

# FluidB: Adaptive storage layout using reversible relational operators

Christos Perivolaropoulos

University of Edinburgh

May 2022

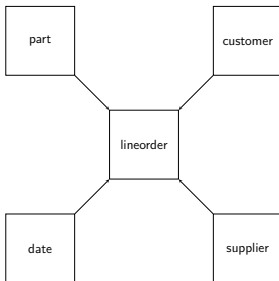
# FluiDB at a glance

- FluiDB is an intermediate result (IR) recycling, in-memory RDBMS
- FluiDB materializes all intermediate results and garbage collects when she runs out of space, unifying **query planning and IR recycling**
- Radical approach to IR recycling: **adapt** data layout to the workload:
  - ▶ enable **efficient plans**
  - ▶ constrained (quality) **budget**
- The main novelty relates to the introduction of **reversible relational operations** which affords a new perspective on query planning and view selection.

# Example 1: Workload based on template query

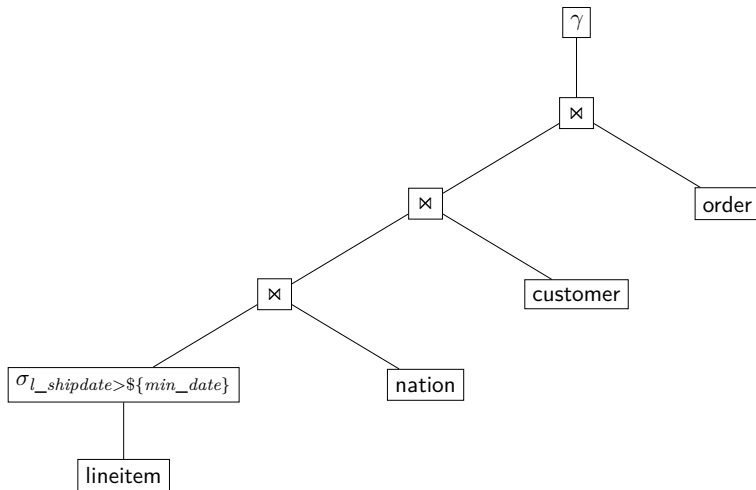
`${min_date}` is instantiated for each query in the workload.

```
select      n_name, avg(l_discount)
from        lineitem, customer, nation, order
where       l_orderkey = o_orderkey
and         c_custkey = o_custkey
and         c_nationkey = n_nationkey
and         l_shipdate > ${min_date}
group by    n_name
```

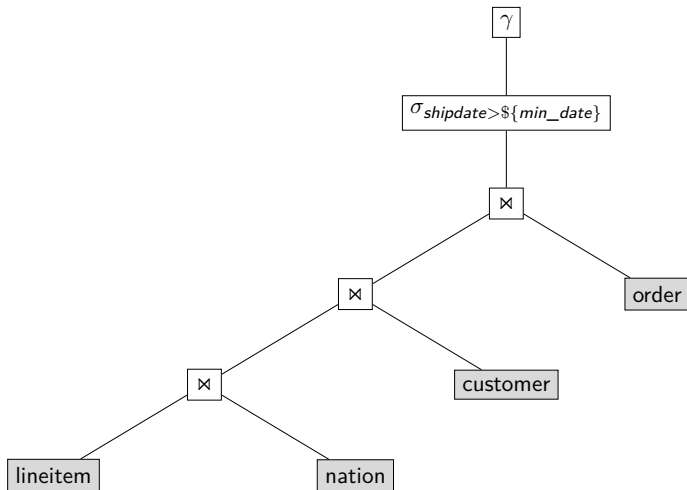


# Example 1: Traditional single-query plan

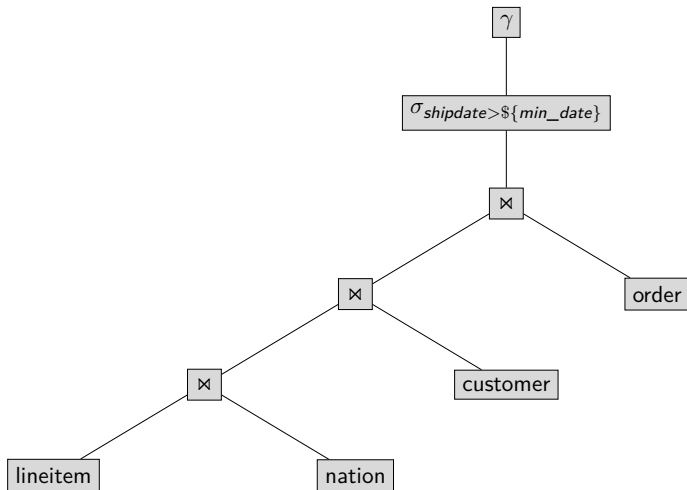
Selection push down



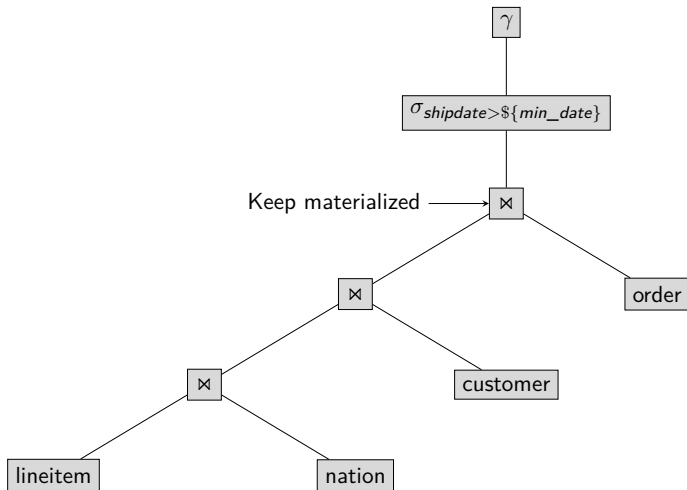
## Example 1: Adapt to workload



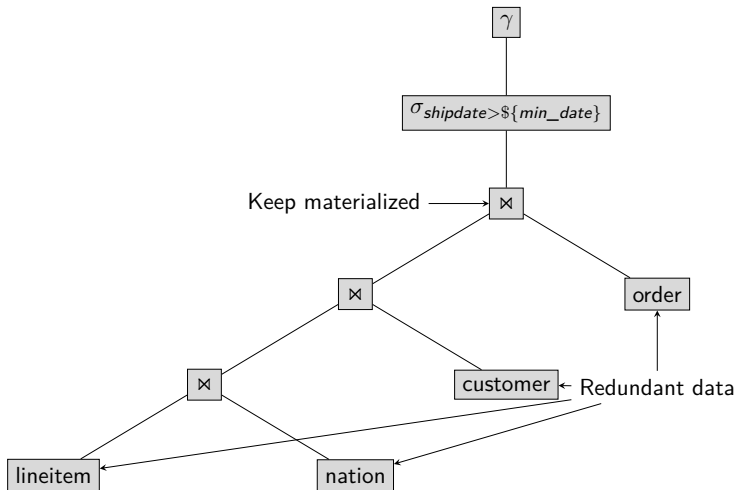
## Example 1: Adapt to workload



## Example 1: Adapt to workload

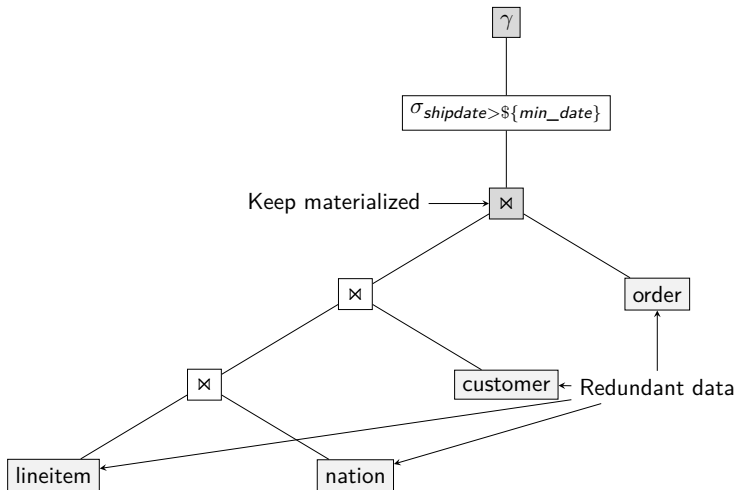


## Example 1: Adapt to workload

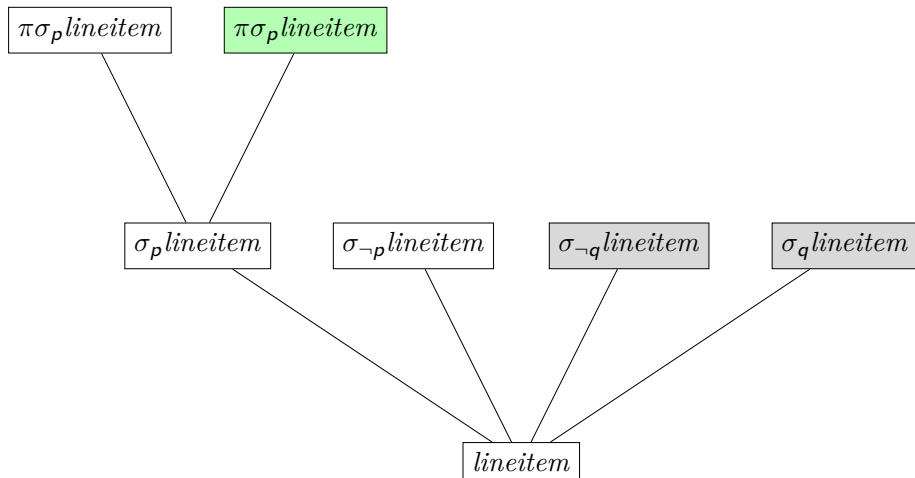




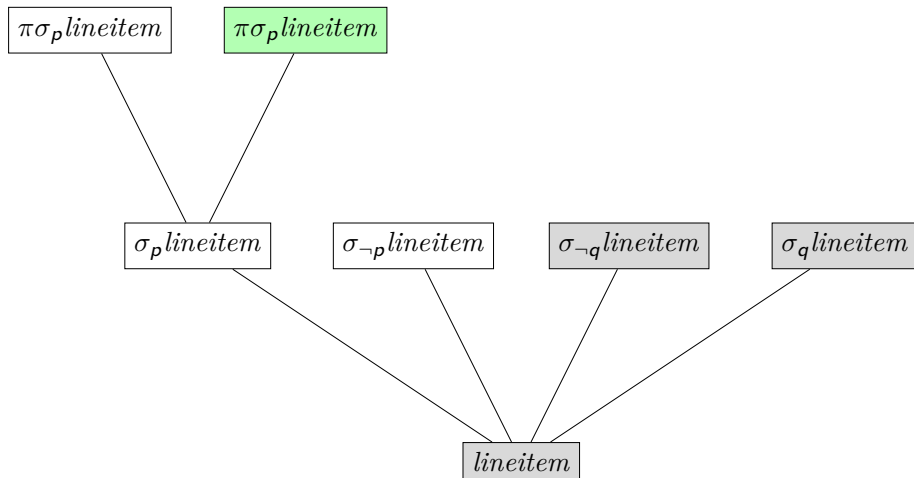
## Example 1: Adapt to workload



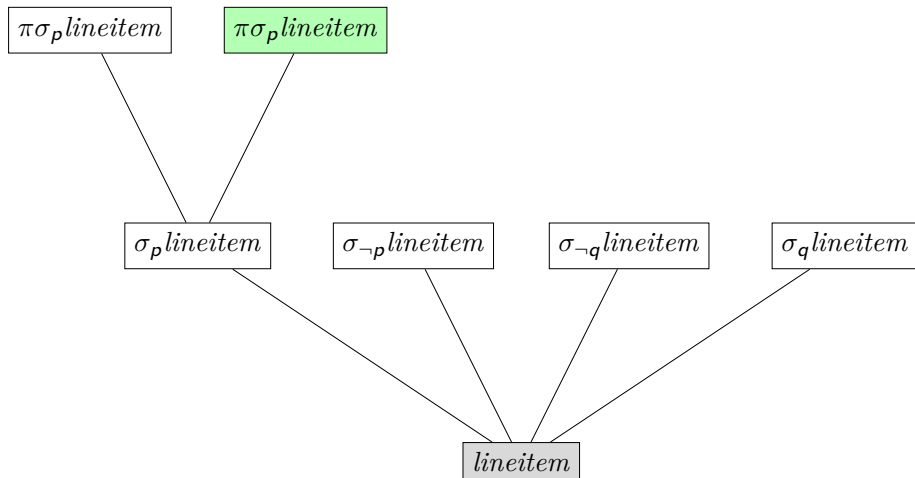
## Example 2: Plan with missing primary data



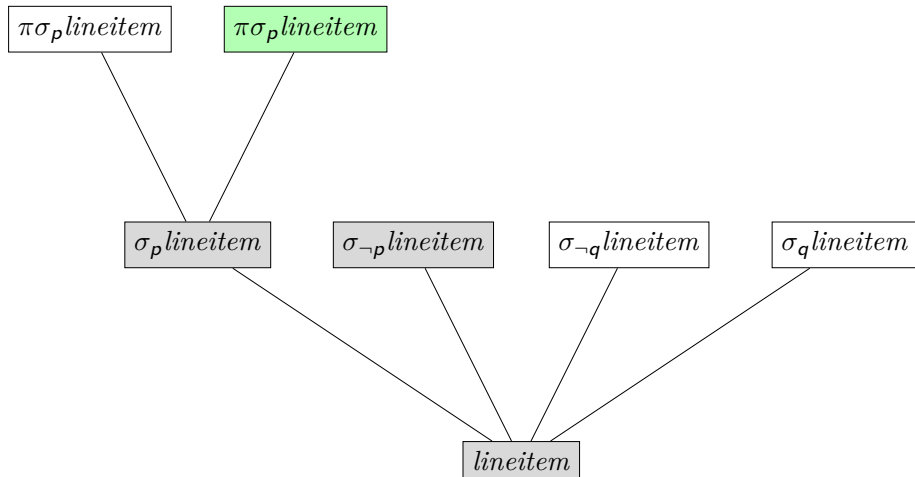
## Example 2: Plan with missing primary data



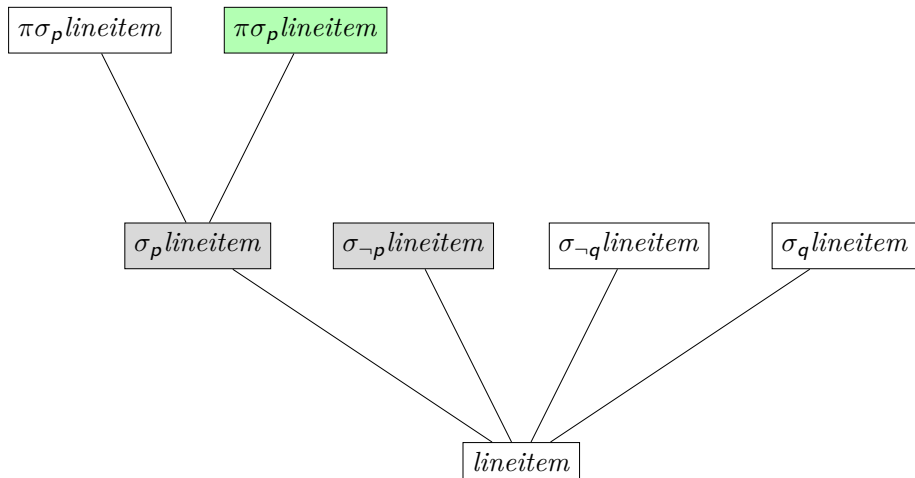
## Example 2: Plan with missing primary data



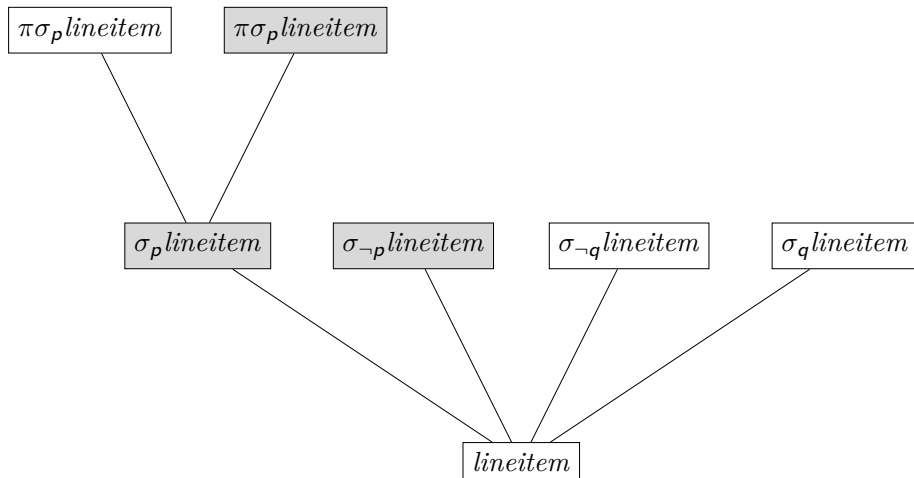
## Example 2: Plan with missing primary data



## Example 2: Plan with missing primary data



## Example 2: Plan with missing primary data

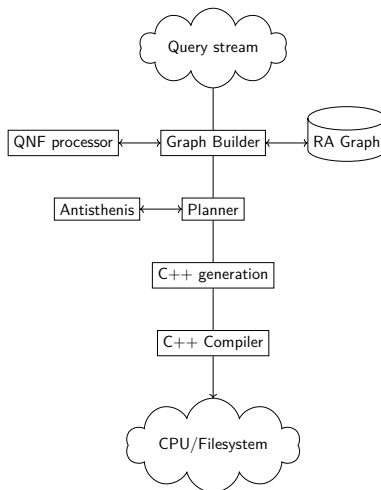


# The interesting components

- Graph management and query normal form representation
- Logical planning infrastructure
- Antisthenis: An incremental numeric evaluation system for cost estimation.
- Logical planning algorithm and garbage collector



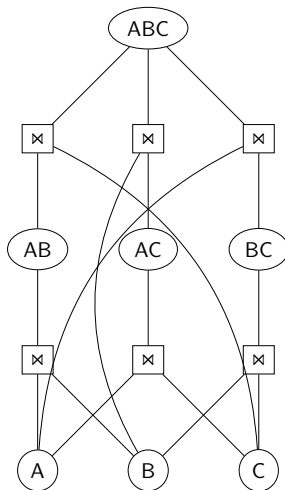
# Architecture



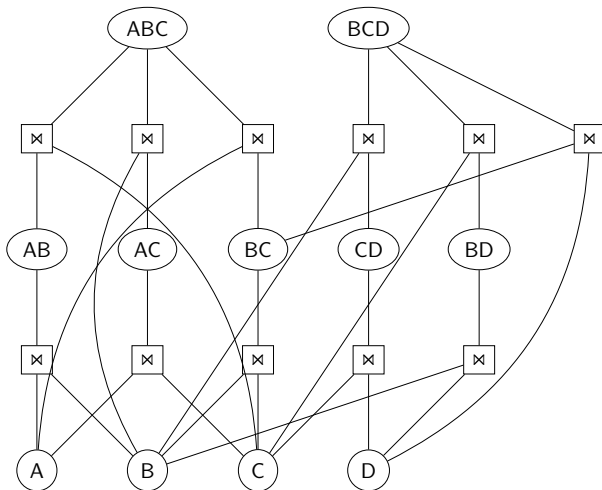
# Logical planning

- Bipartite query graph – RA operations/relations unified for all queries
- Join ordering enumeration
- QNF –  $\pi\sigma(Q_1 \times Q_2 \times \dots)$  or  $\gamma\sigma(Q_1 \times Q_2 \times \dots)$

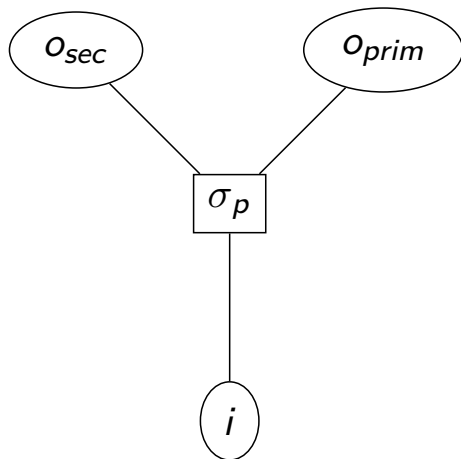
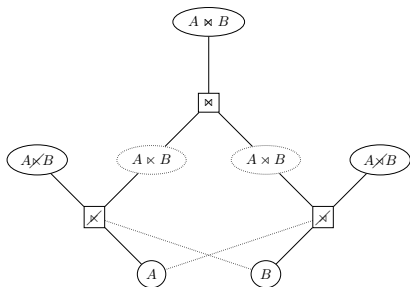
# AND/OR DAG (join ordering enumeration)



# AND/OR DAG (join ordering enumeration)



# Reversible operators

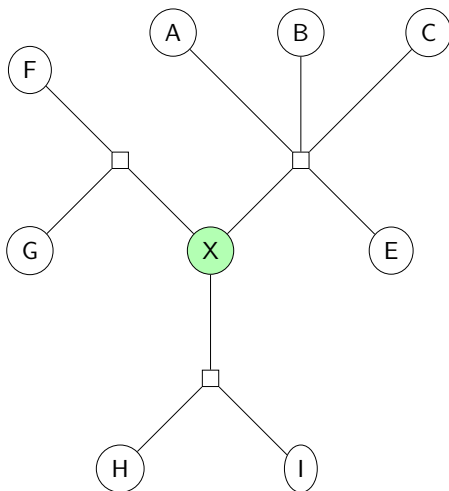


## Reversible operators



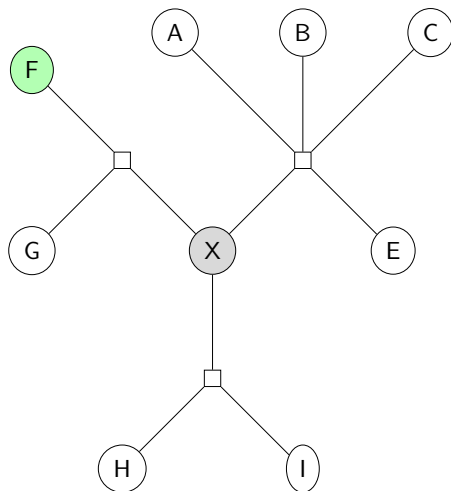
# Planning algorithm

Materialize node X (if it is not already materialized)



# Planning algorithm

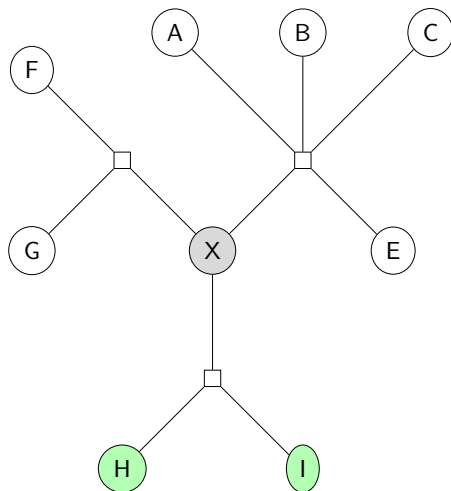
For each (materializable) input set





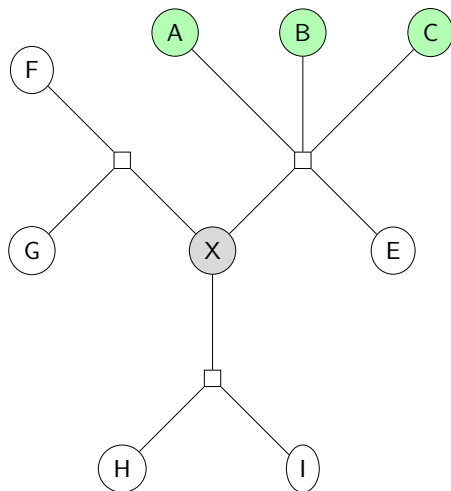
# Planning algorithm

For each (materializable) input set



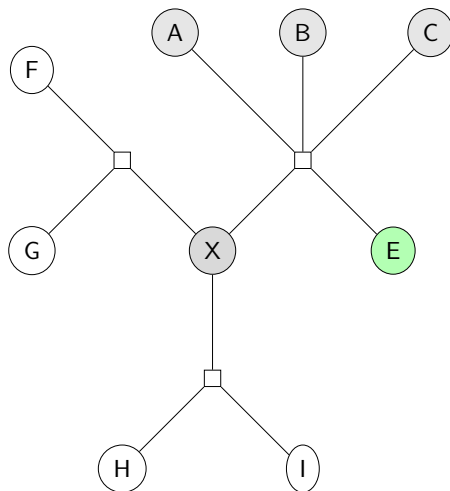
# Planning algorithm

For each (materializable) input set



# Planning algorithm

For each output set containing the node



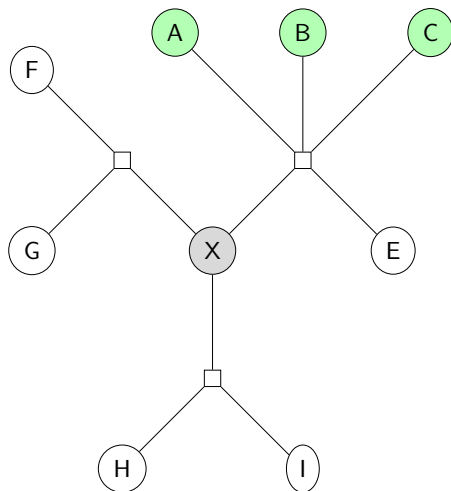
# Planning algorithm

Schedule the current branch

$$priority = \sum_{op \in \text{planned ops}} cost(op) + \sum_{h \in hist.} cost_{stoch.}(h) + \sum_{d \in deps} cost_{exp.}(d)$$

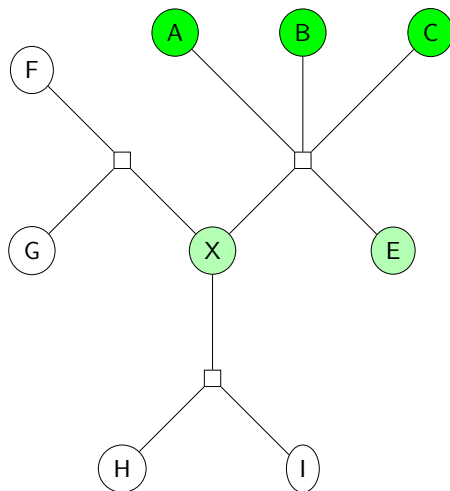
# Planning algorithm

Recursively materialize the inputs



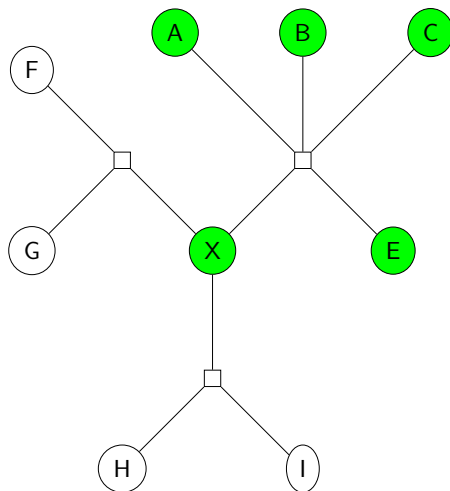
# Planning algorithm

Once the inputs are materialized, garbage collect to make room for the outputs



# Planning algorithm

Mark the outputs as materialized and register the operator for the plan



# Profit!





# Planning algorithm (recap)

- A network with reverse nodes
- Check depsets for materializable
- Chose an output set
- Halt by combining
  - ▶ Cost of operations so far.
  - ▶ Cost of historical costs given the materialized nodes.
  - ▶ Expected cost of input.
- Recurse on chosen inputs.
- Garbage collect to make space for the output set
- Mark outputs as materialized and register the operator as triggered.

# Antisthenis

## Dynamically scheduled incremental computation

Materializability and cost inference are numerical operations:

- Input is mostly the same between runs: **incremental**.
- **Order of computation** highly affects the performance (eg absorbing elements, min).
- Self referential computations may appear earlier than the absorbing element.

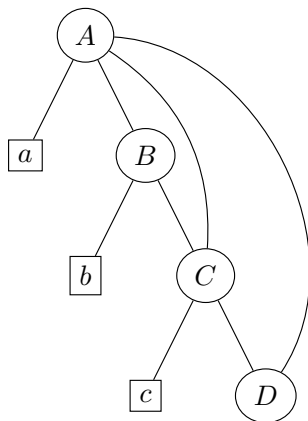
# Antisthenis: Expression graphs

$$A = a + B + C + D$$

$$B = C \times b$$

$$C = D + c$$

$$D = 0$$



## Antisthenis: Absorbing element

$$A = B \times C \times D$$

$$B = \sum_i i$$

$$C = 10 - 10$$

$$D = \sum_i i$$

# Antisthenis: Early stopping – recursive expressions

While expressions may be self-referential, we can sometimes still evaluate them.

$$A = \min(B, C, D)$$

$$B = b_1 + b_2 \cdot D$$

$$C = c_1 + c_2 \cdot A$$

$$D = d_1 + d_2 \cdot B$$

---

$$b_1 = b_2 = d_1 = d_2 = 1$$

$$c_1 = 3$$

$$c_2 = 0$$

# Kinds of operations: Materializability

$$matable(n) := \bigvee_{depset \in depsets(n)} \bigwedge_{dep \in depset} mat(dep) \vee matable(dep)$$

# Kinds of operations: Materializability

$$matable(n) := \bigvee_{depset \in depsets(n)} \bigwedge_{dep \in depset} mat(dep) \vee matable(dep)$$

- Recursive – normally we would maintain a visited set.
- Incremental evaluation is inhibited.
- Both  $\wedge$  and  $\vee$  have absorbing elements.

# Kinds of operations: Estimated cost

$$cost(n) := \min_{depset \in depsets(n)} \left[ cost_{op}(operator(depset)) + \sum_{dep \in depset} \neg mat(n) \cdot cost(dep) \right]$$

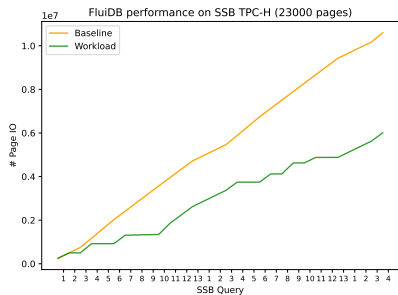
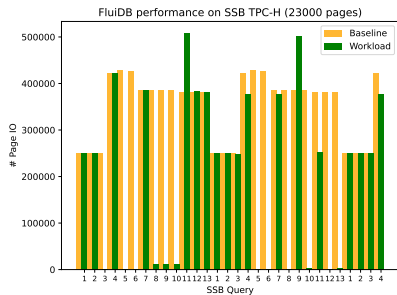


# Kinds of operations: Estimated cost

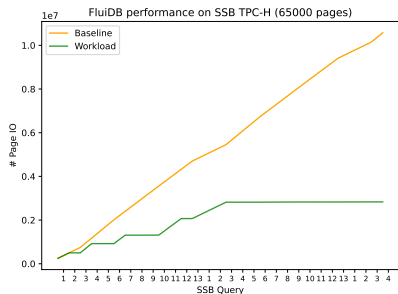
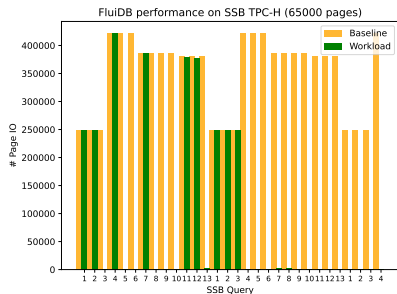
$$cost(n) := \min_{depset \in depsets(n)} \left[ cost_{op}(operator(depset)) + \sum_{dep \in depset} \neg mat(n) \cdot cost(dep) \right]$$

- Recursive – Incremental evaluation is inhibited.
- $\min$  can be exploited for early stopping

# Evaluation: 23K pages budget

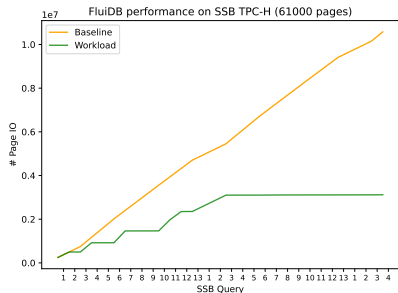
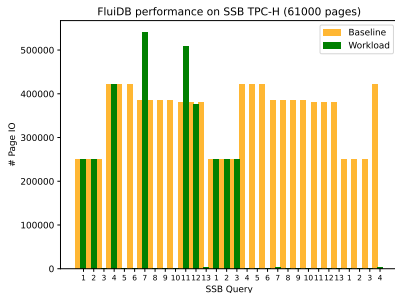


# Evaluation: 65K pages budget

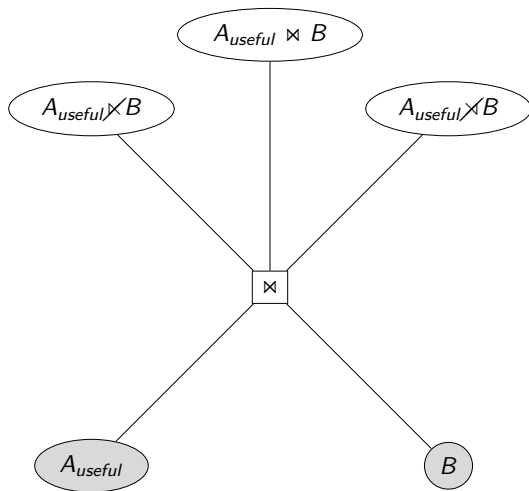


# Evaluation: But ... 61K pages budget

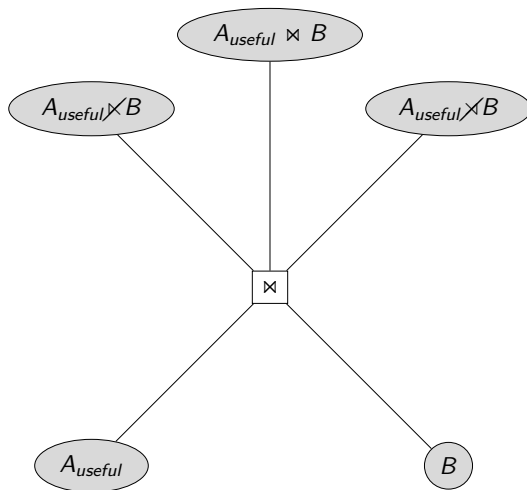
lineorder is deleted at 6 because all join outputs were materialized



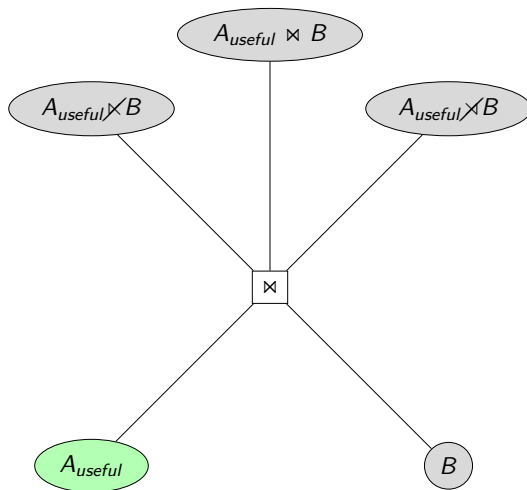
# Plenty of memory



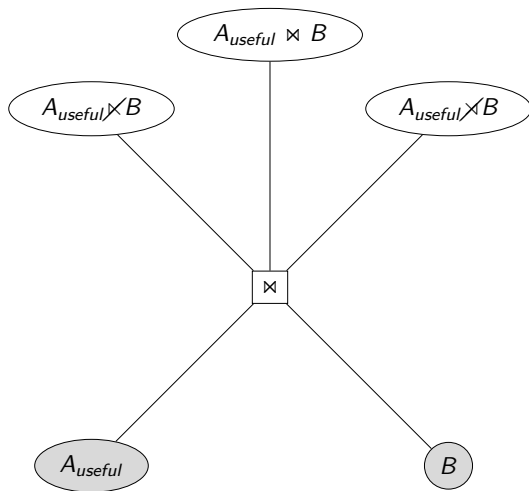
# Plenty of memory



# Plenty of memory

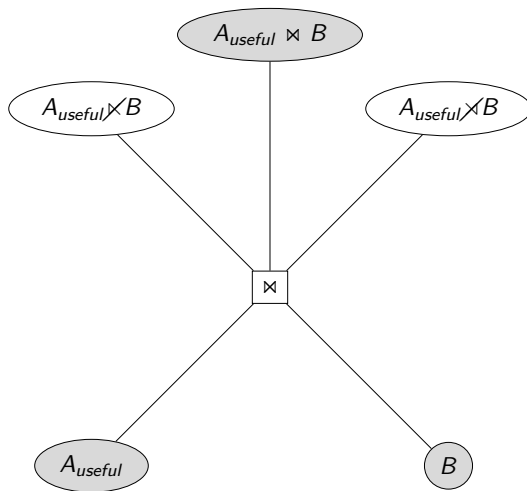


## Being on a budget

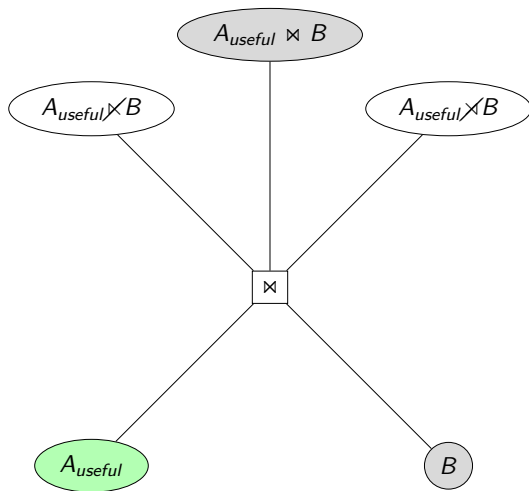




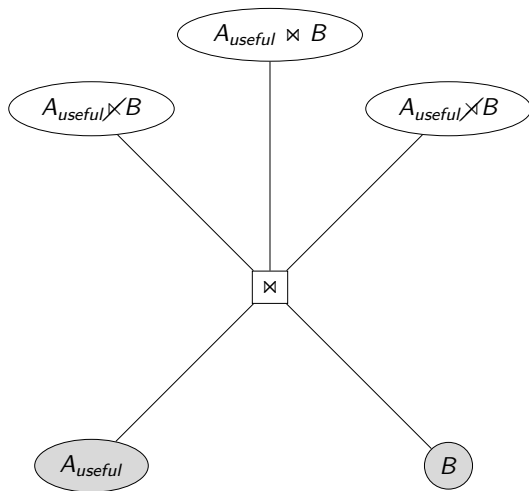
## Being on a budget



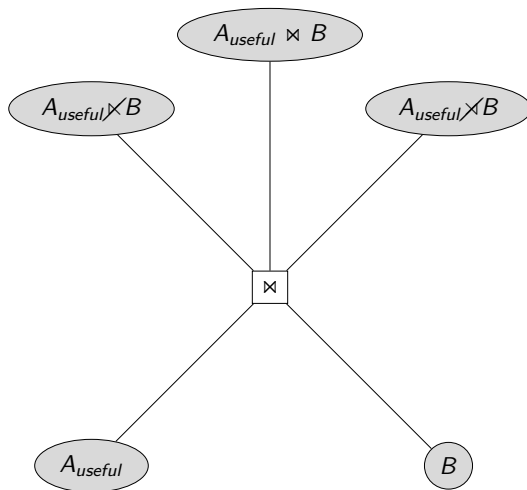
## Being on a budget



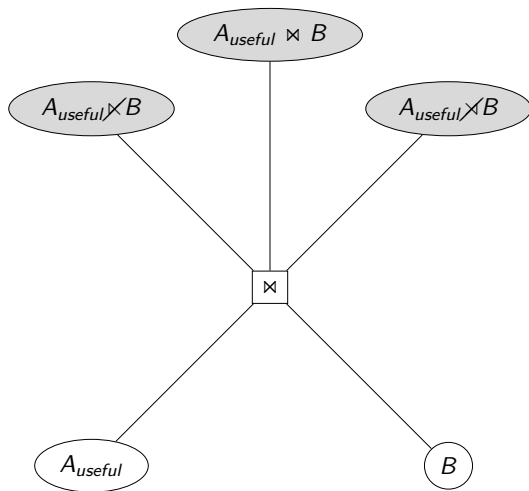
# Having just enough rope



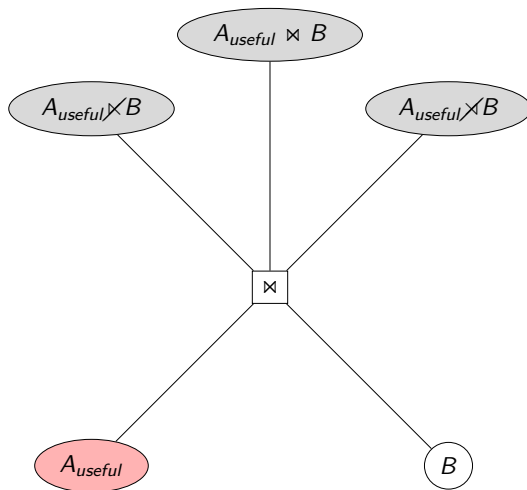
# Having just enough rope



# Having just enough rope



# Having just enough rope



# Conclusions

- FluidB can efficiently use memory budget to store useful intermediate results.
- FluidB is virtually always better than the naive case.
- FluidB can incrementally adapt to the workload.