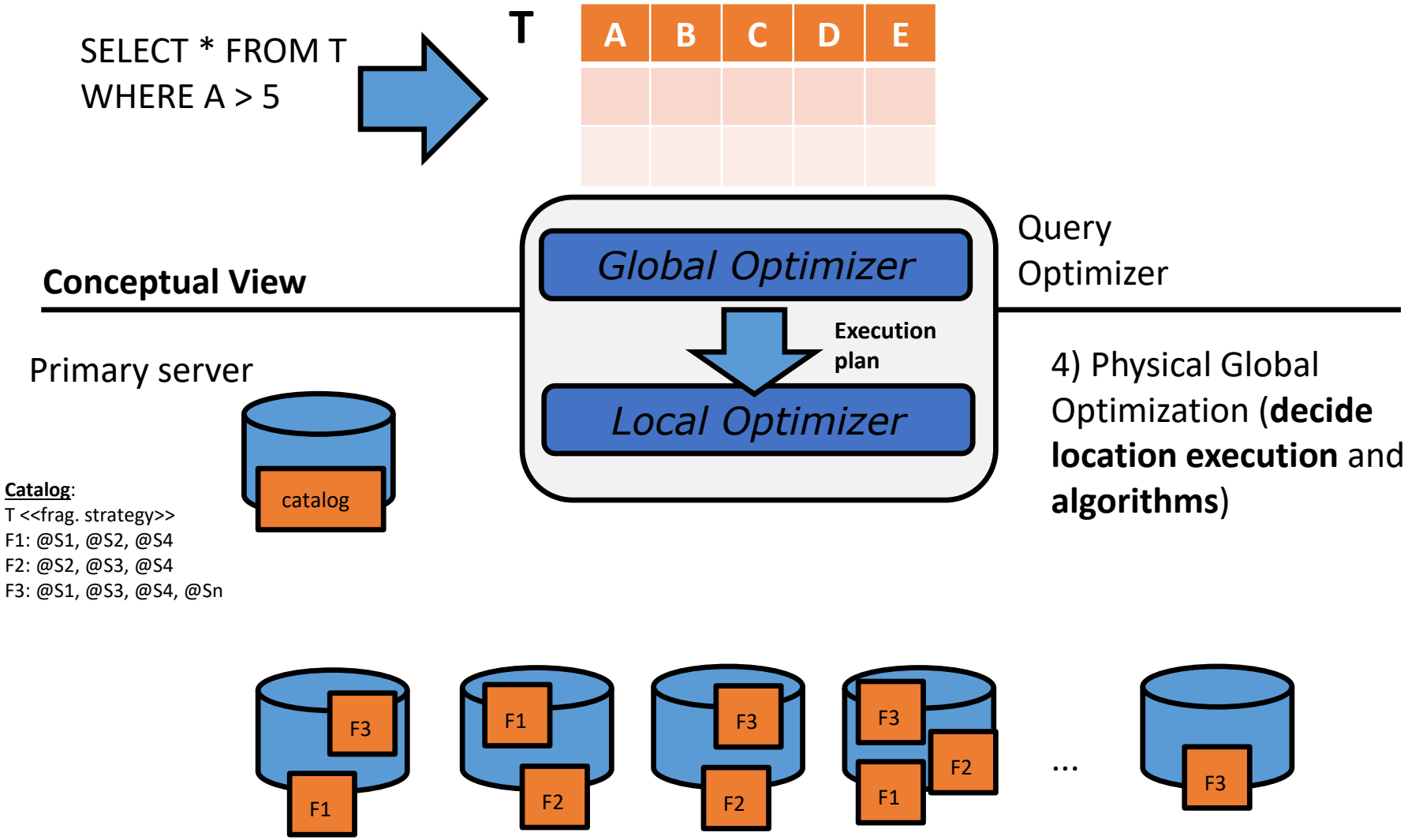
Physical View

Secondary servers

Challenge III: Distributed Query Processing



Challenge III: Distributed Query Processing



Physical Global Optimization

Factors to consider to decide the overall query schedule:

- Communication cost (*data shipping*)
  - Not that critical for LAN networks if assuming high enough I/O cost
- Fragmentation / Replication
  - Location of the fragments / replicas (global catalog)
  - Statistics about each fragment / replica (required by the cost model)
- Join Optimization
  - Joins order
  - Semi-join strategy
- How to decide the execution plan
  - Who executes what
  - Exploit parallelism (!)

PHYSICAL GLOBAL OPTIMIZATION

DISTRIBUTED AND PARALLEL QUERY PROCESSING

Global Physical Optimizer

- **Objective**: Transforms an optimize the output of the syntactic optimizer into an efficient access plan
  - Replaces the logical query operators by specific algorithms (plan operators) and access methods
  - Decides in which order to execute them
- This is done in four steps…
  1. Generating the process tree
     1. The process tree is a dataflow diagram that pipes data through a graph of physical query operators
  2. Enumerating alternative but equivalent *access plans*
     1. Generate Alternatives
     2. Site Selection
  3. Estimating the cost of each alternative access plan
     1. Using available statistics regarding the physical state of the system
  4. Selecting the best solution
     1. The best solution is then scheduled and passed to the local optimizers

Steps of the Global Physical Optimizer

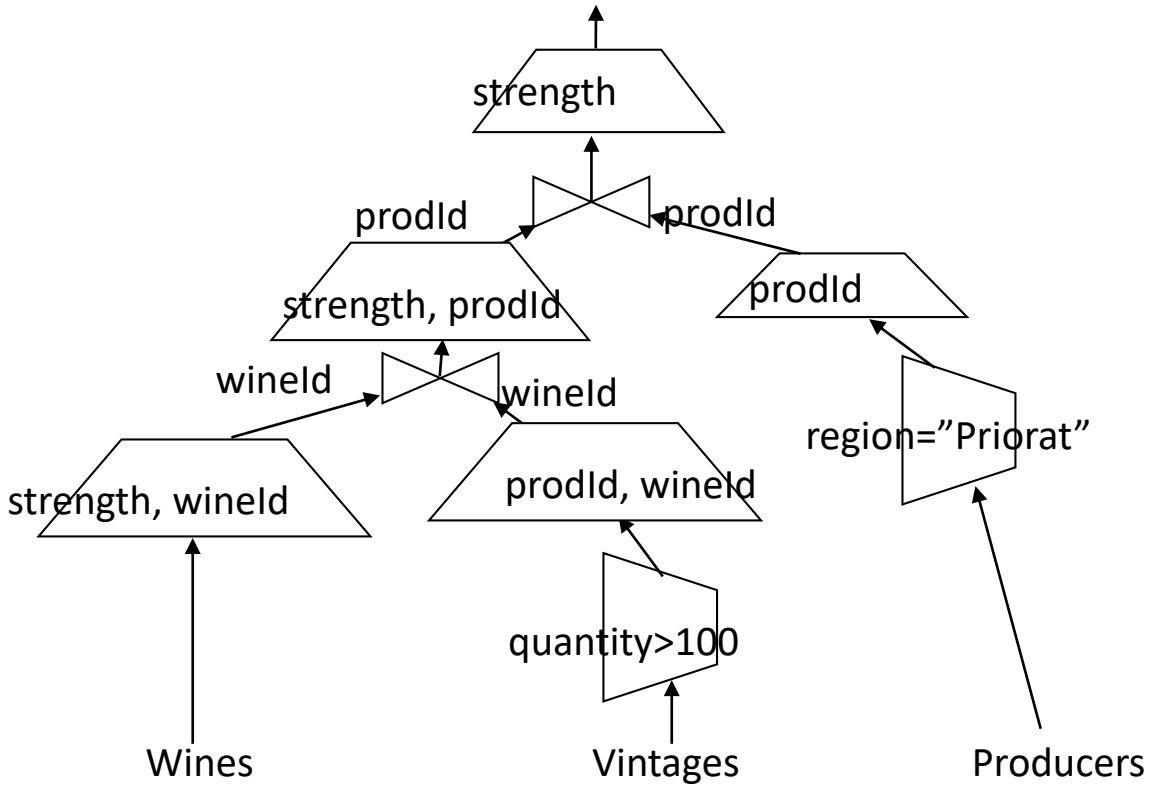
Process tree operations:

- Physical selection (+ projection)
- Physical join (+ projection)
- Set operations:
  - Union (+ projection)
  - Difference (+ projection)
- Duplicate removal
- Sorting
- Grouping
- Aggregation

1. Generating the process tree
   1. The process tree is a dataflow diagram that pipes data through a graph of physical query operators
2. Enumerating alternative but equivalent *plans (generate alternatives)*
   1. Decide the order in which to execute the operators
   2. Decide in which site execute each operator
3. Estimating the cost of each alternative access plan
   1. Using available statistics regarding the physical state of the system
4. Selecting the best solution
   1. The best solution is then scheduled and passed to the local optimizers

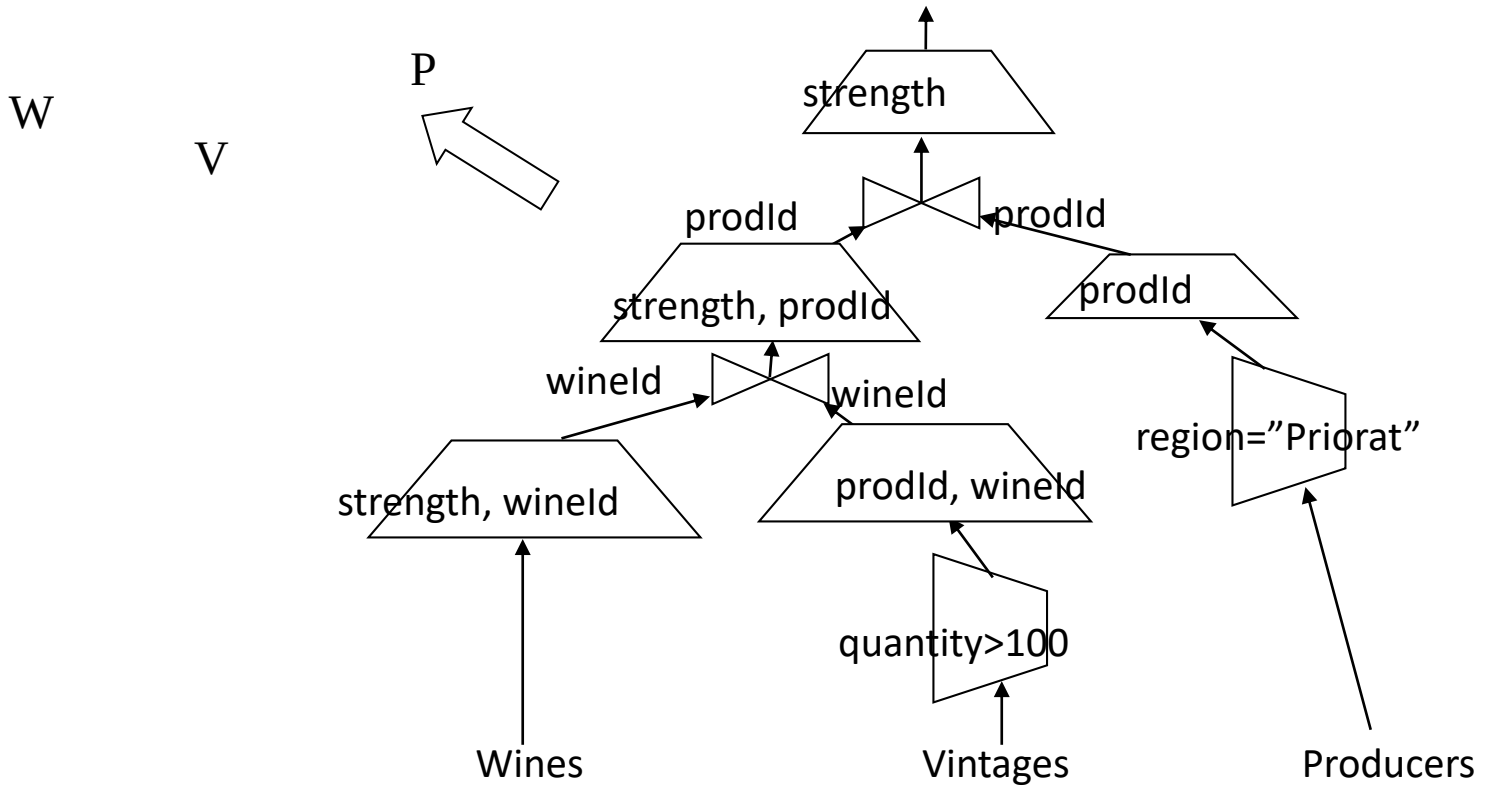
Generating the Process Tree

```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineId=w.wineId
        AND p.prodId=v.prodId
        AND p.region="Priorat"
        AND v.quantity>100;
```



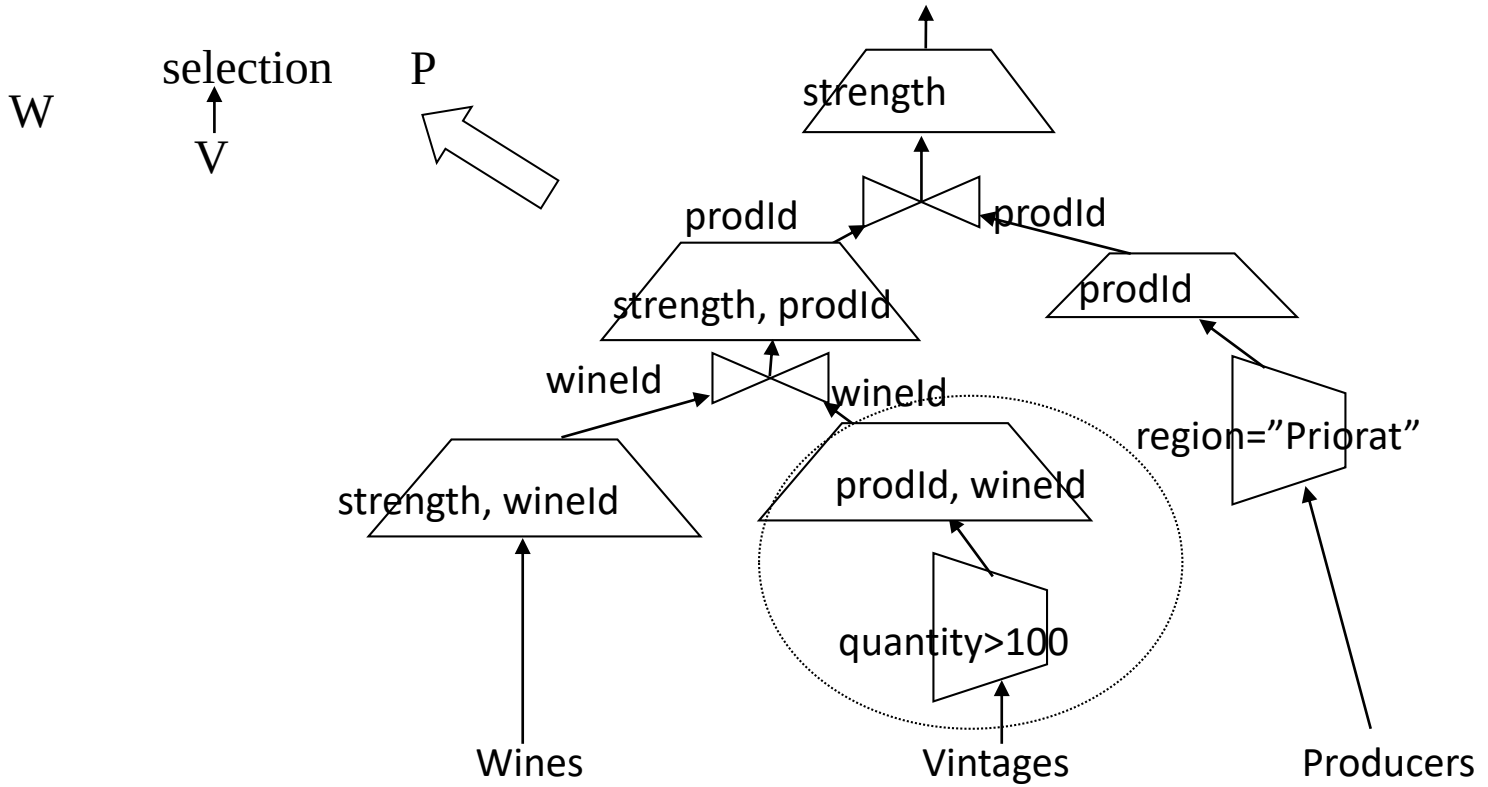
Generating the Process Tree

```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineId=w.wineId
        AND p.prodId=v.prodId
        AND p.region="Priorat"
        AND v.quantity>100;
```



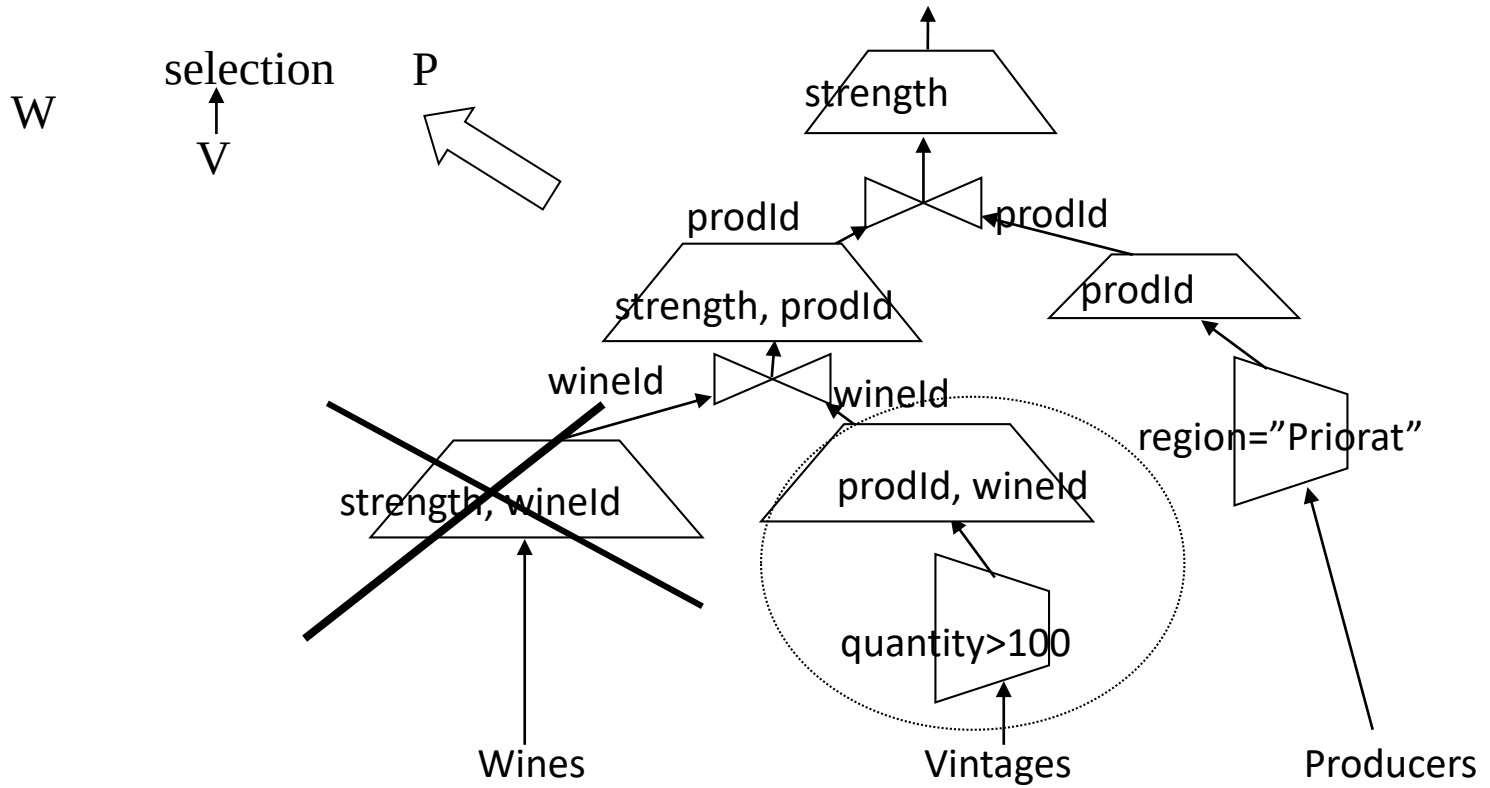
Generating the Process Tree

```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineId=w.wineId
        AND p.prodId=v.prodId
        AND p.region="Priorat"
        AND v.quantity>100;
```



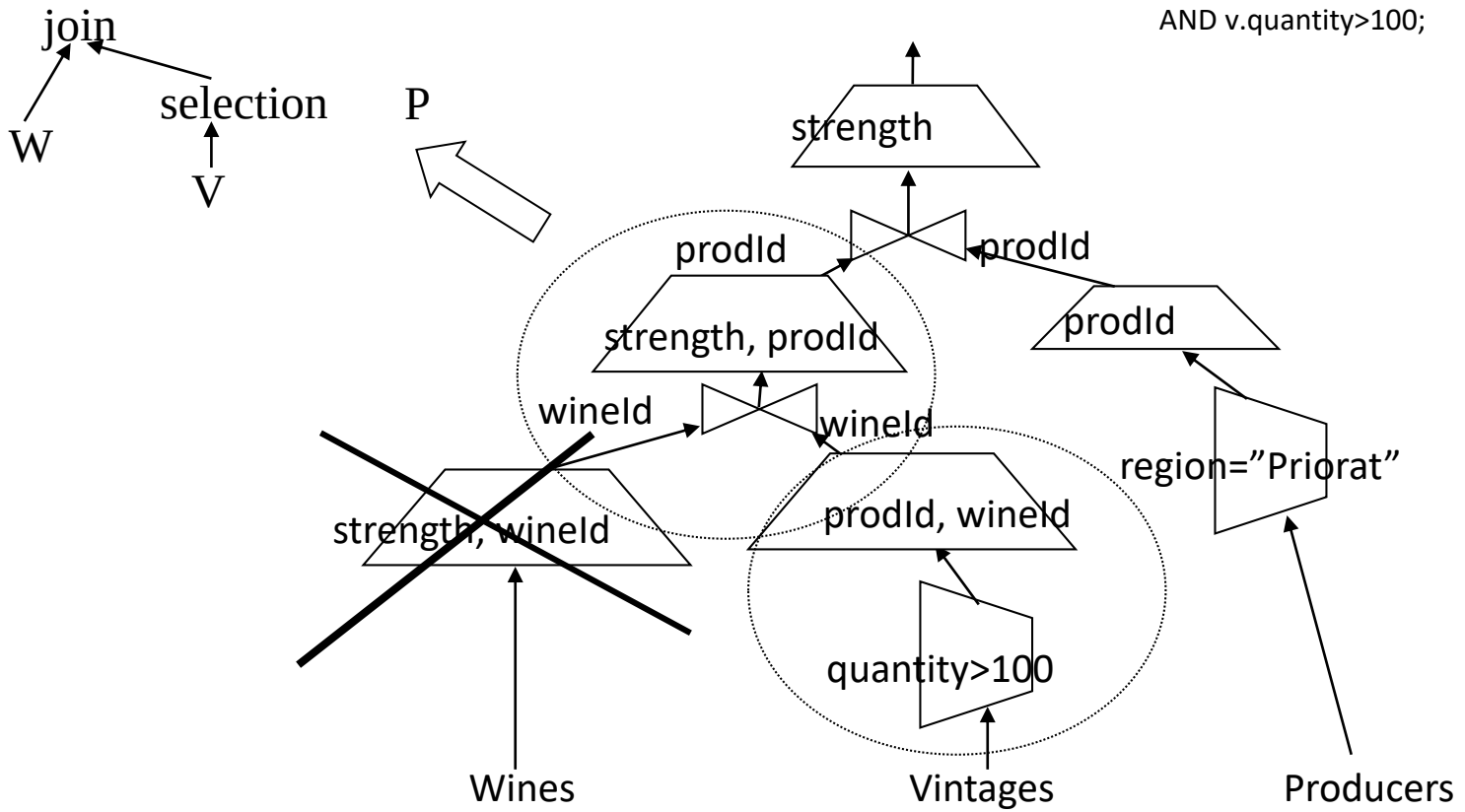
Generating the Process Tree

```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineId=w.wineId
        AND p.prodId=v.prodId
        AND p.region="Priorat"
        AND v.quantity>100;
```



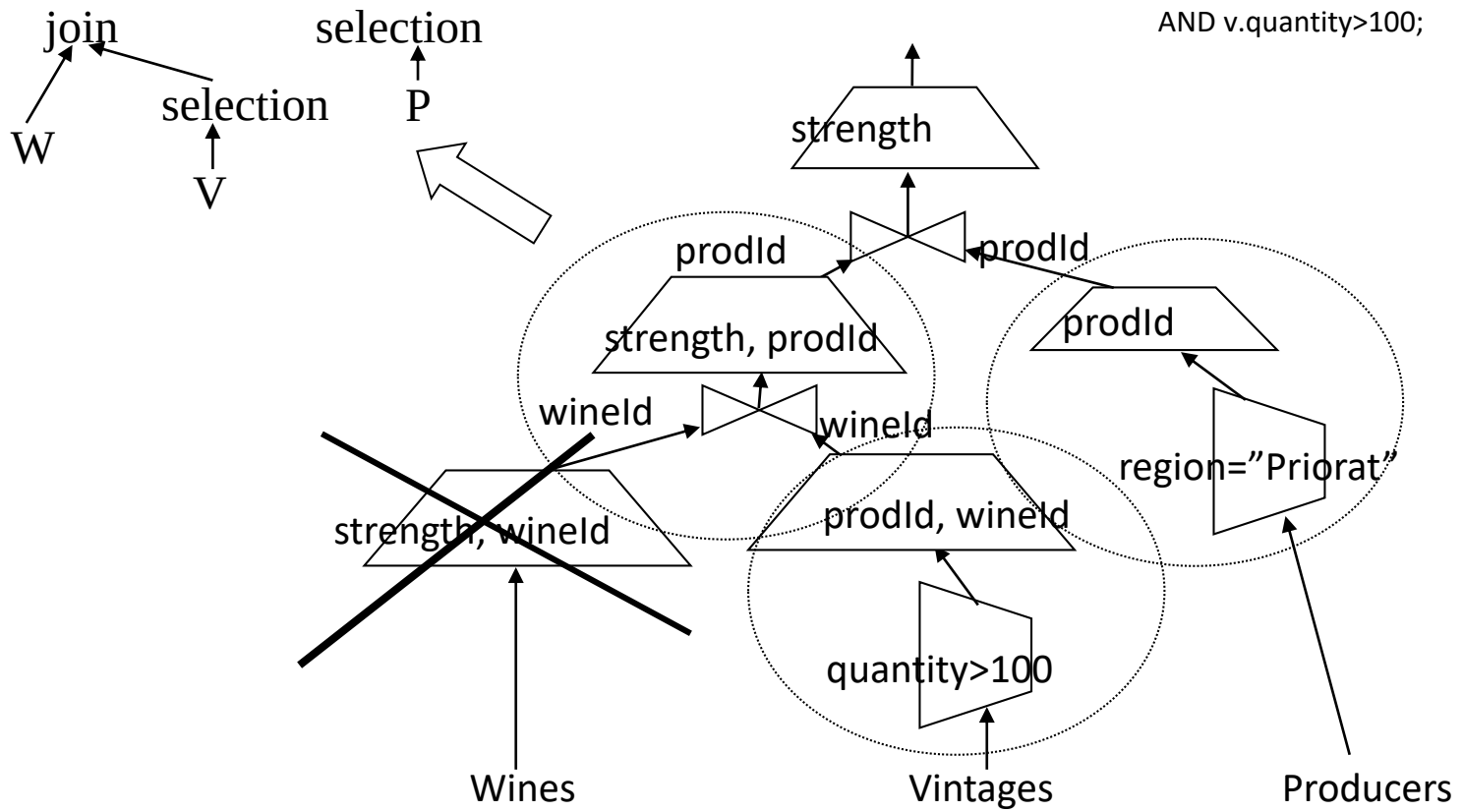
Generating the Process Tree

```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineId=w.wineId
        AND p.prodId=v.prodId
        AND p.region="Priorat"
        AND v.quantity>100;
```



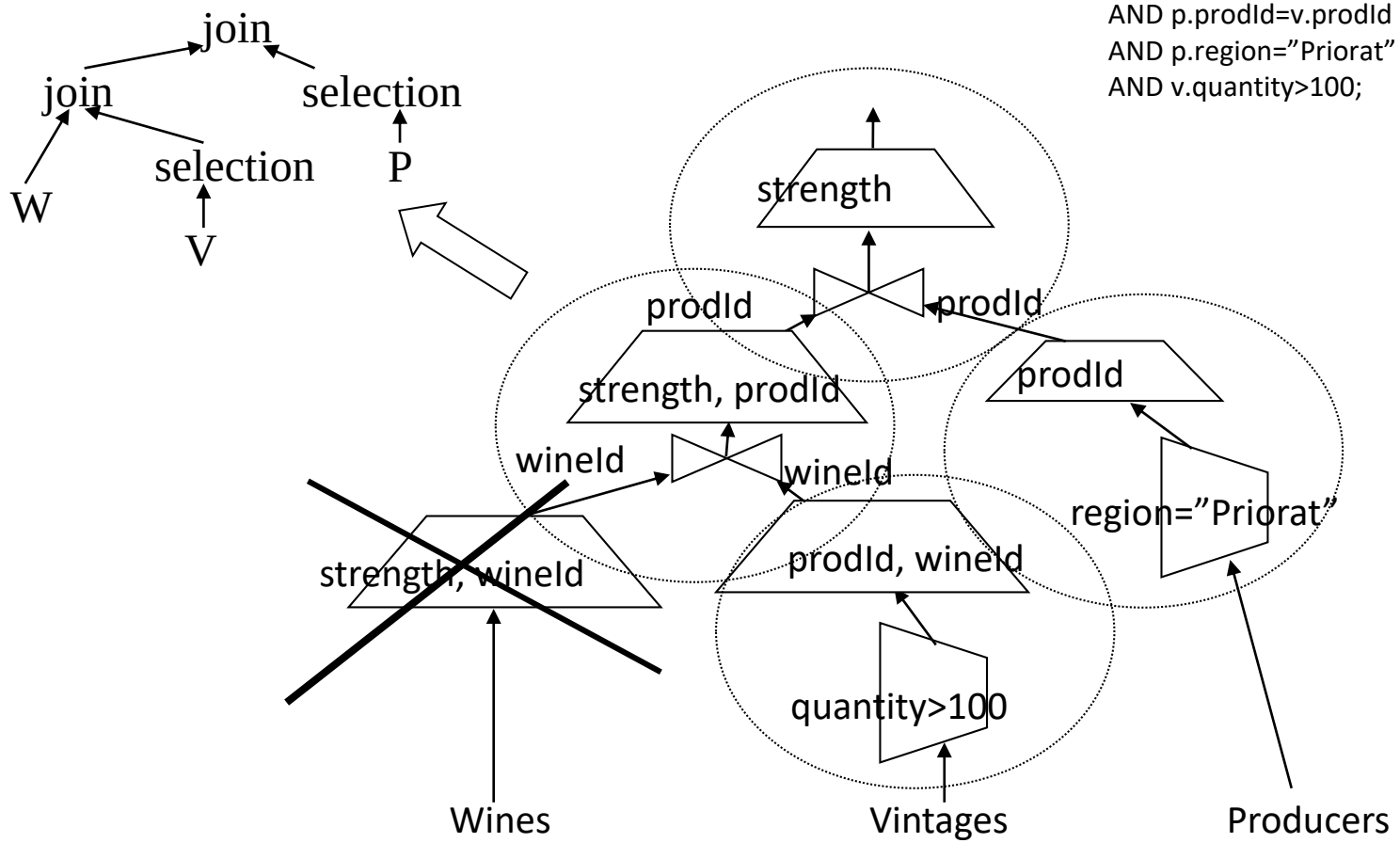
Generating the Process Tree

```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineId=w.wineId
        AND p.prodId=v.prodId
        AND p.region="Priorat"
        AND v.quantity>100;
```

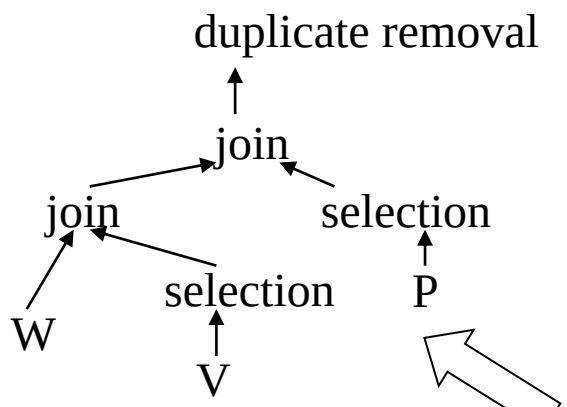


Generating the Process Tree

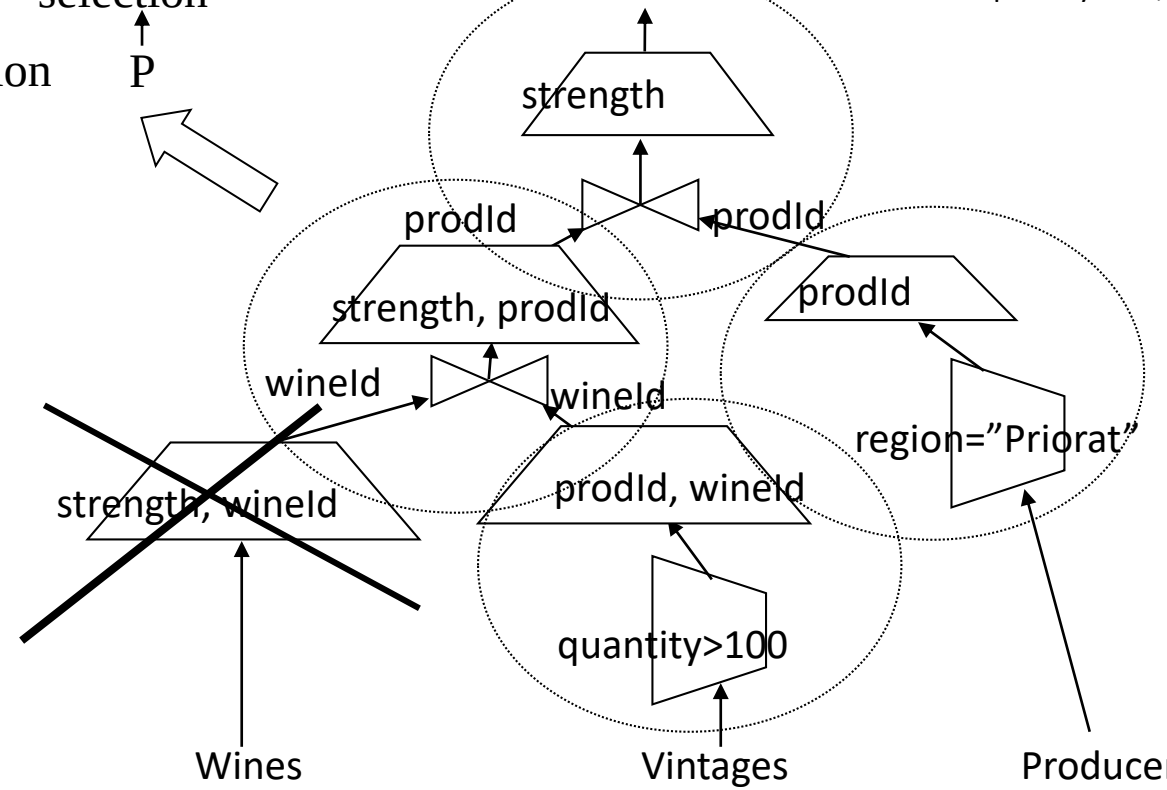
```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineId=w.wineId
        AND p.prodId=v.prodId
        AND p.region="Priorat"
        AND v.quantity>100;
```



Generating the Process Tree




```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineId=w.wineId
        AND p.prodId=v.prodId
        AND p.region="Priorat"
        AND v.quantity>100;
```

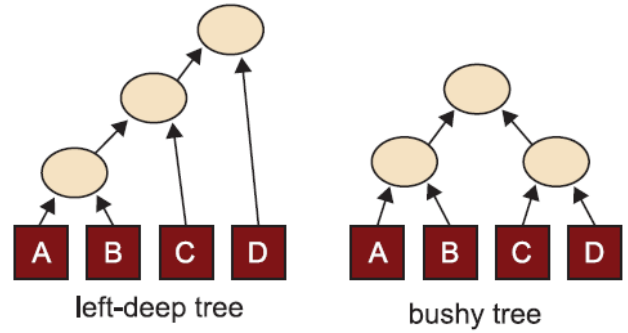


Steps of the Global Physical Optimizer

1. Generating the process tree
    1. The process tree is a dataflow diagram that pipes data through a graph of physical query operators
2. Enumerating alternative but equivalent *plans (generate alternatives)*
    1. Decide the order in which to execute the operators
    2. Decide in which site execute each operator
3. Estimating the cost of each alternative access plan
    1. Using available statistics regarding the physical state of the system
4. Selecting the best solution
    1. The best solution is then scheduled and passed to the local optimizers

Decide the Execution Order

- The process tree execution order is set by one of the following strategies:
  - **Pipelining** operators (left and right deep trees)
    - At most, one operator is an intermediate result
  - **Maximizing parallelism** (bushy tree)
    - Both operators can be intermediate results



- This decision cannot typically be chosen (it is predefined in the distributed system). For example:
  - Bushy Tree: MapReduce, Spark
  - Right-Deep Tree: Aggregation Framework (MongoDB)

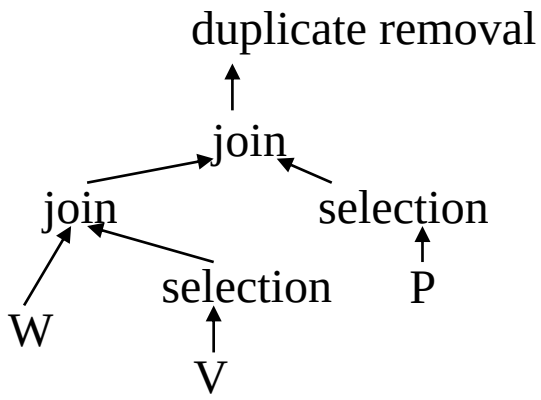
Decide the Execution Order: Maximize Parallelism

- In a bushy tree, each operator benefits from the fragments created and, ideally, each operator uses all fragments to parallelise its execution
- However, some operators require inter-operator communication due to internal dependencies in the execution that hinder parallelism
  - **Non-blocking operators**: the operator can be executed in each fragment independently, without communicating
    - Unary operators are typically fully parallelizable: selection, projection. Union, despite being a binary operator, is also an exemple of non-blocking operator
  - **Blocking operators**: to execute the operator, fragments must communicate to each other in order to solve some internal dependencies (e.g., two attributes from two instances have the same value to determine if joinable)
    - Binary operators are typically not fully parallelizable: join, difference
    - When fragments must communicate parallelism must stop and wait. These are typically called **synchronization barriers** and by definition generate stalls

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

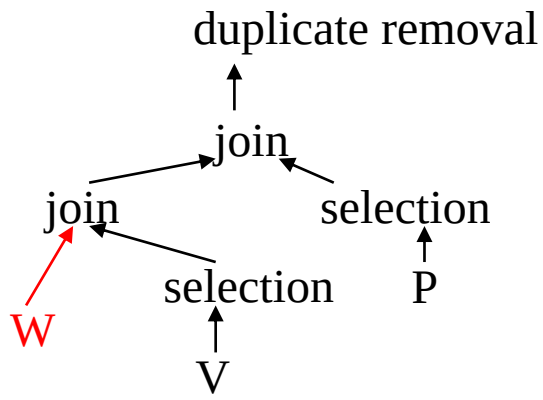
- Pipelining:



Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Pipelining:

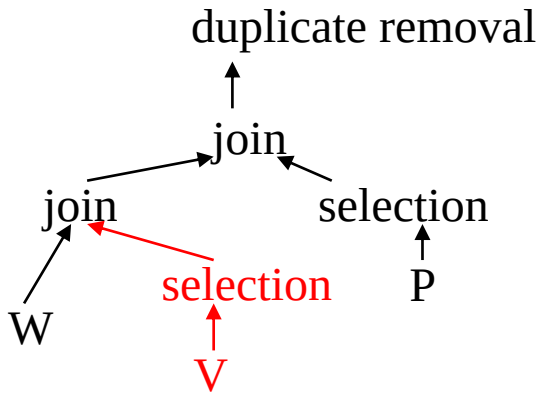


@S_N read(W) starts and its results are pipelined to the join

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Pipelining:

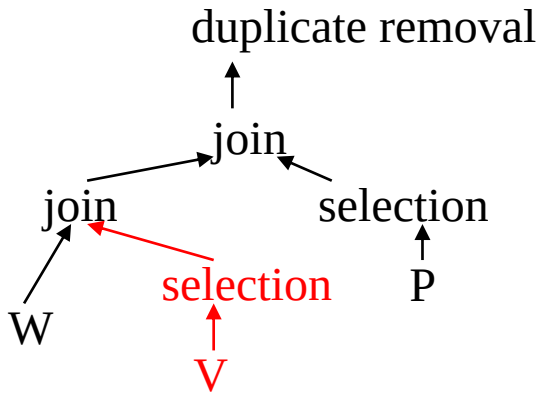


@S_M read(V) starts and it pipelines its results to selection(V). When the selection starts executing, it pipelines its results to the join

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Pipelining:



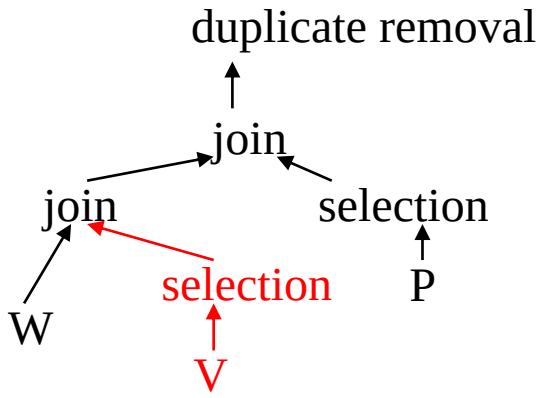
@S_M read(V) starts and it pipelines its results to selection(V). When the selection starts executing, it pipelines its results to the join
Realize the read and selection operators are **non-blocking**, since they can execute and send data to the next operator

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Pipelining:

We say the read and selection operations are **non-blocking** because there are no inter-operation dependencies when executed



@S_M read(V) starts and it pipelines its results to selection(V). When the selection starts executing, it pipelines its results to the join
Realize the read and selection operators are **non-blocking**, since they can execute and send data to the next operator

Decide the Execution Order: Example

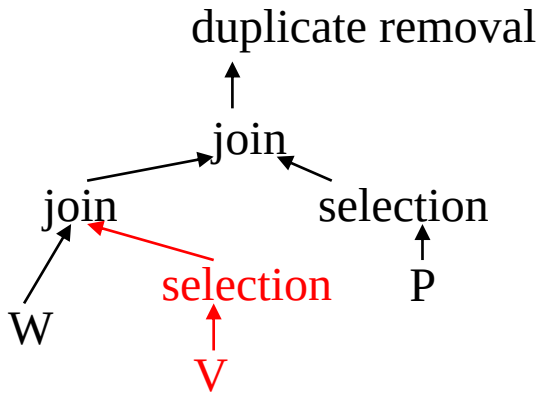
- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Pipelining:

We say the read and selection operations are **non-blocking** because there are no inter-operation dependencies when executed

Non-blocking operations generate no stalls

Stall definition: the pipeline **stalls** when an operator becomes ready and no new inputs are available



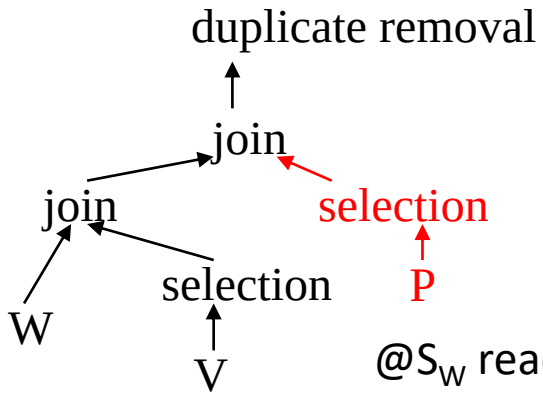
@S_M read(V) starts and it pipelines its results to selection(V). When the selection starts executing, it pipelines its results to the join

Realize the read and selection operators are **non-blocking**, since they can execute and send data to the next operator

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Pipelining:



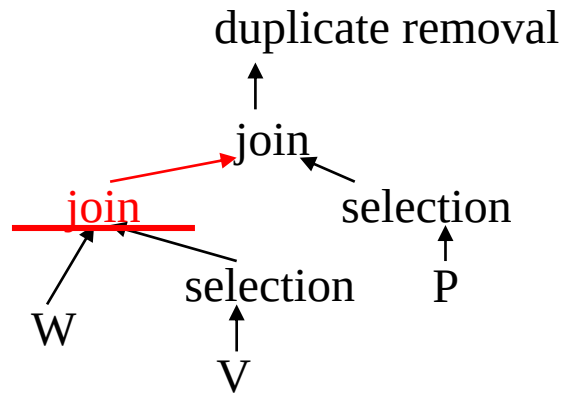
@S_W read(P) starts and it pipelines its results to selection(P). When the selection starts executing, it pipelines its results to the join. **Both are non-blocking**

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Pipelining:

@S_Z join(W,V) needs read(W) and selection(V) to finish before starting. A join has inter-operation dependencies and therefore it is a **blocking operation** whose **synchronization barrier (sb)** generates a stall.
Once the join starts, it pipelines its results to the next join

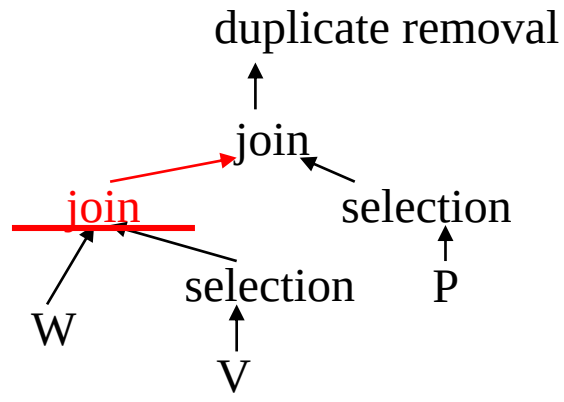


Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Pipelining:

@S_Z join(W,V) needs read(W) and selection(V) to finish before starting. A join has inter-operation dependencies and therefore it is a **blocking operation** whose **synchronization barrier (sb)** generates a stall.
Once the join starts, it pipelines its results to the next join

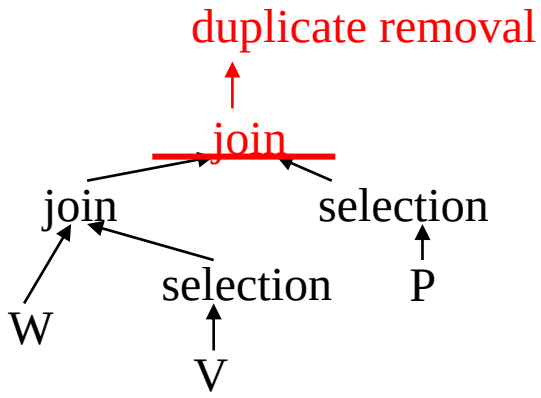


Blocking operators will always generate a synchronization barrier (i.e., a stall) when pipelining

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Pipelining:

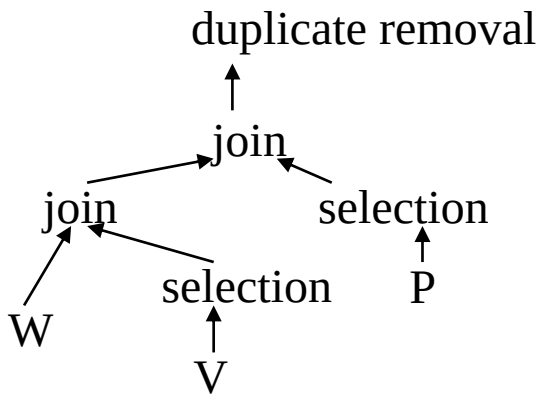


@S_T join(join(W,V), P) is a **blocking operator** generating a **synchronization barrier**. As it executes, it pipelines results to the duplicate removal that can start executing straight away since it is a **non-blocking operator**

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

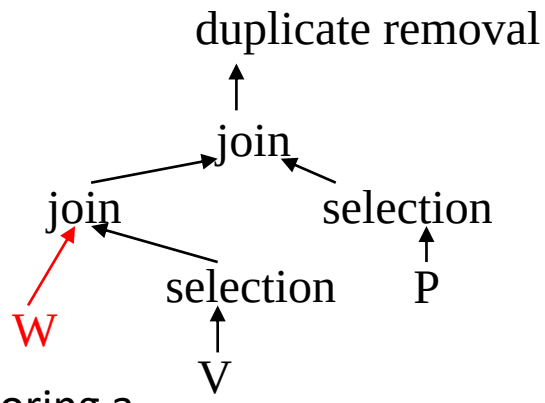
- Maximize Parallelism:



Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Maximize Parallelism:

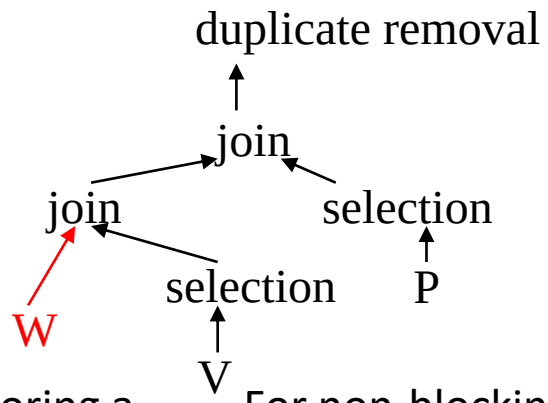


W is read from 100 sites (S_1,...,S_{100}), each of them storing a fragment of W. This read is done in parallel and each read pipelines its results to the join

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Maximize Parallelism:



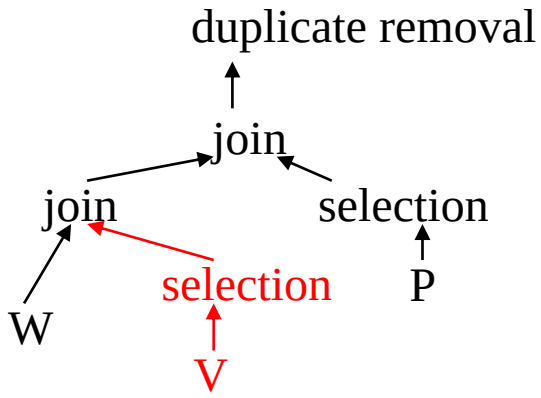
W is read from 100 sites (S_1,...,S_{100}), each of them storing a fragment of W. This read is done in parallel and each read pipelines its results to the join

For non-blocking operators the only difference is that in this strategy we also exploit **inter-operation parallelism** thanks to the fragments created

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Maximize Parallelism:

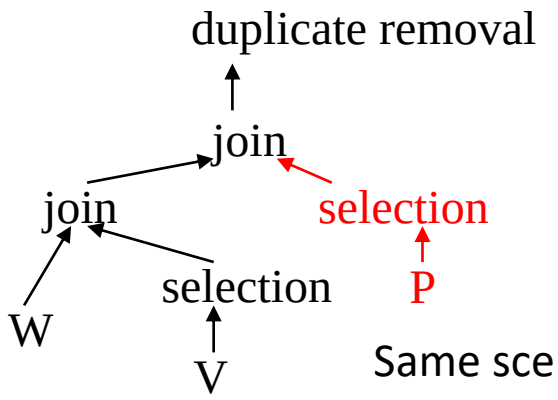


Similarly, read(V) would be executed in 100 sites (one execution per fragment). Inside each side, the partial result is pipelined to selection(V) that starts execution immediately (both are **non-blocking** operators). Finally, each result is pipelined to the join

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Maximize Parallelism:

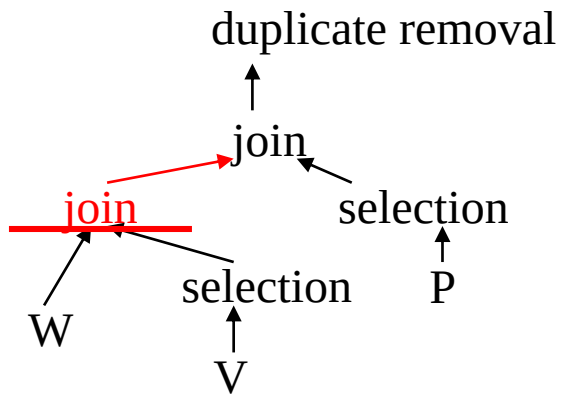


Same scenario as in the previous slide

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Maximize Parallelism:



join(W,V) needs read(W) and selection(V) to finish before starting since it is a blocking operator. Nevertheless, this join will be executed in **N*** sites and the synchronization barrier is more complex than before: each site executing W and each site executing V+selection(V) must know to which join executor (i.e., one of the N sites selected to execute the join) send its data

N*: N is now a parameter of the system

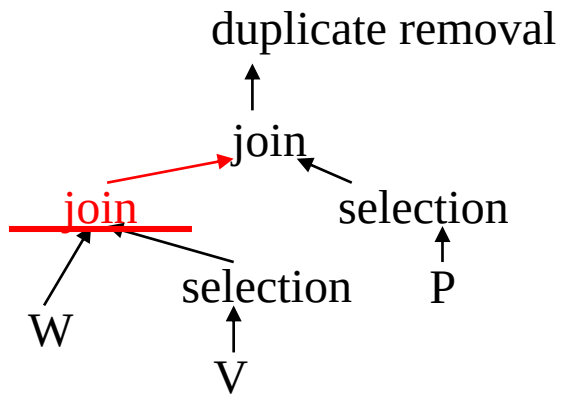
Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Maximize Parallelism:

join(W,V) needs read(W) and selection(V) to finish before starting since it is a blocking operator. Nevertheless, this join will be executed in **N*** sites and the synchronization barrier is more complex than before: each site executing W and each site executing V+selection(V) must know to which join executor (i.e., one of the N sites selected to execute the join) send its data

N*: N is now a parameter of the system



Importantly, note, this complexity is also an advantage. We can benefit from the synchronization barrier to partially parallelize the join. For example, by hashing the join attributes (e.g., %N) and sending those with the same result to the same join executor

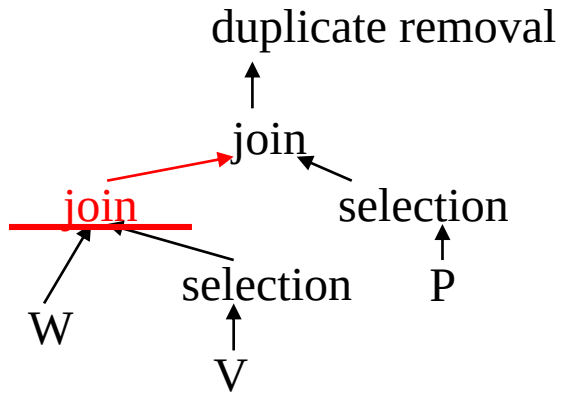
Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Maximize Parallelism:

join(W,V) needs read(W) and selection(V) to finish before starting since it is a blocking operator. Nevertheless, this join will be executed in **N*** sites and the synchronization barrier is more complex than before: each site executing W and each site executing V+selection(V) must know to which join executor (i.e., one of the N sites selected to execute the join) send its data

N*: N is now a parameter of the system



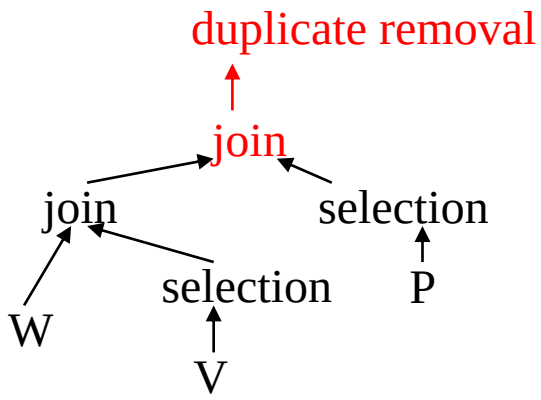
Blocking operators **still generate stalls when receiving the input data** from its operators **but its execution can be parallelized**. Thus, they are **partially blocking**

Importantly, note, this complexity is also an advantage. We can benefit from the synchronization barrier to partially parallelize the join. For example, by hashing the join attributes (e.g., %N) and sending those with the same result to the same join executor

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Maximize Parallelism:

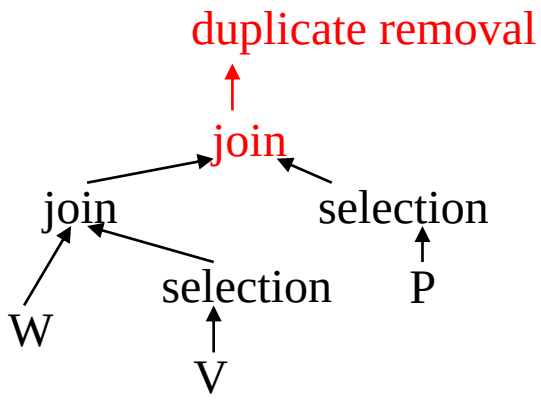


join(join(W,V), P) is partially blocking, and when executed in N sites it pipelines its results to the duplicate removal

Decide the Execution Order: Example

- **Example**: consider W, V and P be fragmented in 100 fragments. All of them replicated 3 times (assume a cluster with 500 machines; no need to consider now the specific distribution of fragments)

- Maximize Parallelism:



join(join(W,V), P) is partially blocking, and when executed in N sites it pipelines its results to the duplicate removal

The duplicate removal is also executed in N sites. The result of duplicate removal can be either stored in a distributed fashion (taking advantage of its N executors) or pipelined to a single site

Decide the Execution Order: Kinds of Parallelism

- Summing up, the following kinds of parallelism can be applied:
  - Inter-query (*always possible*)
  - Intra-query
    - Intra-operator (*parallelism maximization)*
    - Inter-operator
      - Independent: i.e., parallel branches of the process tree *(always possible)*
      - Pipelined: i.e., within the same branch (*always possible*)

Decide the Execution Order: Measuring Parallelism

- Main metrics:
  - Speed-up
    - Consider a problem of constant size
    - **How much faster would be its execution** if we add additional hardware to exploit as much as possible all kinds of parallelism?
      - Ideally, adding computing power should yield a proportional increase in performance
        - N nodes should solve the problem in 1/N time
  - Scale-up
    - Consider a problem of constant size solved in T seconds by exploiting parallelism in N nodes
    - **How much does the system scale** to larger sizes by adding additional hardware?
      - Ideally, adding computing power proportional to the increase of the problem size should yield a sustained performance
        - N nodes should solve a problem N times bigger in the same time

Steps of the Global Physical Optimizer

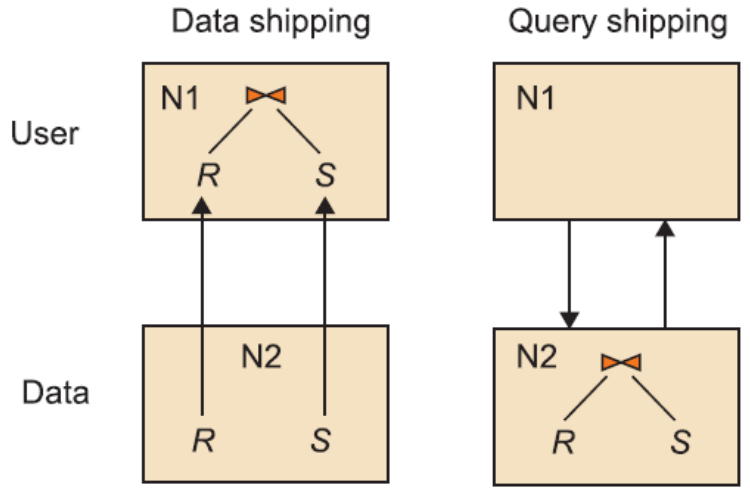
1. Generating the process tree
    1. The process tree is a dataflow diagram that pipes data through a graph of physical query operators
2. Enumerating alternative but equivalent *plans (generate alternatives)*
    1. Decide the order in which to execute the operators
    2. Decide in which site execute each operator
3. Estimating the cost of each alternative access plan
    1. Using available statistics regarding the physical state of the system
4. Selecting the best solution
    1. The best solution is then scheduled and passed to the local optimizers

Choosing the Site Execution

- Site selection
  - Unary operators: operations over replicated fragments can be executed in any of the replicas
  - Binary operators: if both fragments are not co-located, one needs to be shipped through the network. Different criteria to choose which one to send:
    - Comparing size of the relations
    - In general, it is more difficult for multi-way joins
      - Size of the intermediate joins must be considered

Choosing the Site Execution

- In general, when choosing a site, we are indeed making a decision about how two consecutive operators in the pipeline executed in different sites will communicate:
  - If an operator sends data to the next operator, we incurr in **data shipping**
    - The data moves through the network
  - If an operator is executed in many sites (typical of intra-operator parallelism) **query shipping** is the best option
    - Data does not move, and the query is sent instead
    - Avoids transferring large amount of data
- Hybrid strategies



Steps of the Global Physical Optimizer

1. Generating the process tree
    1. The process tree is a dataflow diagram that pipes data through a graph of physical query operators
2. Enumerating alternative but equivalent *plans (generate alternatives)*
    1. Decide the order in which to execute the operators
    2. Decide in which site execute each operator
3. Estimating the cost of each alternative access plan
    1. Using available statistics regarding the physical state of the system
4. Selecting the best solution
    1. The best solution is then scheduled and passed to the local optimizers

Estimating each Alternative Plan Cost

For each alternative generated, we estimate its plan via a model cost. Models costs typically measure either the respone time or resource consumption:

- Response Time (latency)
  - Time needed to execute a query (user's clock)
  - Benefits from parallelism
    - Operations divided into N operations
- Resources Used (throughput)
  - Sum of local cost and communication cost
    - Local cost
      - Cost of central unit processing (#cycles),
      - Unit cost of I/O operation (#I/O ops)
    - Communication cost
      - Commonly assumed it is linear in the number of bytes transmitted
      - Cost of initiating a message and sending a message (#messages)
      - Cost of transmitting one byte (#bytes)
  - Knowledge required
    - Size of elementary data units processed
    - Selectivity of operations to estimate intermediate results
  - Does not account the usage of parallelisms (!)
- Hybrid solutions

Examples of Model Costs

- Parameters:
  - Local processing:
    - Average CPU time to process an instance (T_{cpu})
    - Number of instances processed (#inst)
    - I/O time per operation ($T_{I/O}$)
    - Number of I/O operations (#I/Os)
  - Global processing:
    - Message time (T_{Msg})
    - Number of messages issued (#msgs)
    - Transfer time (send a byte from one site to another) (T_{TR})
    - Number of bytes transferred (#bytes)
      - It could also be expressed in terms of packets
- Calculations:

Resources = T_{cpu} * #inst + $T_{I/O}$ * #I/Os + T_{Msg} * #msgs + T_{TR} * #bytes

Respose Time = T_{cpu} * $seq_{\#inst}$ + $T_{I/O}$ * $seq_{\#I/Os}$ + T_{Msg} * $seq_{\#msgs}$ + T_{TR} * $seq_{\#bytes}$

The alternative with the lowest cost is considered the best one

Realize such alternative schedules the site (or sites) where to execute each operation and the communication strategy (query or data shipping) between operators executed in different sites

This information is passed to the local physical optimizers (identical to that of a centralized database) that *execute each query piece* and are the ultimate responsibles to execute the query in parallel

Summary

- Phases of the Distributed Query Processing
  - Semantic
  - Syntactic (syntactic tree, data localization, reduction)
  - Physical (global and local optimization)
- Physical Global Optimization
  - Process tree
  - Generate alternatives
    - Execution strategy
      - Kinds of parallelism
      - Parallelism metrics
    - Site selection
  - Evaluating the generated alternatives
    - Cost models
  - Select the best alternative

Bibliography

- M.T. Özsu and P. Valduriez. *Principles of distributed database systems*. Second edition. Prentice Hall, 1999
- G. Graefe. *Query Evaluation Techniques*. In ACM Computing Surveys, 25(2), June 1993
- L. Liu, M.T. Özsu (Eds.). Encyclopedia of Database Systems. Springer, 2009