FluiDB: Adaptive storage layout using reversible relational operators

<Subtitle>

Christos Perivolaropoulos

University of Edinburgh

January 1, 1980

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.
- 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.

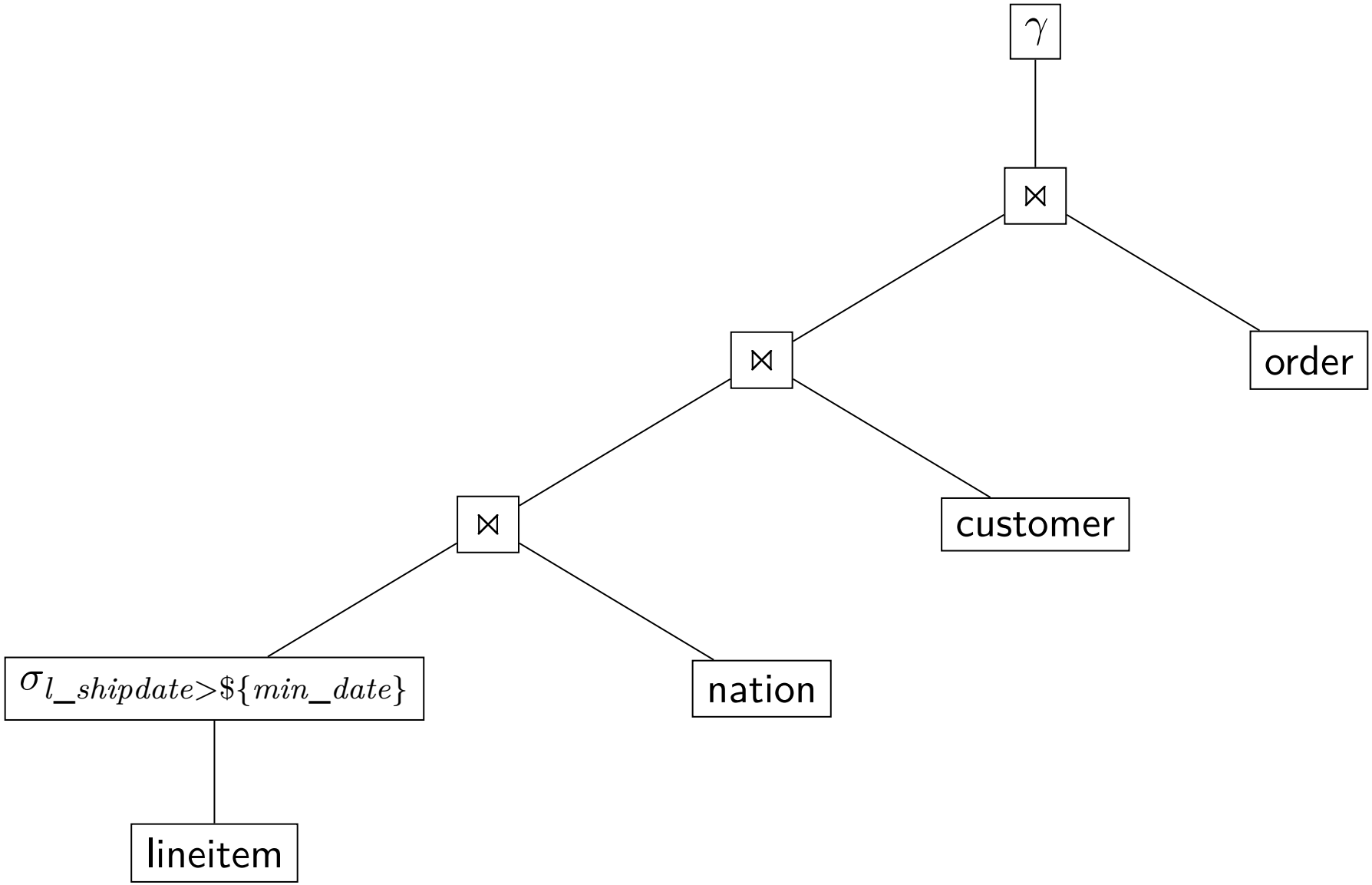
Example 1: Workload based on template query

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

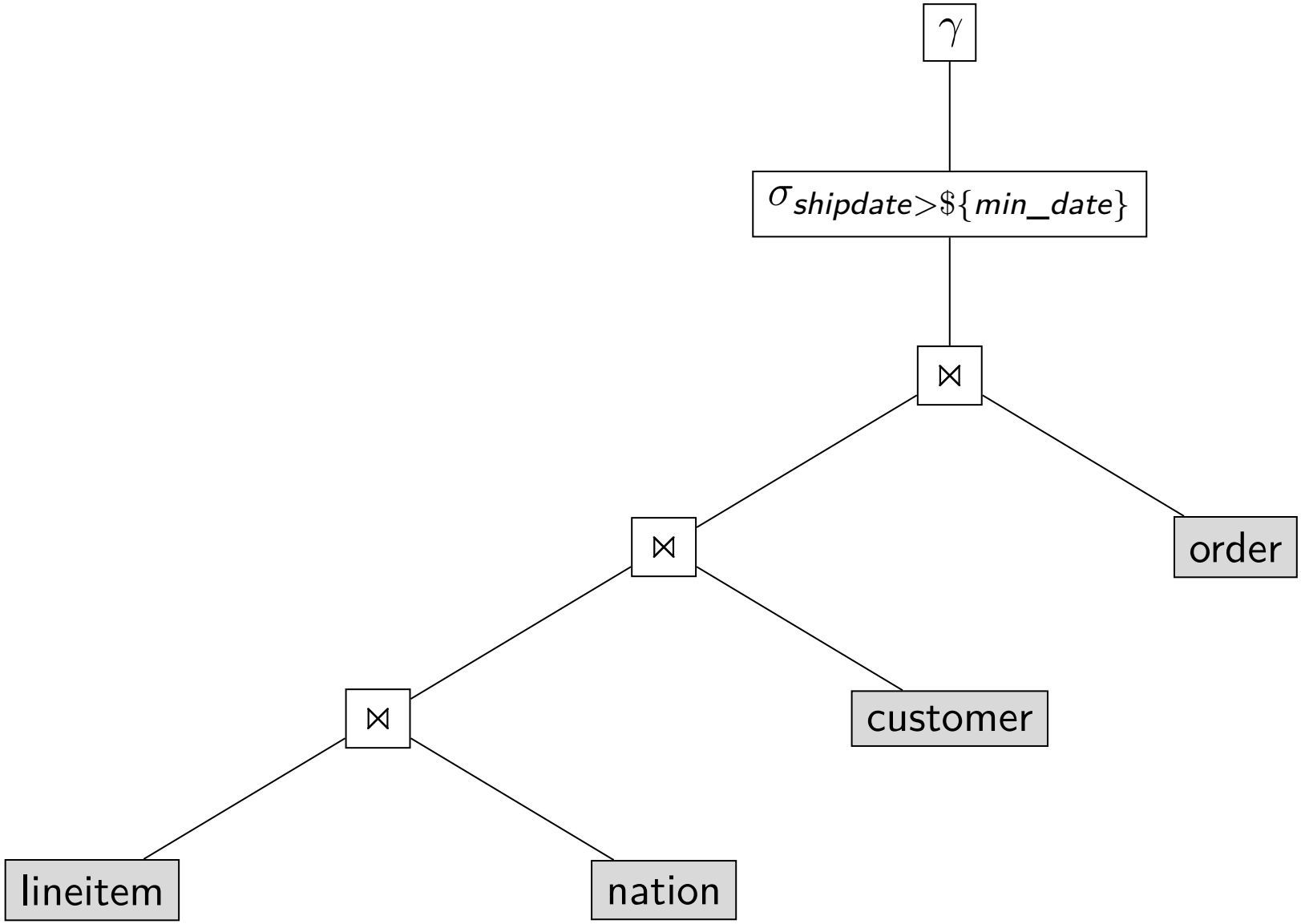
```
select      n_name, avg(l_discount)
from        lineitem, customer, nation
where       l_orderkey = o_orderkey
and         c_custkey = o_custkey
and         c_nationkey = n_nationkey
and         l_shipdate > ${min_date}
group by    n_name
```

Example 1: Traditional single-query plan

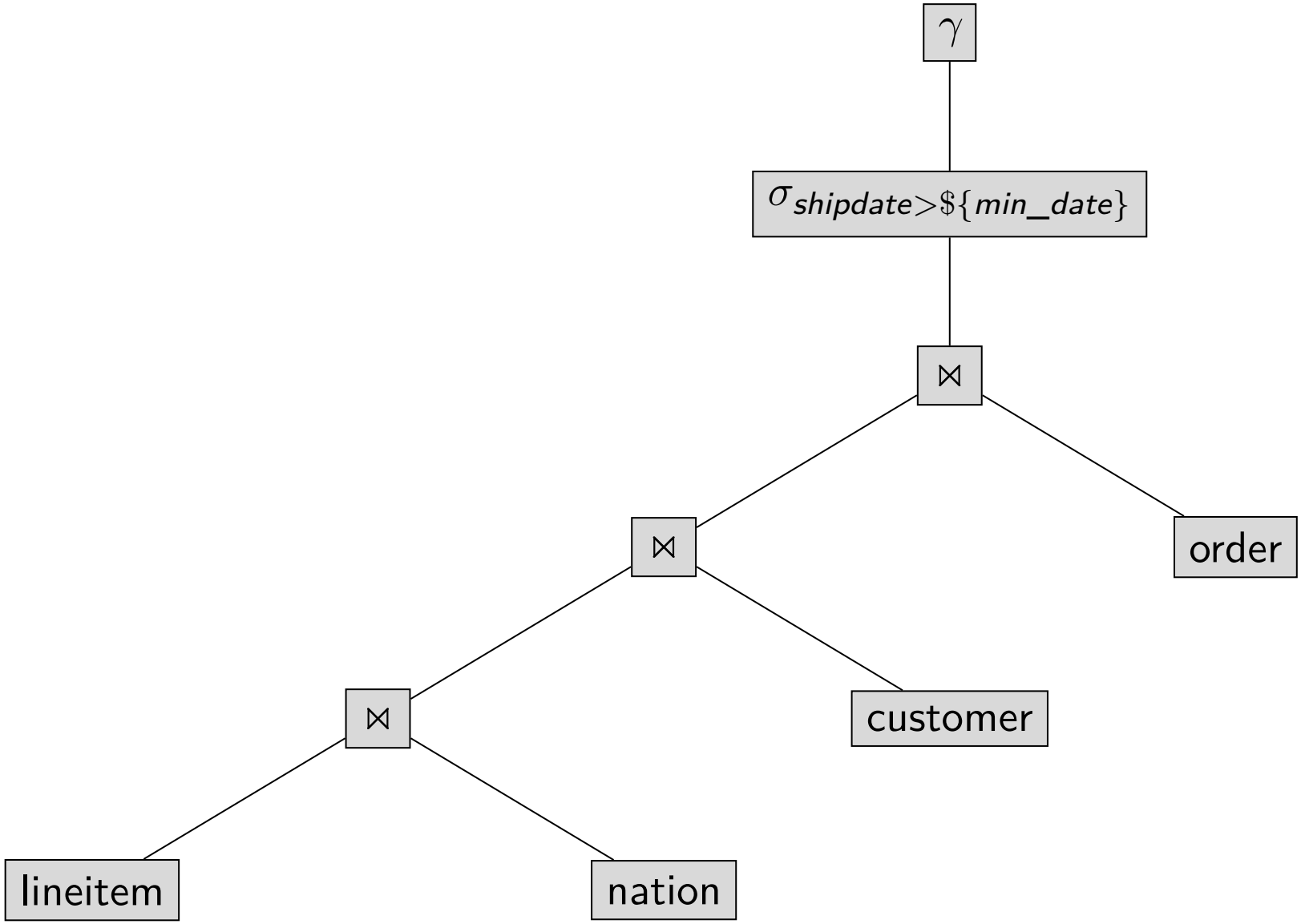
Selection push down



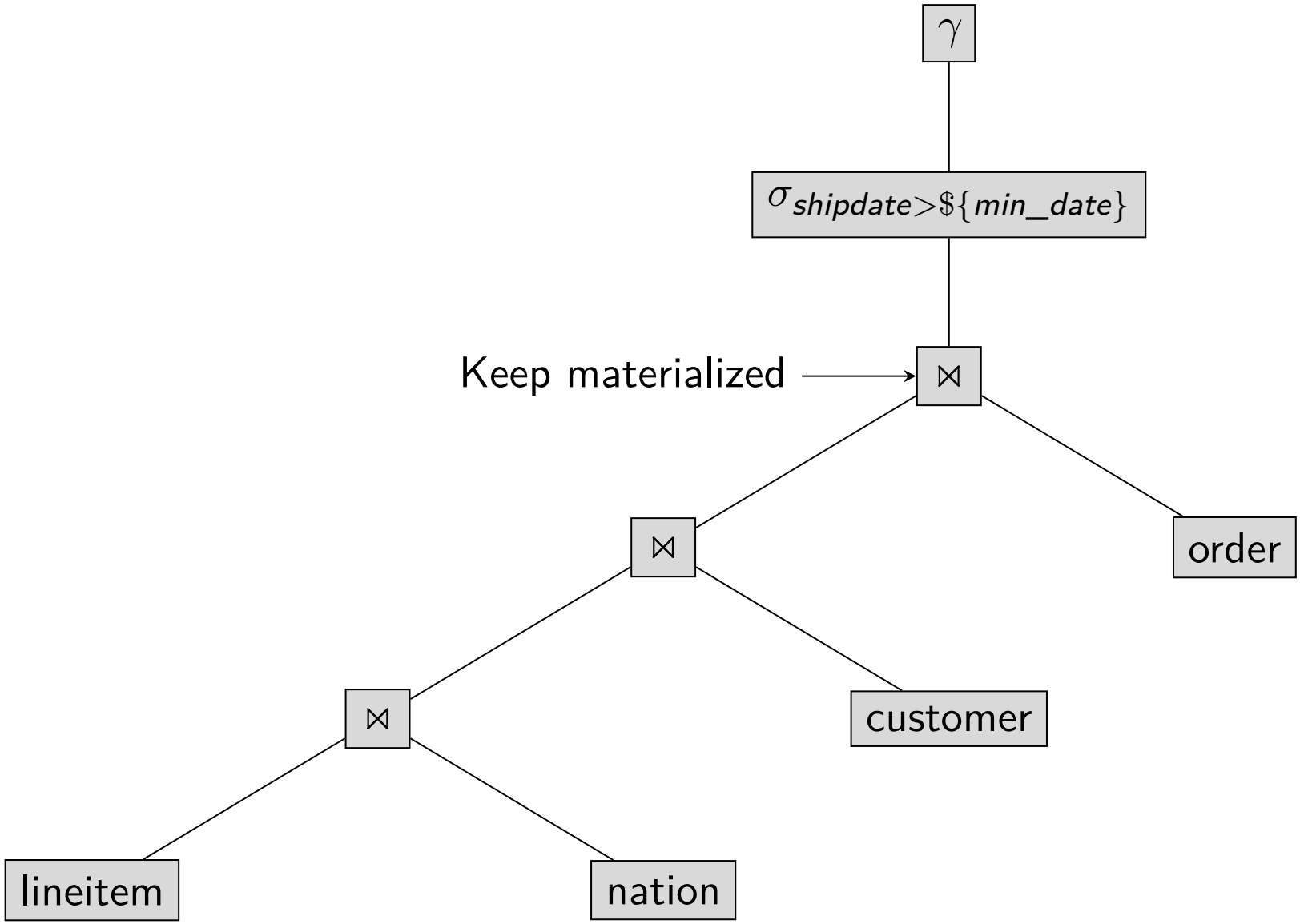
Example 1: Adapt to workload



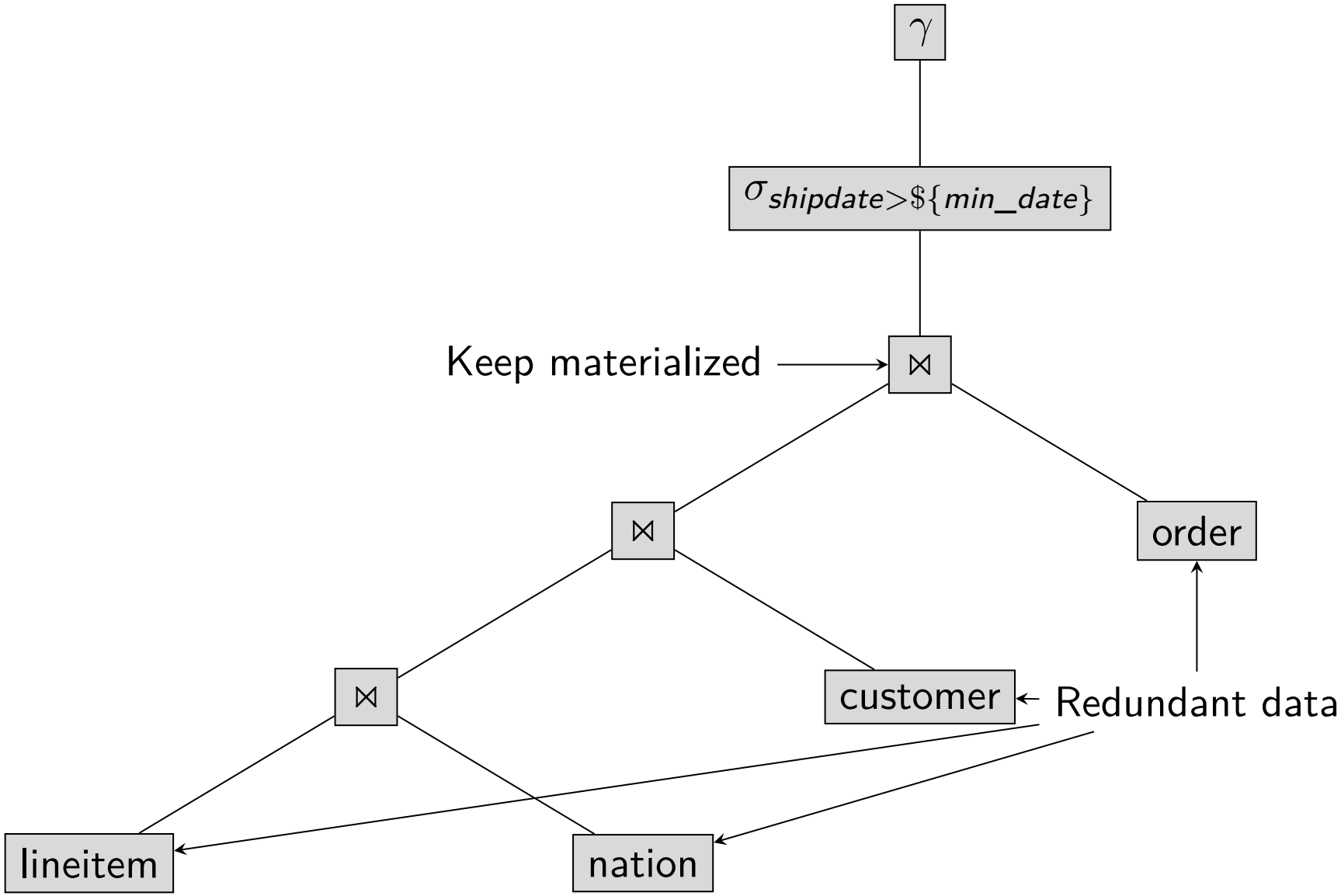
Example 1: Adapt to workload



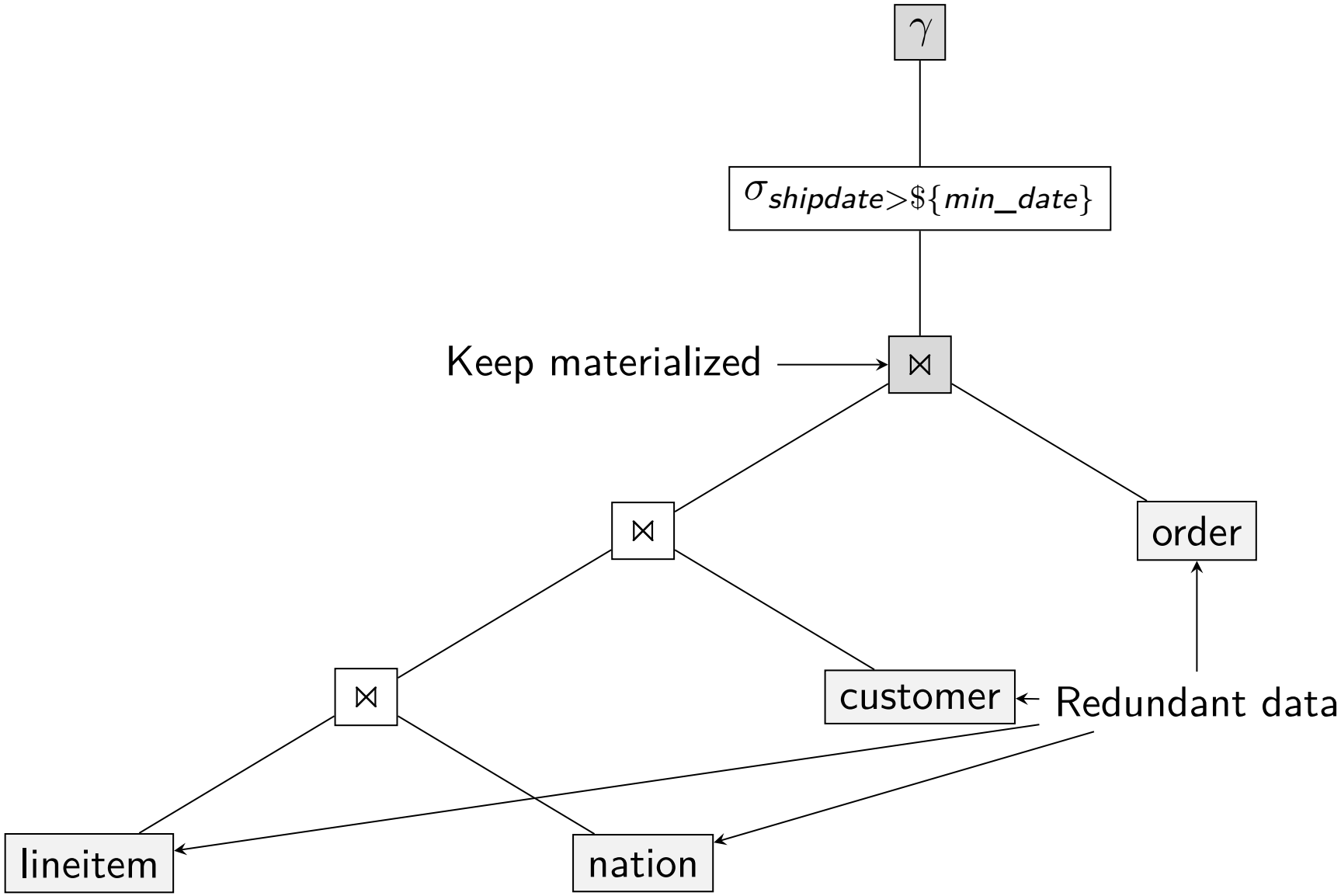
Example 1: Adapt to workload



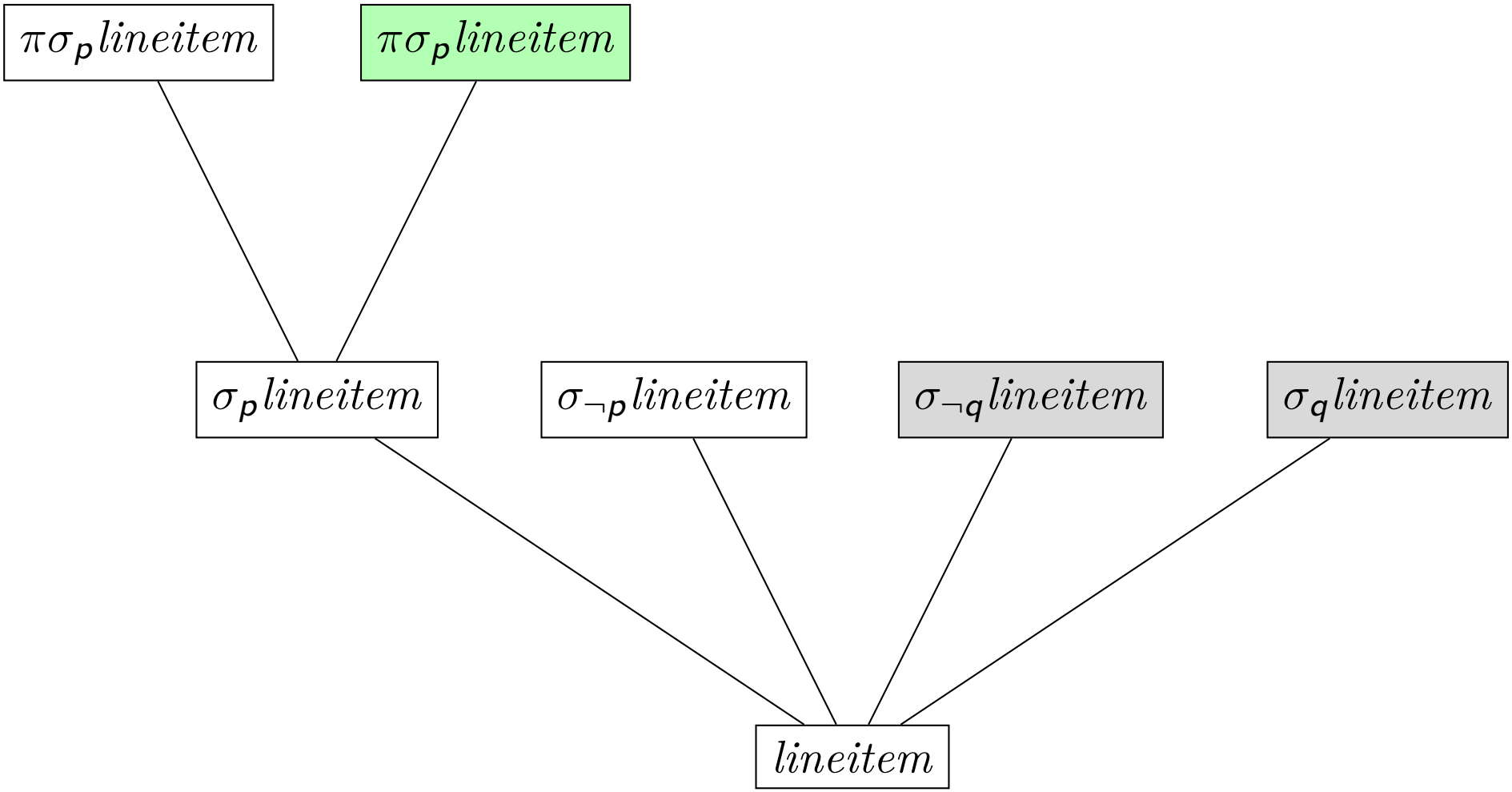
Example 1: Adapt to workload



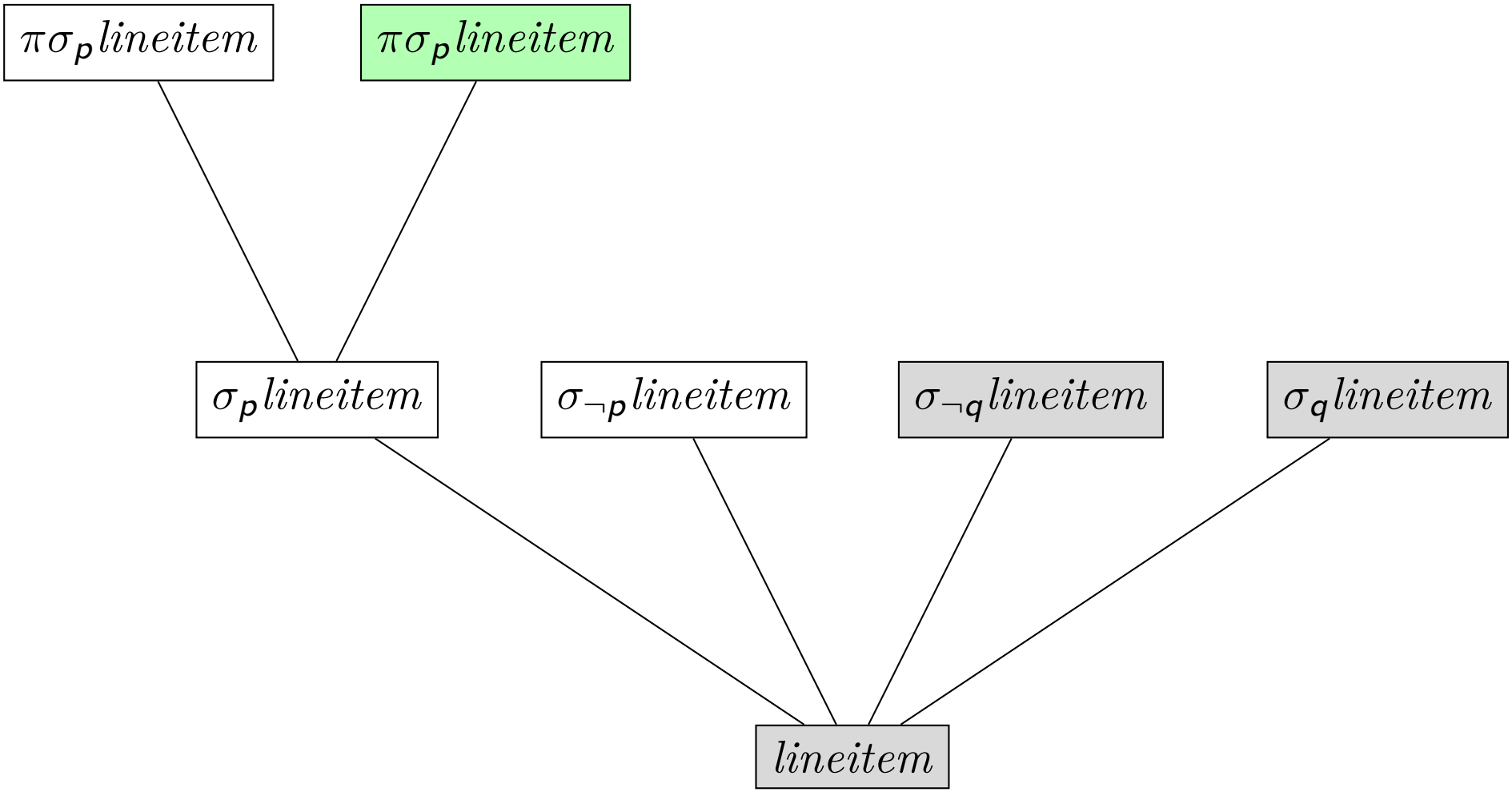
Example 1: Adapt to workload



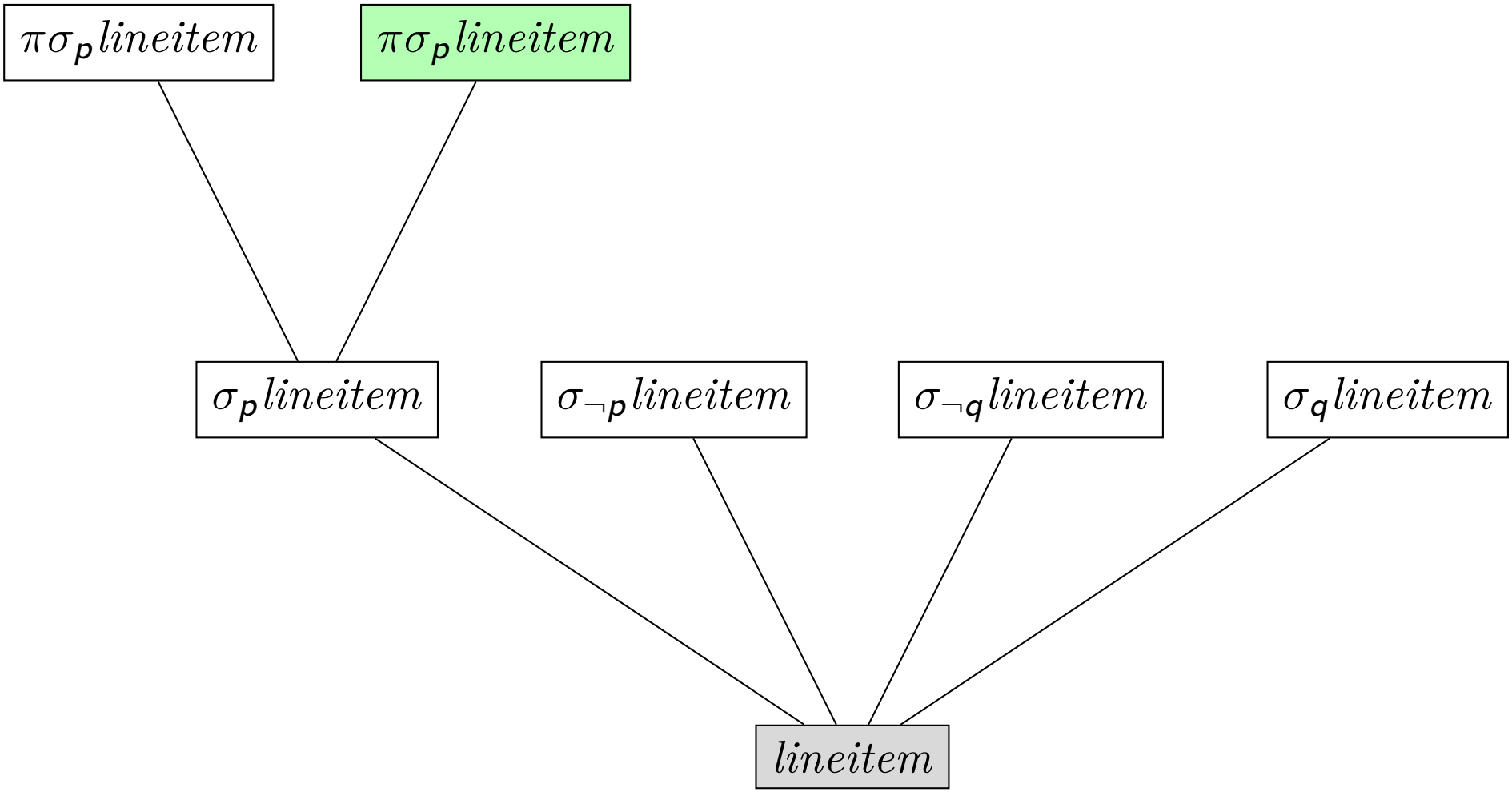
Example 2: Plan with missing primary data



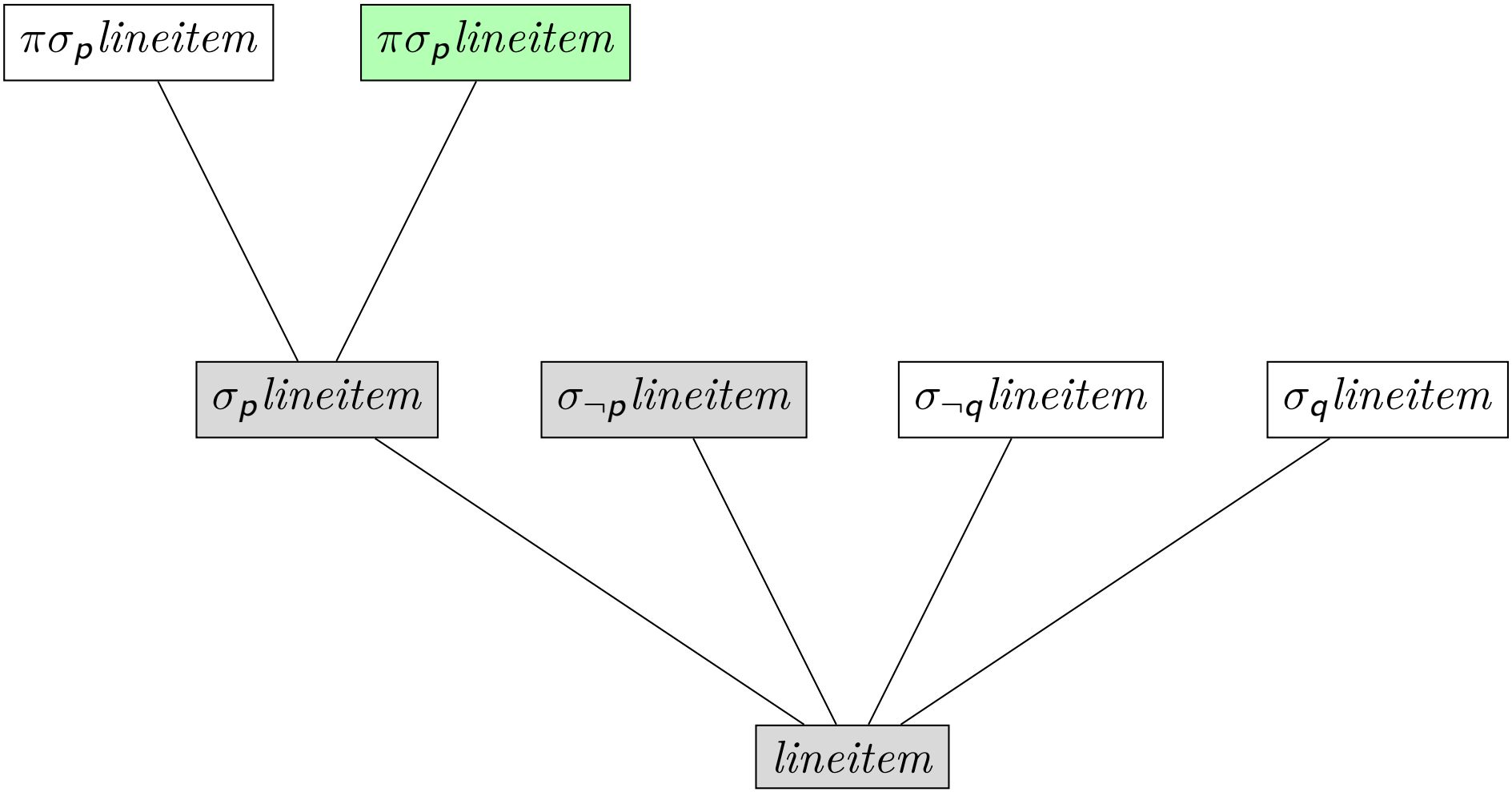
Example 2: Plan with missing primary data



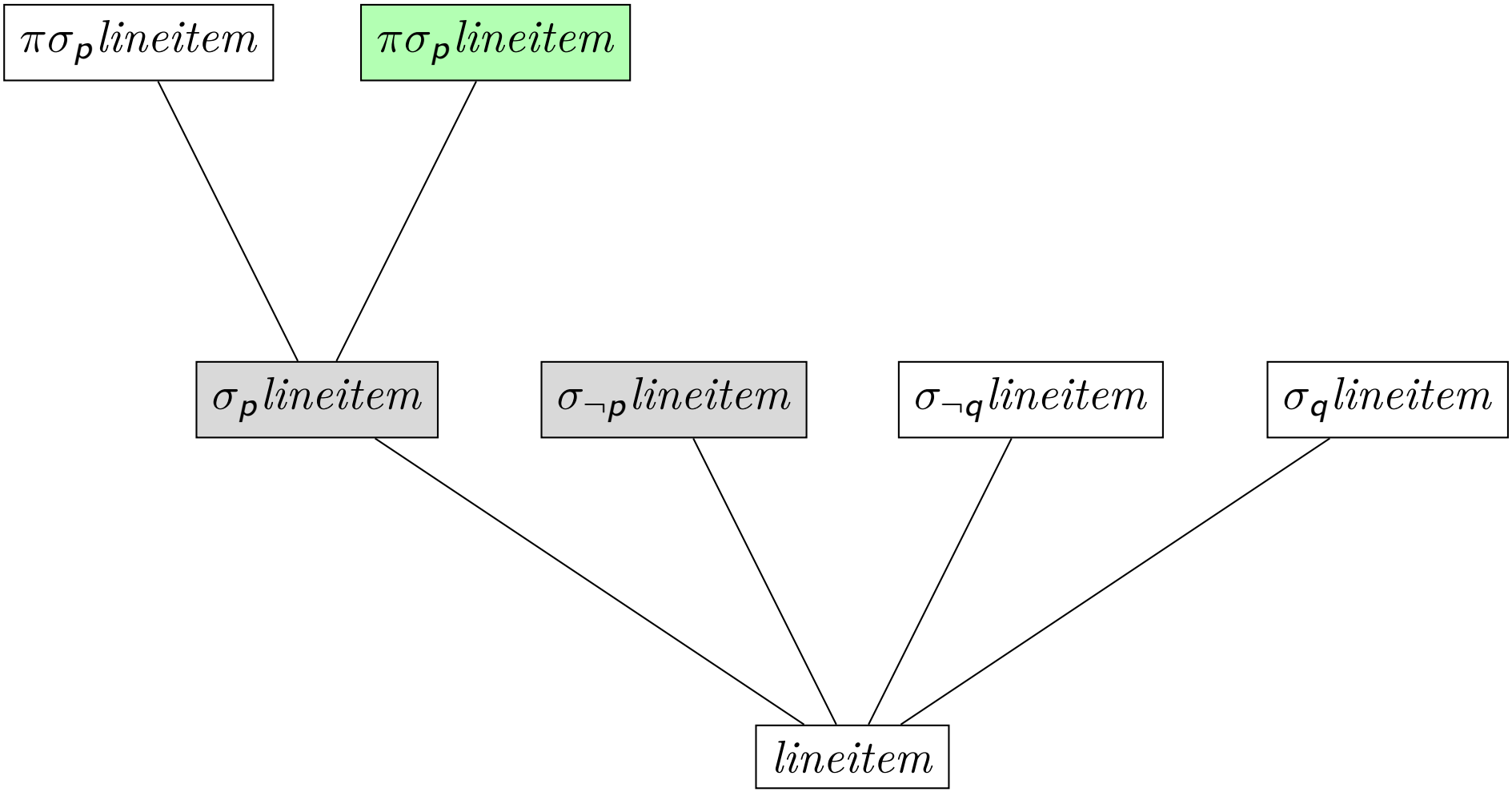
Example 2: Plan with missing primary data



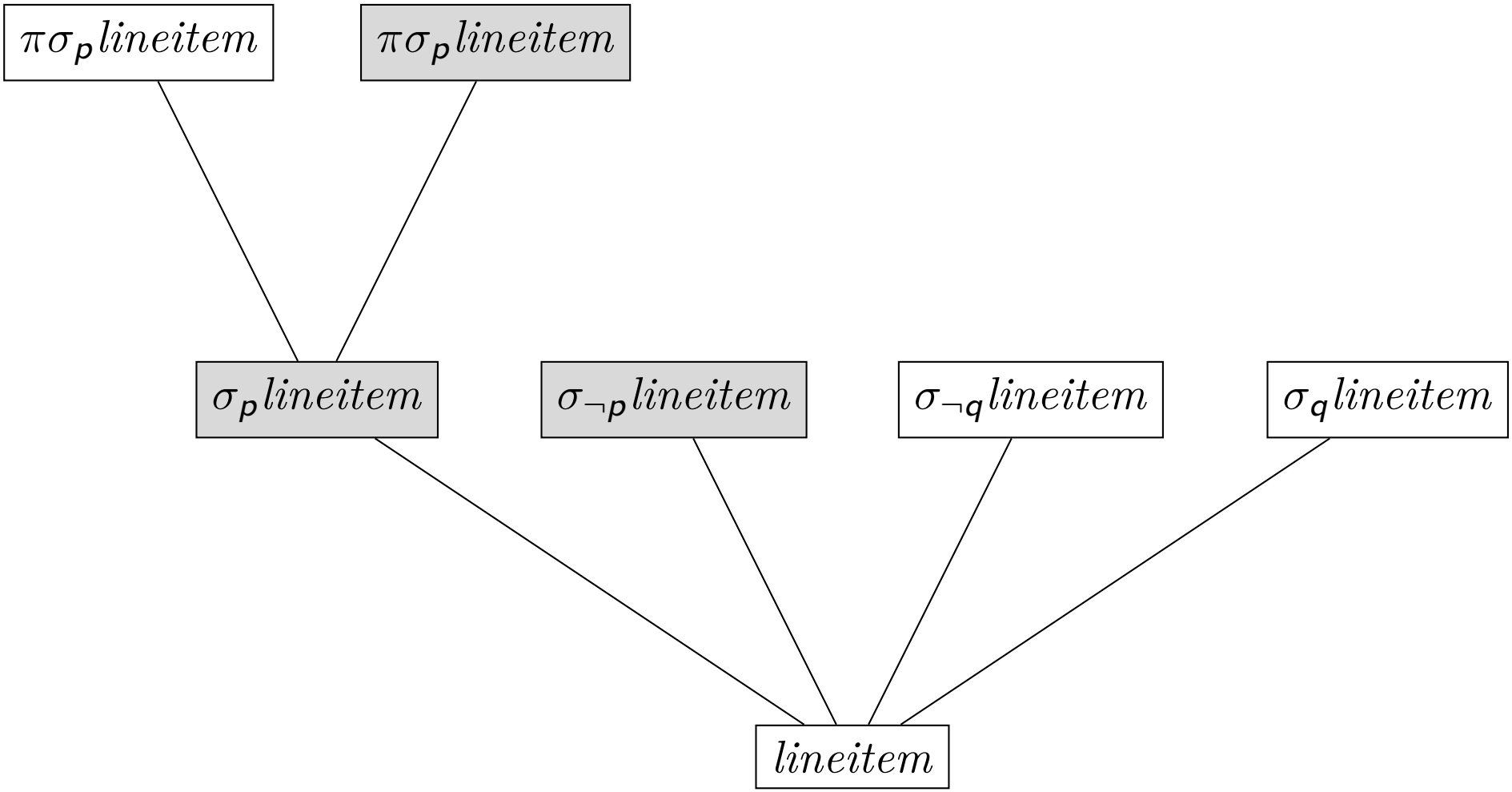
Example 2: Plan with missing primary data



Example 2: Plan with missing primary data



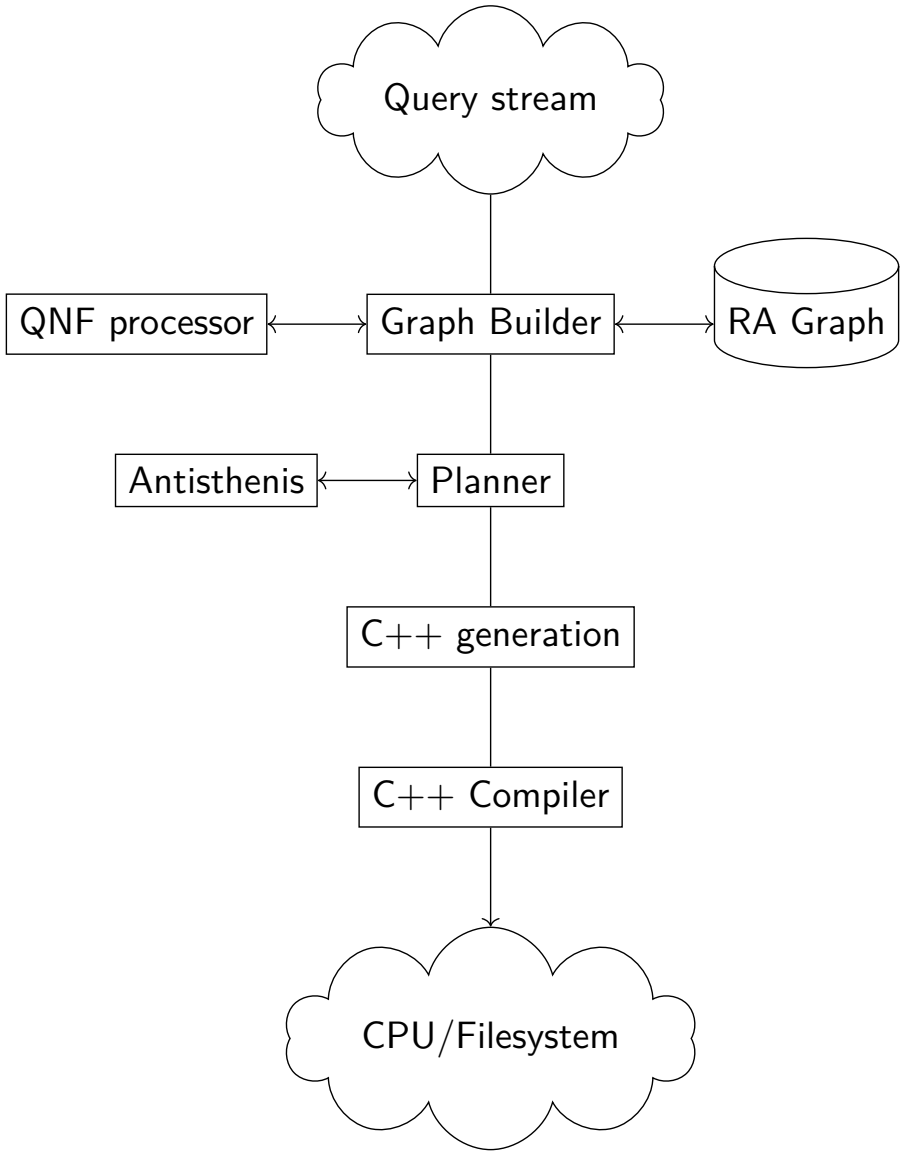
Example 2: Plan with missing primary data



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
- Code generation system.

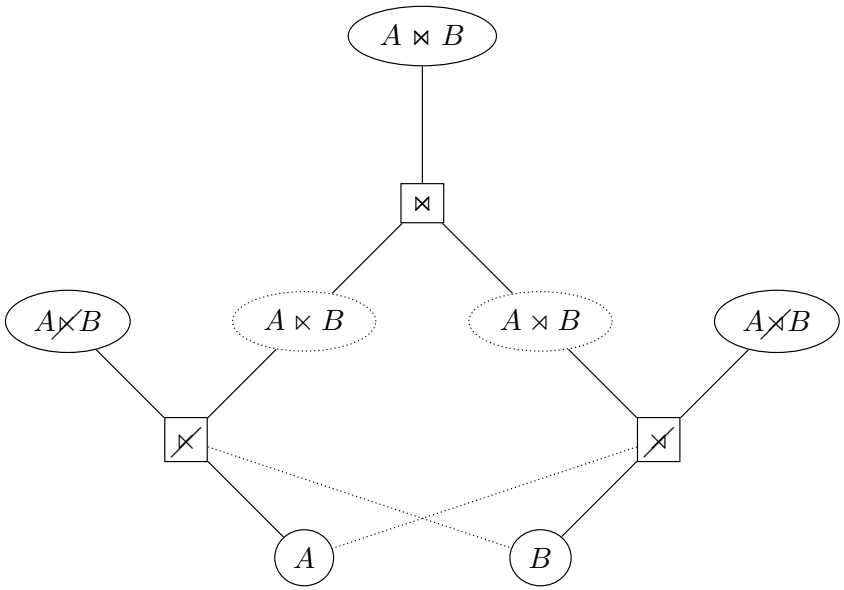
Architecture

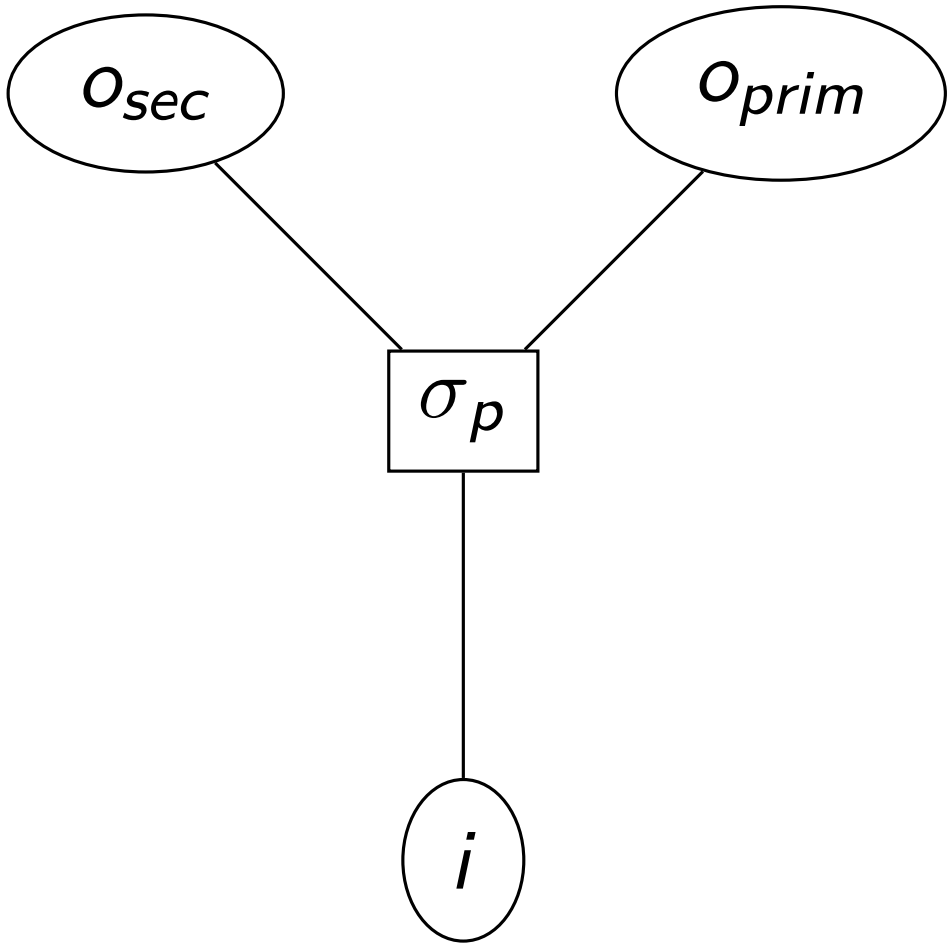


Logical planning

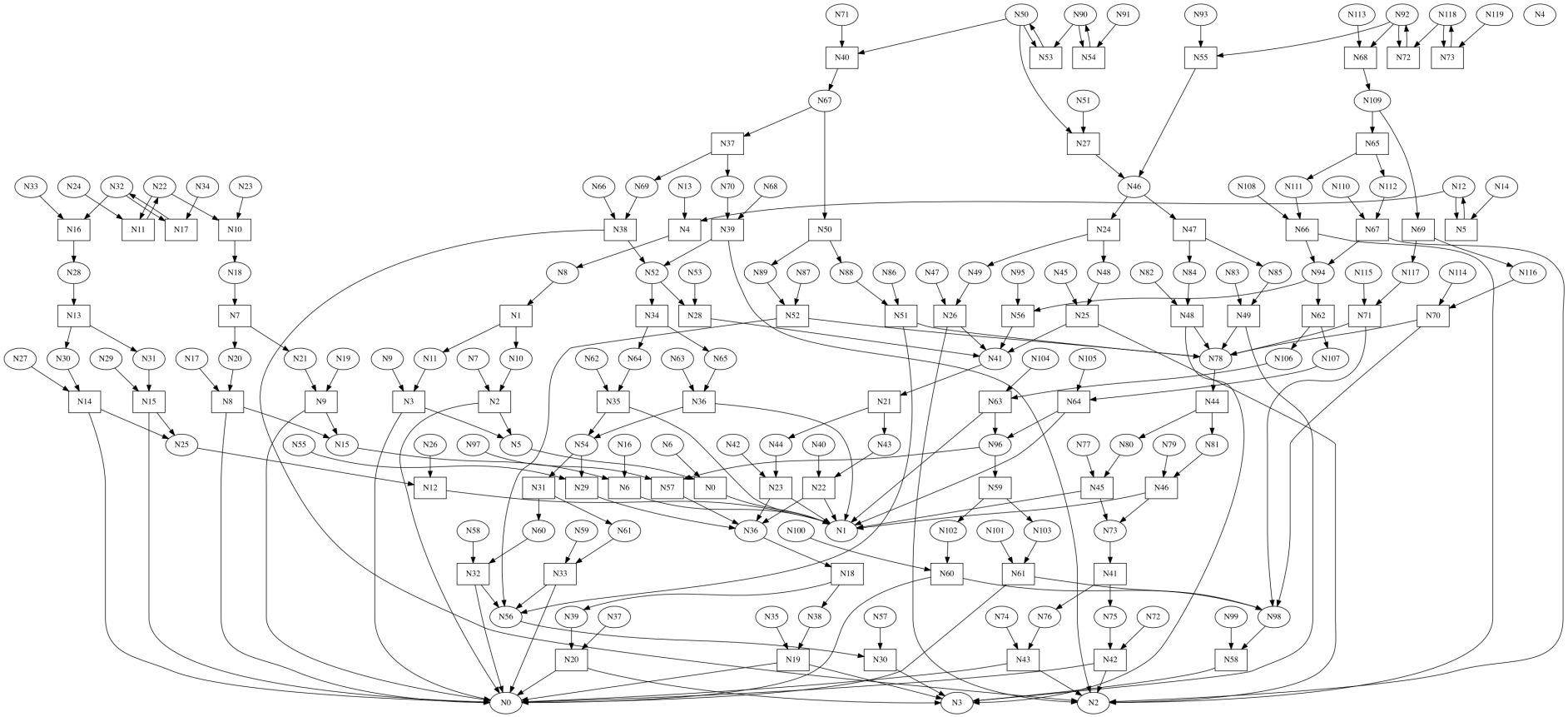
- Bipartite query graph – RA operations/relations unified for all queries
- Join ordering enumeration
- QNF – $\pi\sigma(Q_1 \times Q_2 \times \dots)$ or $\gamma\sigma(Q_1 \times Q_2 \times \dots)$
- Relation shape propagation – cardinality, columns/types, unique subtuples

Reversible operators





Reversible operations



List as a monad = backtracking

```haskell
indexWords :: [(String,String)]
indexWords = do
  sentence ← return "the red and brown fox" <|> return "the black and
  ↪  blue bear"
  w ← words sentence -- for each word
  guard $ w `notElem` ["the", "and"] -- skip the boring words
  return (w,sentence)
```

List as a monad = backtracking

```
indexWords :: [(String,String)]
indexWords = do
  sentence ← return "the red and brown fox" <|> return "the black and
  ↪  blue bear"
  w ← words sentence -- for each word
  guard $ w `notElem` ["the", "and"] -- skip the boring words
  return (w,sentence)

> indexWords
[("red","the red and brown fox")
,("brown","the red and brown fox")
,("fox","the red and brown fox")
,("black","the black and blue bear")
,("blue","the black and blue bear")
,("bear","the black and blue bear")]
```

List based logic/backtracking (unfair)

```haskell
nonTerm :: [(Int,Int,Int)]
nonTerm = do
  -- >  (,) <$> [1..3] <*> [1..3]
  -- [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
  (a,b,c) ← (,,) <$> [0..] <*> [0..] <*> [..]
  guard $ a + b - c == 10
  return (a,b,c)
```

List based logic/backtracking (unfair)

```haskell
nonTerm :: [(Int,Int,Int)]
nonTerm = do
  -- >  (,) <$> [1..3] <*> [1..3]
  -- [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
  (a,b,c) ← (,,) <$> [0..] <*> [0..] <*> [..]
  guard $ a + b - c == 10
  return (a,b,c)

> take 3 nonTerm
```

⊥

List based logic/backtracking (fair)

```haskell
term :: [(Int,Int,Int)]
term = do
  -- (>*<) :: [a → b] → [a] → [b]
  -- > (,) <$> [1..3] >*< [1..3]
  -- [(1,1),(2,1),(1,2),(2,2),(3,1),(3,2),(1,3),(2,3),(3,3)]
  (a,b,c) ← (,,) <$> [0..] >*< [0..] >*< [0..]
  guard $ a + b - c == 10
  return (a,b,c)
```

List based logic/backtracking (fair)

```haskell
term :: [(Int,Int,Int)]
term = do
  -- (>*<) :: [a → b] → [a] → [b]
  -- > (,) <$> [1..3] >*< [1..3]
  -- [(1,1),(2,1),(1,2),(2,2),(3,1),(3,2),(1,3),(2,3),(3,3)]
  (a,b,c) ← (,,) <$> [0..] >*< [0..] >*< [0..]
  guard $ a + b - c == 10
  return (a,b,c)

  > take 5 term
  [(5,5,0),(6,4,0),(6,5,1),(4,6,0),(5,6,1)]
```

Physical planning

`HCntT` logic monad

Logic framework for "fair" traversal of the plan search space. Intricudes:

- `a <//> b`: Try the rest of the computation with `a` and if it fails try `b`.
- `once c`: try the continuation with values from `c` until one works and stick with that one.
- `halt n`: yield to a scheduler and assigne priority `n` to the continuation.

The GC

```
gc reqSize = do
  -- Try the current epoch and if that fails retry with a new epoch.
  return () <//> newEpoch
  -- find the deletable n-nodes and sort them by size
  deleteables ← sortOnM getNodeSize =≪ filterM isDeletable =≪
  ↪  getAllNodes
  -- Try deleting each n-node and stop deleting when amassing enough
  -- free pages.
  forM_ deletables $ \n → do
    freePgs ← getFreePages
    when (freePgs < reqSize) $ tryDelete n <//> markAsConcrete n
```

Physical planning

Business logic

```
materialize n = unless (materialized n) $ do
  op ← inputOps n
  outputs ← possibleOutputs n op
  let inputs = inputsOf op
  -- Assuming we materialized the output, what is the cost of the
  -- outputs
  once (gc outputs)
  histCost ← withMaterialized outputs $ historicalCosts
  -- Stop and schedule this branch according to its cost
  halt (cost op + histCost + anticipatedCost inputs)
  -- Recursively materialize the input relations
  mapM materialize inputs
  registerPlan op
  mapM (setState Materialized) output
```

Antisthenis

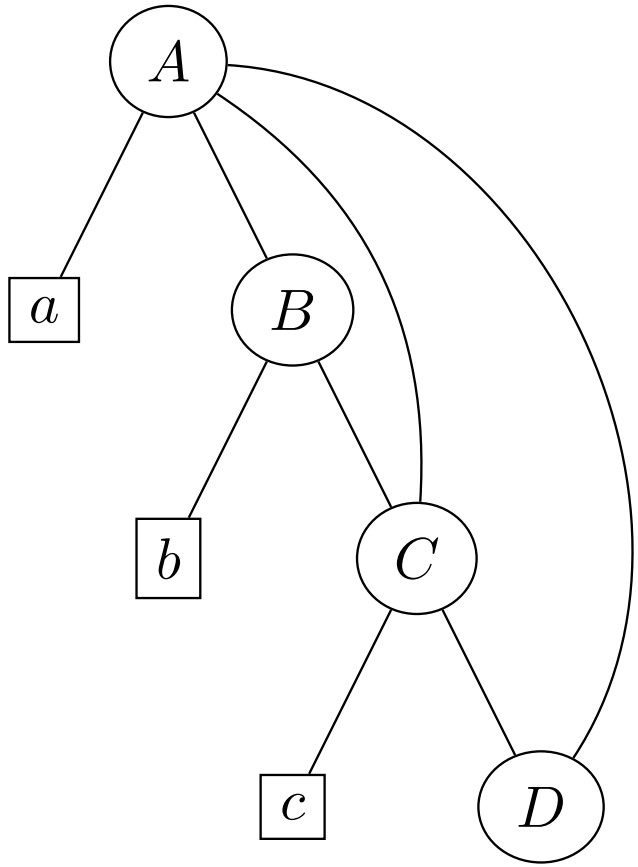
Dynamically scheduled incremental computation

Materializablility and cost inference are numerical operations:

- Input is mostly the same between runs: **incremental**.
- **Order of computation** highly affects the performance (eg absorbig elements, min).
- Self referrential 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.

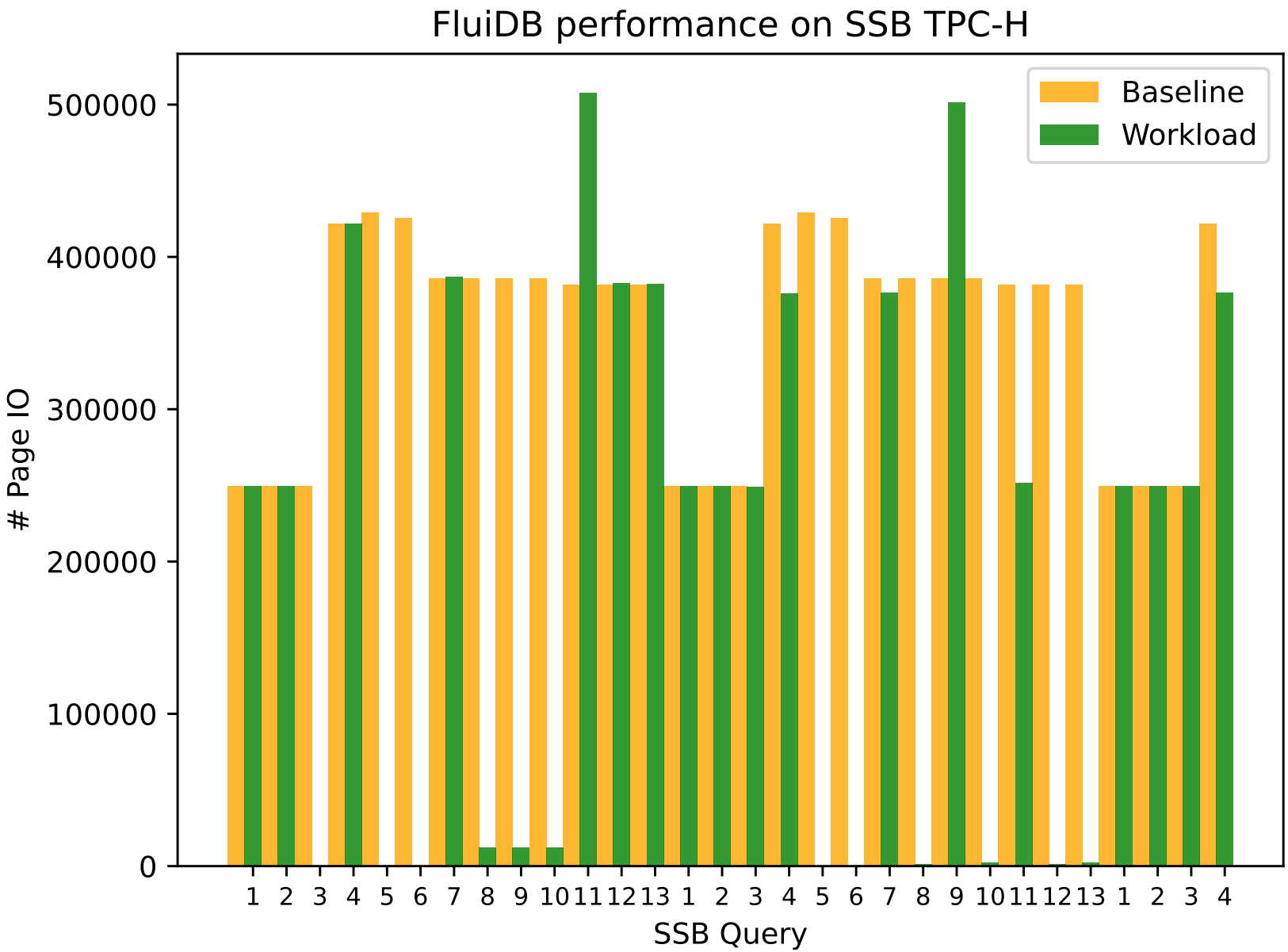
$$A = min(B, C, D)$$
$$B = b_1 + b_2 \cdot D$$
$$C = c_1 + c_2 \cdot A$$
$$D = d_1 + d_2 \cdot B$$

$$b_1 = b_2 = d_1 = d_2 = 1$$
$$c_1 = 3$$
$$c_2 = 0$$

Data layout

Code generation

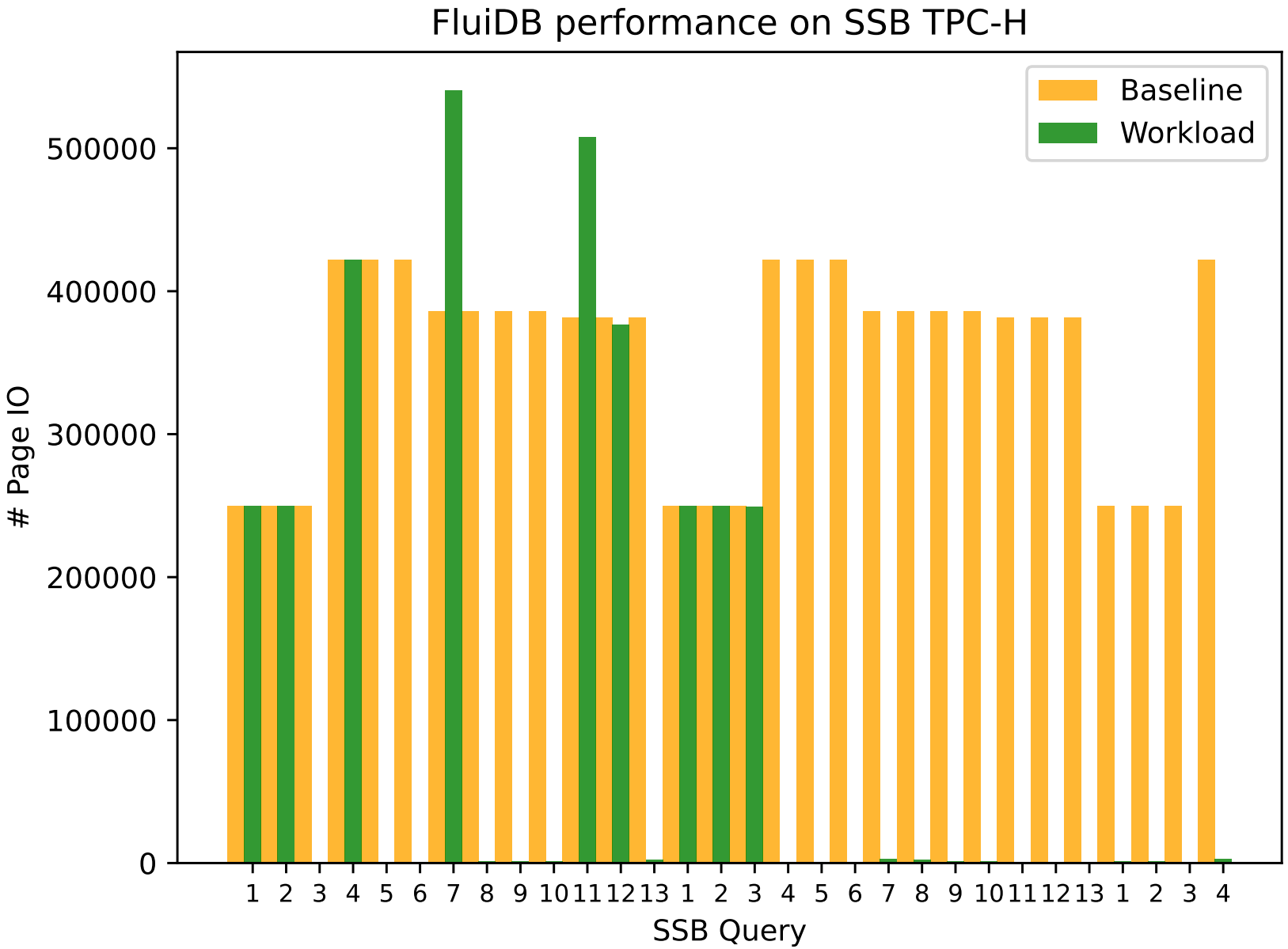
Evaluation: 23K pages budget



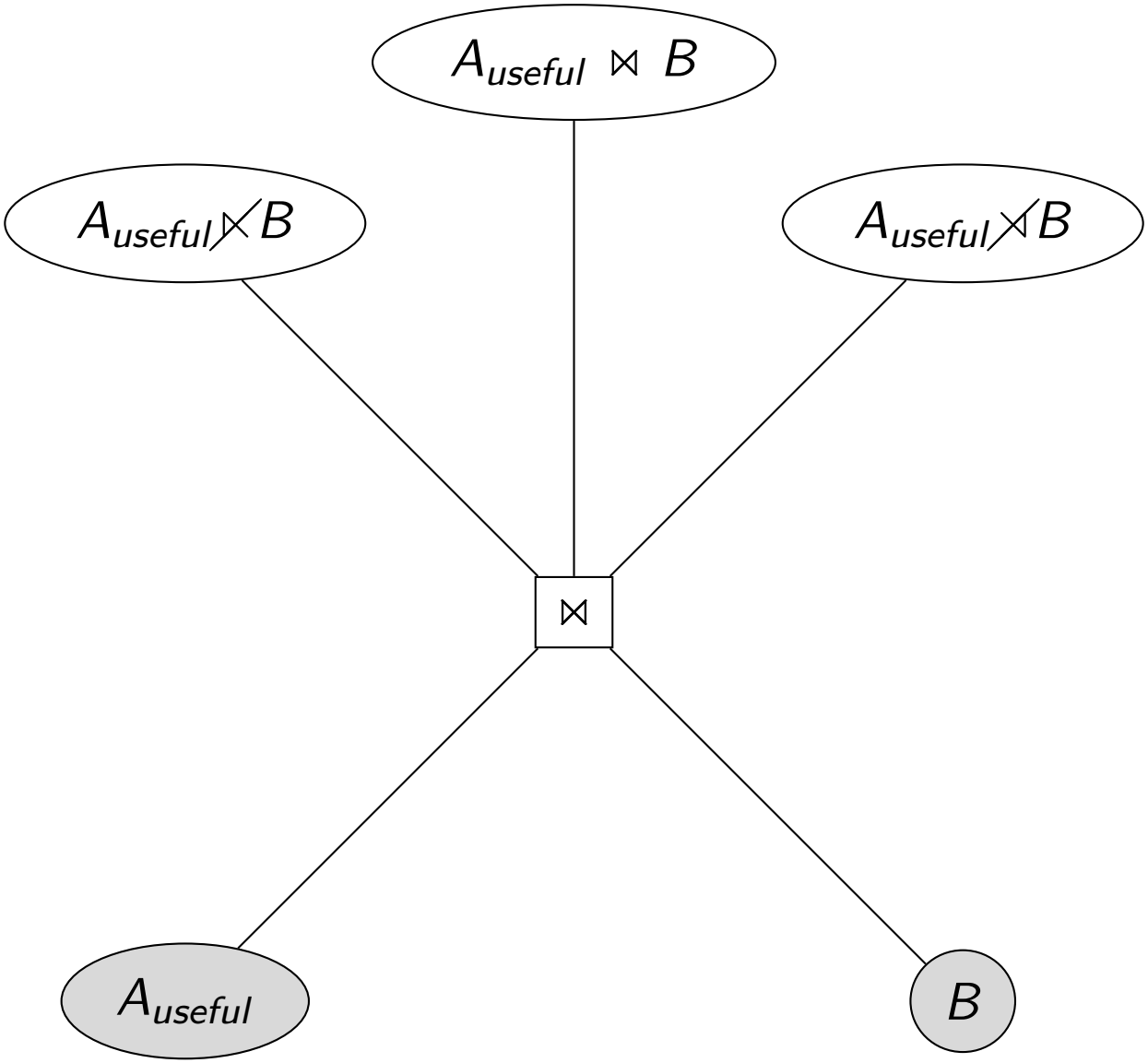
Evaluation: 65K pages budget

Evaluation: But ... 61K pages budget

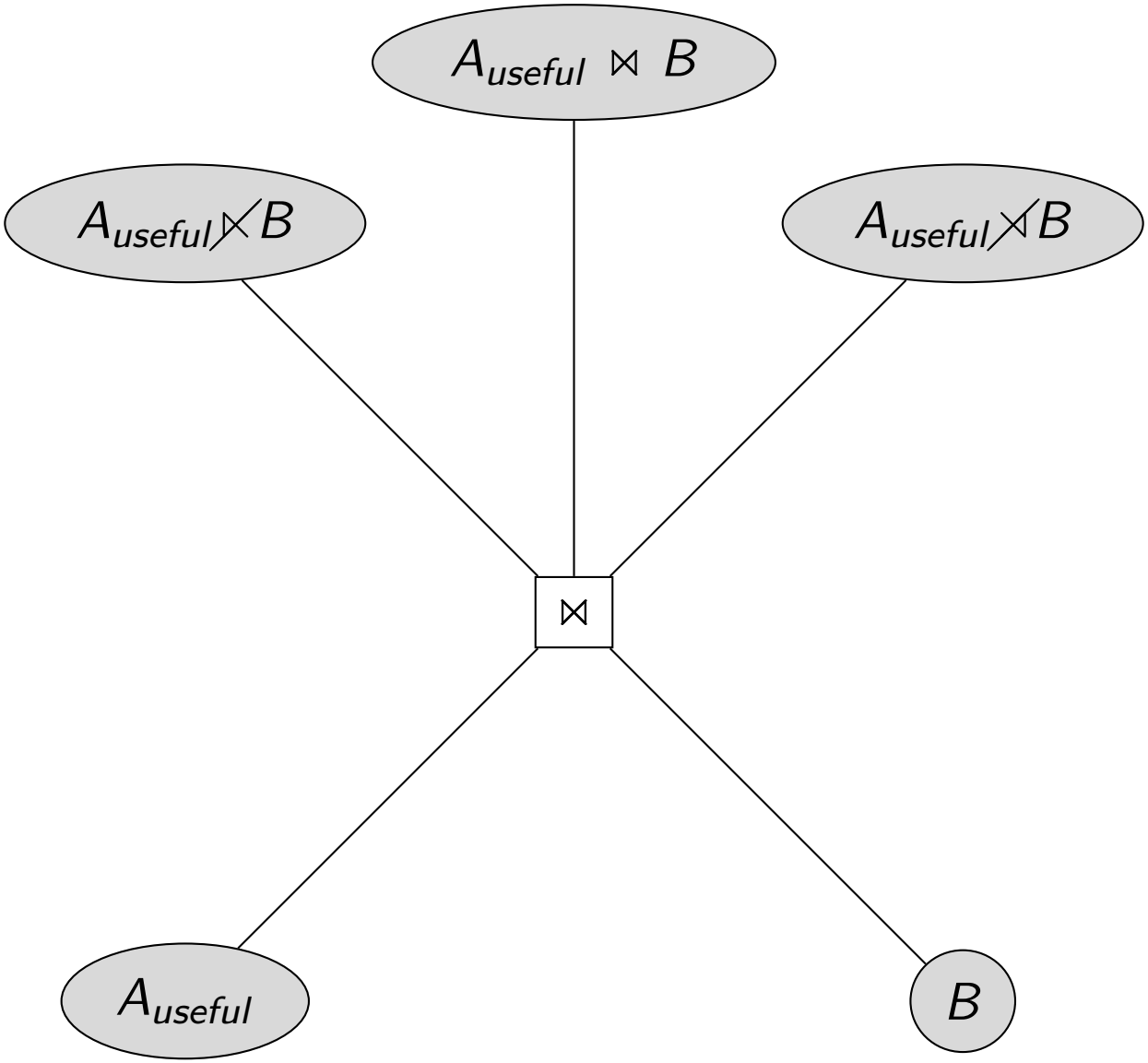
lineorder is deleted at 6 because all join outputs were materialized



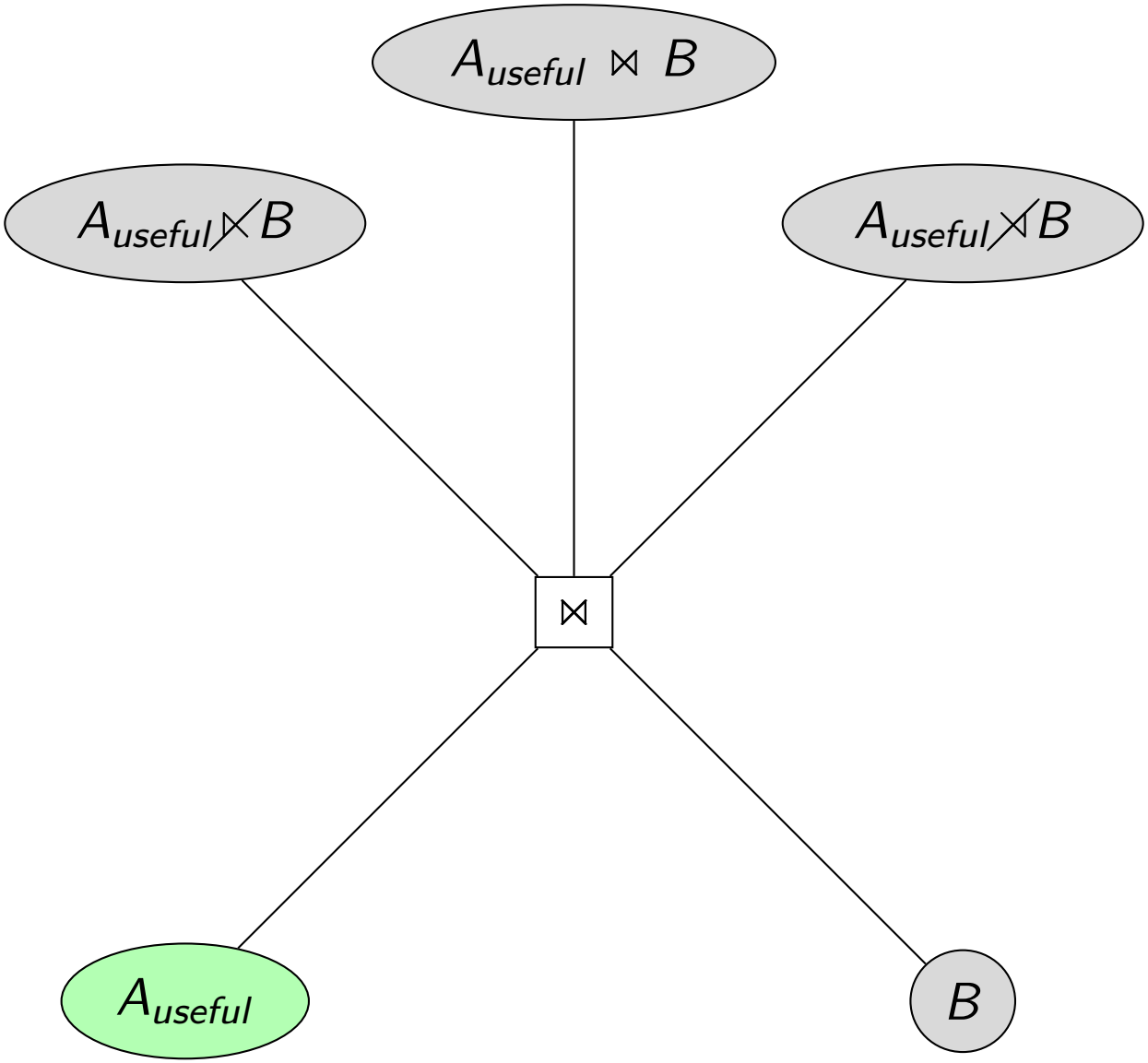
Plenty of memory



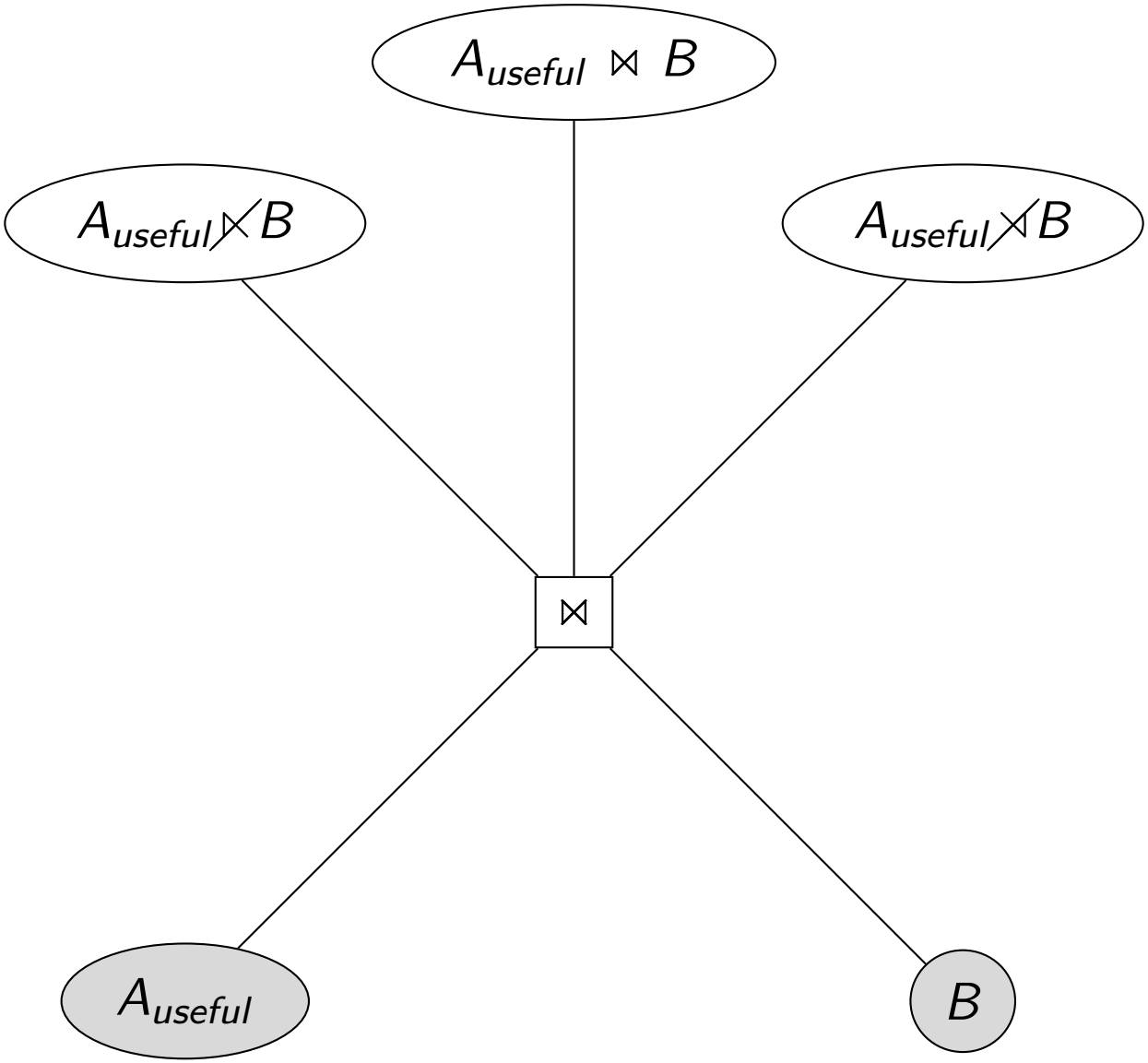
Plenty of memory



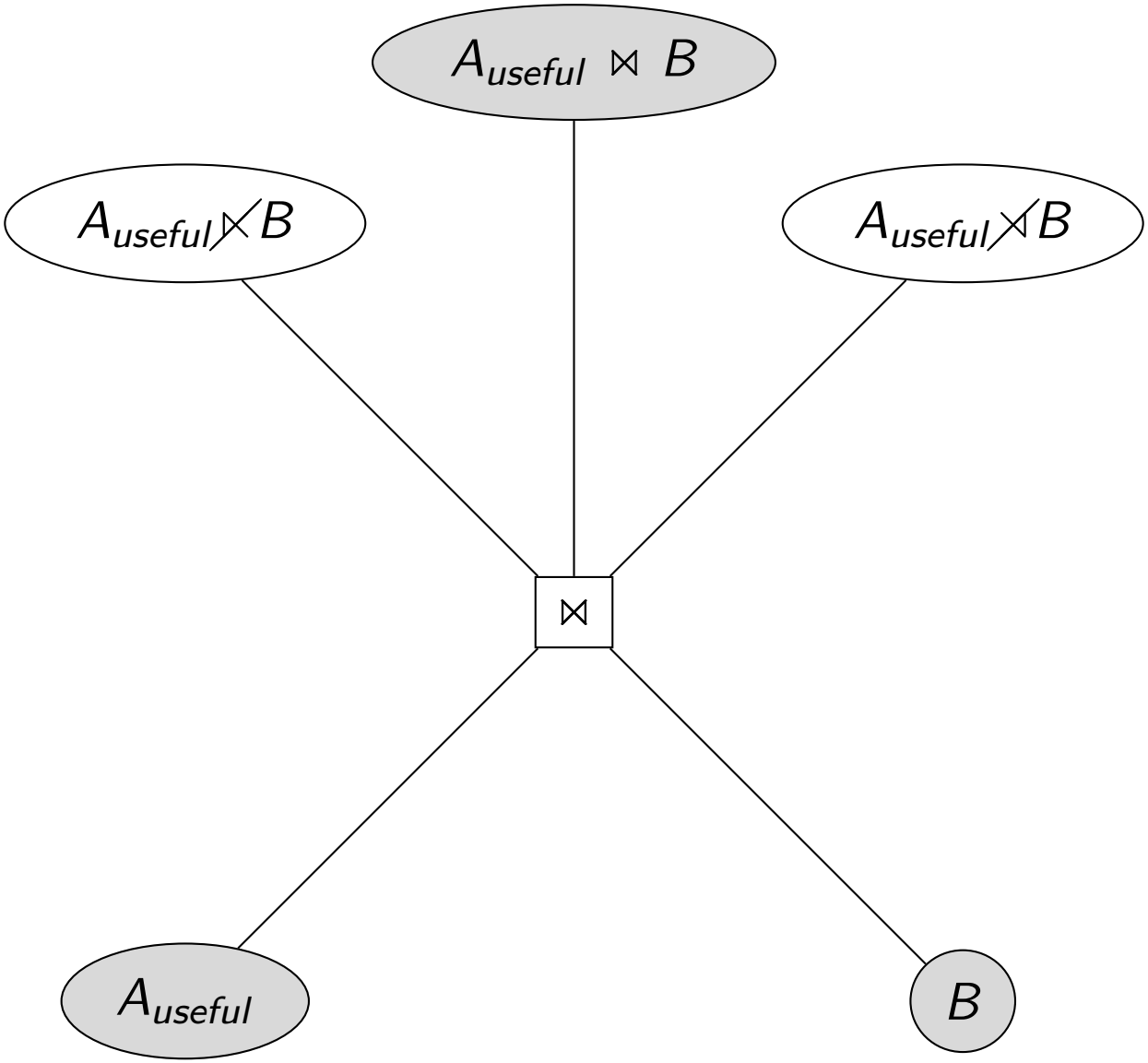
Plenty of memory



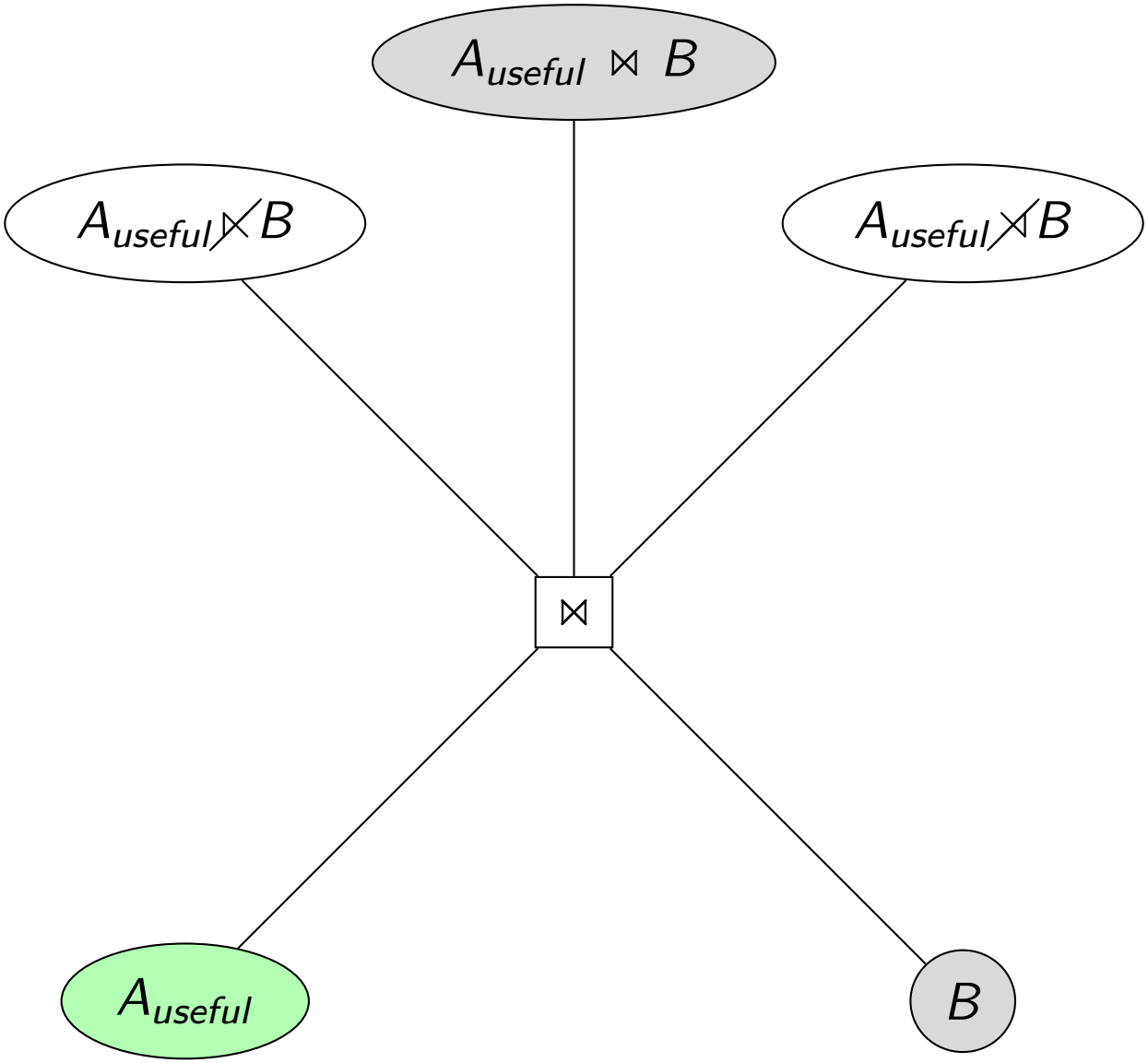
Being on a budget



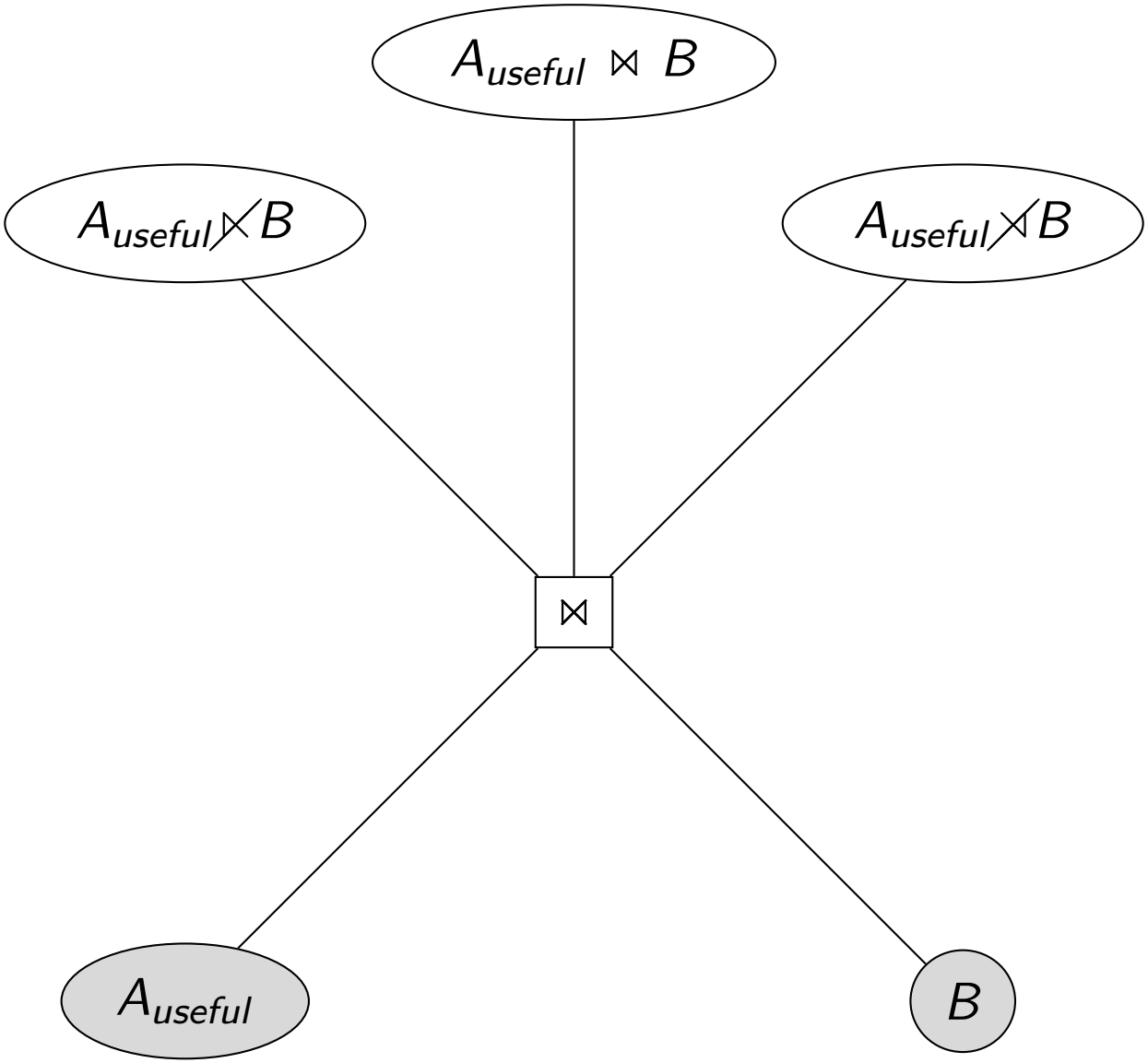
Being on a budget



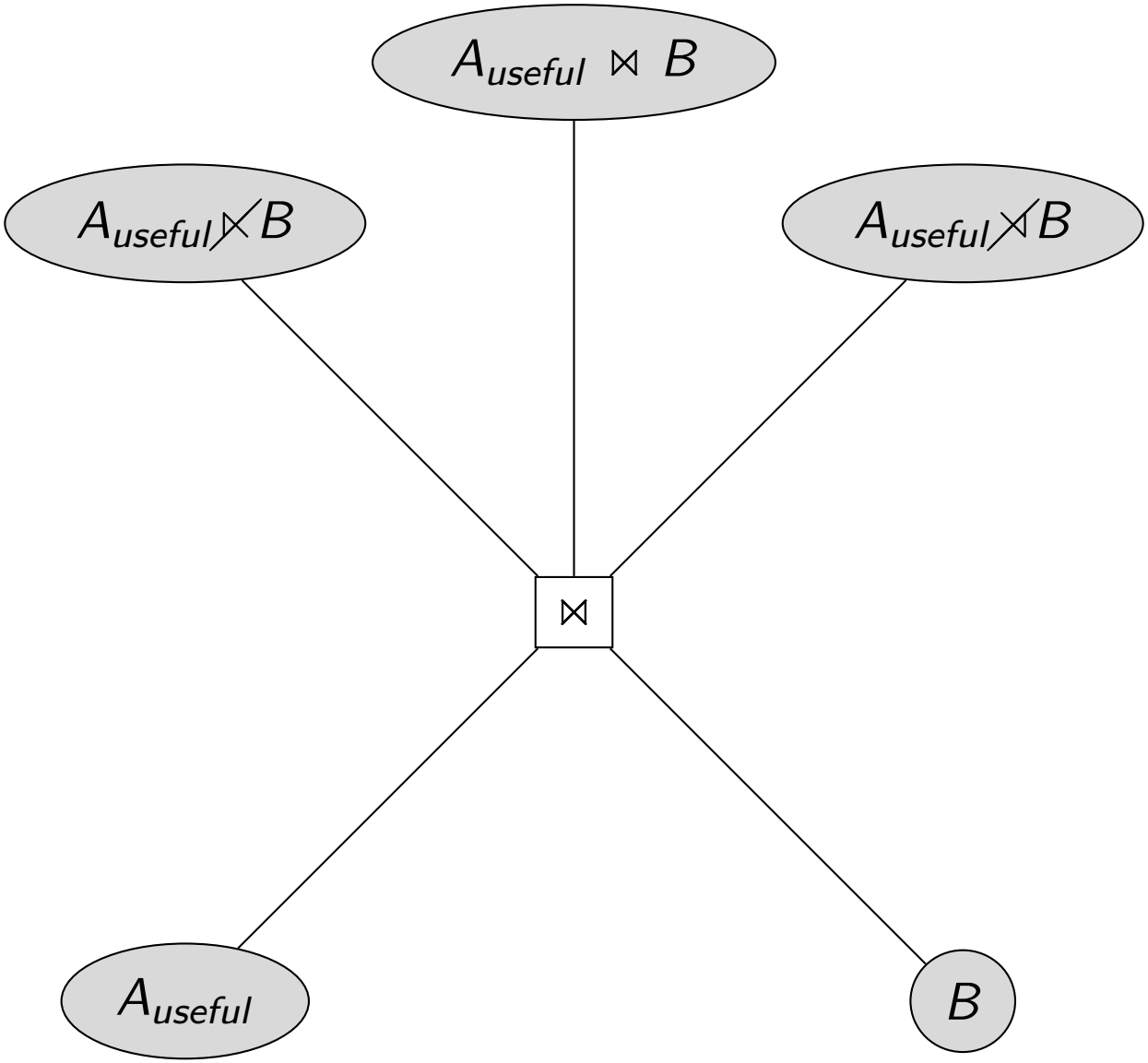
Being on a budget



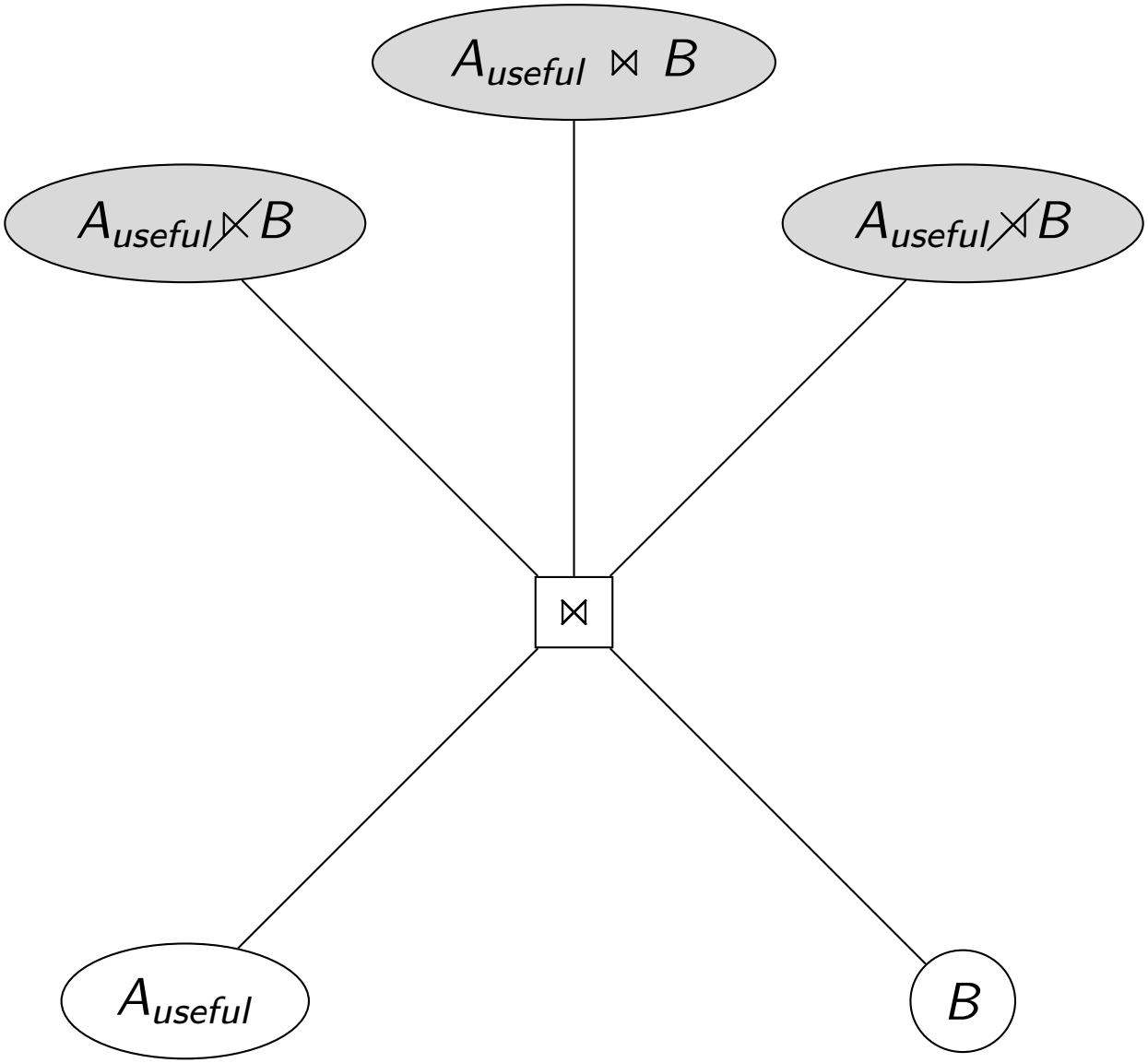
Having just enough rope



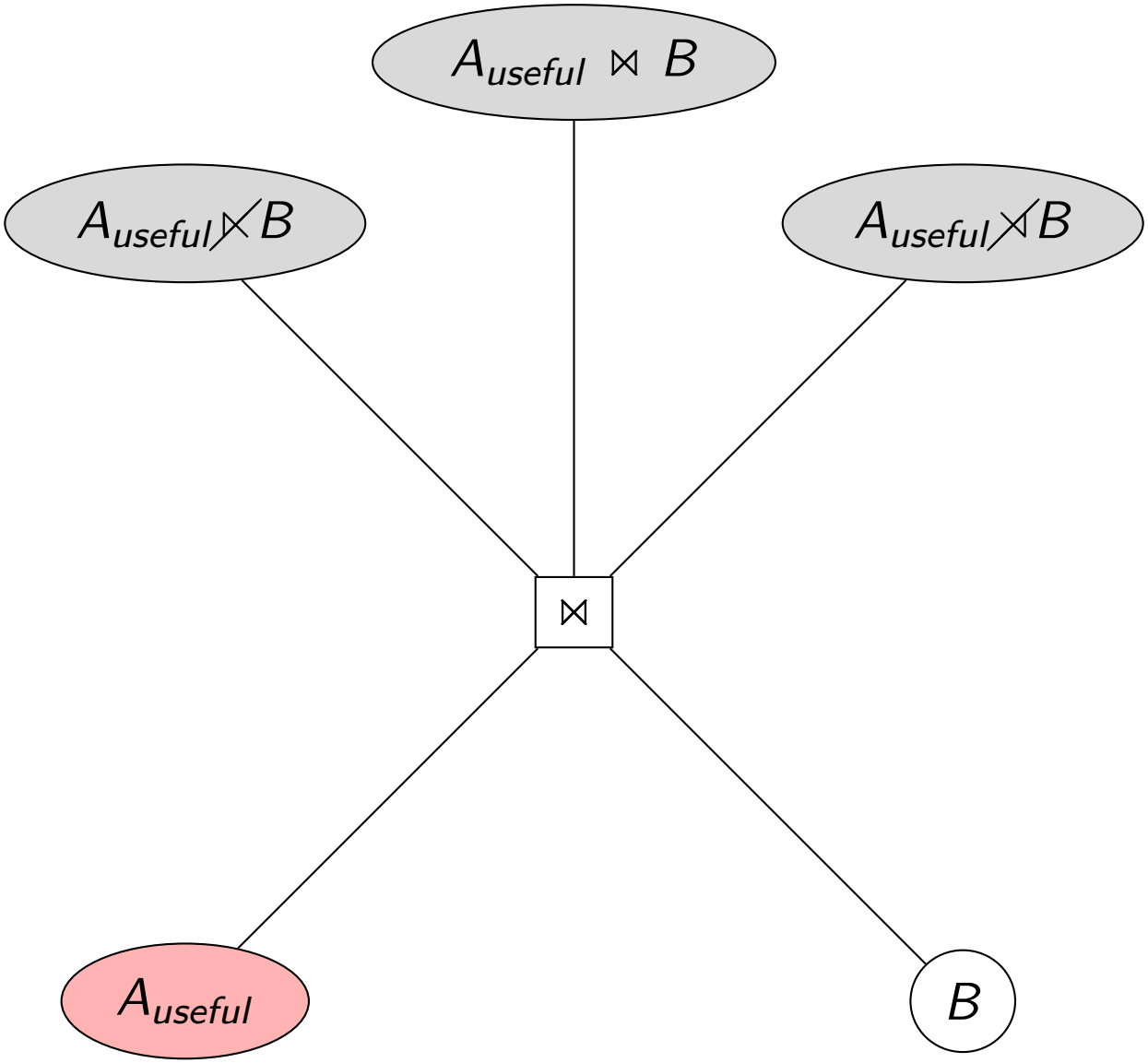
Having just enough rope



Having just enough rope



Having just enough rope



Coclusions and future perspectives

- FluiDB can efficiently use memory budget to store useful intermediate results.
- It would be interesting to:
  - Cardinality estimation is a major pain point for FluiDB, the architecture is accomodating to propagation of statistics
  - Parallel query processing
  - Support updates
  - extend the algebra with index-building operators.
  - Drop the C++ compiler.