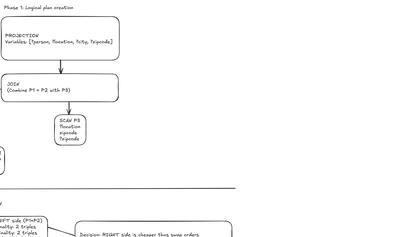
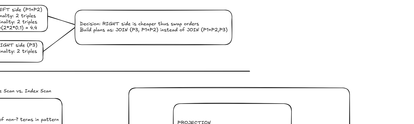
```
triples = [
    Pedro worksat KULAK
    Charlotte worksat Ubisoft
    KULAK located Kortrijk
    Ubisoft located Ghent
    KULAK episode 8000
    Ubisoft episode 9000
]
```

```
SELECT ?person ?place ?city ?episode WHERE {
  ?person worksat?x ?place .        # Pattern 1
  ?place located?y ?city .          # Pattern 2
  ?location earlyepisode ?episode .  # Pattern 3
}
```

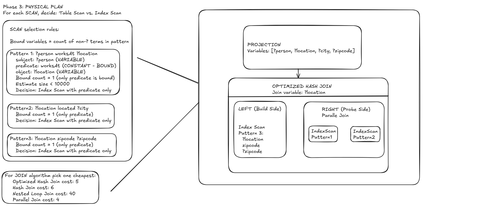
Phase 1: Logical plan creation



Phase 2: COST ESTIMATION
For JOIN (P1+P2, P3)



Phase 3: PHYSICAL PLAN
For each SCAN, decide: Table Scan vs. Index Scan



Phase 4: EXECUTION (Bottom-Up)

STEP 1: Execute IndexScan for Pattern 3 (?location earlyepisode ?episode)

Input Pattern:
subject: ?location (variable)
predicate: http://example.org/episode (CONSTANT - ID=5)
object: ?episode (variable)

Index Selection: Only predicate is bound -> Use PSO index (Predicate-Subject-Object)

Query PSO Index: pso.get(predicate_id=5)
Returns: [
[subject=2, object=6], // kulak -> "8000"
[subject=3, object=7] // ubisoft -> "9000"
]

Output Bindings (2 rows):
?location ?episode
2 (kulak) 6 ("8000")
3 (ubisoft) 7 ("9000")

STEP 2: Execute ParallelJoin (Pattern 1 ⋈ Pattern 2)

STEP 2a: Execute IndexScan for Pattern 1 (?person worksat ?location)

Input Pattern:
subject: ?person (variable)
predicate: http://example.org/worksat (CONSTANT - ID=1)
object: ?location (variable)

Query PSO Index: pso.get(predicate_id=1)
Returns: [
[subject=0, object=2], // peter -> kulak
[subject=1, object=3] // charlotte -> ubisoft
]

Output Bindings (2 rows):
?person ?location
0 (peter) 2 (kulak)
1 (charlotte) 3 (ubisoft)

STEP 2b: Execute IndexScan for Pattern 2 (?location located ?city)

Input Pattern:
subject: ?location (variable)
predicate: http://example.org/located (CONSTANT - ID=4)
object: ?city (variable)

Query PSO Index: pso.get(predicate_id=4)
Returns: [
[subject=2, object=4], // kulak -> kortrijk
[subject=3, object=5] // ubisoft -> ghent
]

Output Bindings (2 rows):
?location ?city
2 (kulak) 4 (kortrijk)
3 (ubisoft) 5 (ghent)

STEP 2c: Perform ParallelJoin using SIMD optimization

This uses: perform_join_per_simd_with_strict_filter_&_reordered_streaming

LEFT input (from Pattern 1):
?person ?location
0 2
1 3

RIGHT input (from Pattern 2):
?location ?city
2 4
3 5

Join on common variable: ?location

Process:
Row 1 LEFT: [?person:0, ?location:2]
Find matching RIGHT where ?location=2
-> Found: [?location:2, ?city:4]
-> Merge: [?person:0, ?location:2, ?city:4]

Row 2 LEFT: [?person:1, ?location:3]
Find matching RIGHT where ?location=3
-> Found: [?location:3, ?city:5]
-> Merge: [?person:1, ?location:3, ?city:5]

Output (2 rows):
?person ?location ?city
0 2 4
1 3 5

STEP 3: Execute OptimizedHashJoin (Pattern 3 ⋈ Result from ParallelJoin)

LEFT (Build Side - Pattern 3):
?location ?episode
2 6
3 7

RIGHT (Probe Side - ParallelJoin):
?person ?location ?city
0 2 4
1 3 5

Common variable: ?location

BUILD PHASE (Create hash table from LEFT):

Hash Table:
{
[2] [[?location:2, ?episode:6]], // Key is VarId32 with join var ID=2
[3] [[?location:3, ?episode:7]]
}

PROBE PHASE (Query hash table with RIGHT rows):

Probe Row 1: [?person:0, ?location:2, ?city:4]
Extract join key [2]
Look up in hash table: Found [[?location:2, ?episode:6]]
Merge: [?person:0, ?location:2, ?city:4] + [?location:2, ?episode:6]
Result: [?person:0, ?location:2, ?city:4, ?episode:6]

Probe Row 2: [?person:1, ?location:3, ?city:5]
Extract join key [3]
Look up in hash table: Found [[?location:3, ?episode:7]]
Merge: [?person:1, ?location:3, ?city:5] + [?location:3, ?episode:7]
Result: [?person:1, ?location:3, ?city:5, ?episode:7]

Output (2 rows):
?person ?location ?city ?episode
0 2 4 6
1 3 5 7

STEP 4: Execute Projection (Select only requested variables)

Keep only: [?person, ?location, ?city, ?episode]
(In this case, no filtering needed - all variables are already present)

STEP 5: Decode IDs to Strings

0 -> peter 2 -> kulak 4 -> kortrijk 6 -> "8000"
1 -> charlotte 3 -> ugent 5 -> ghent 7 -> "9000"

FINAL OUTPUT:

Row 1: ?person=http://example.org/peter,
?location=http://example.org/kulak,
?city=http://example.org/kortrijk,
?episode="8000"

Row 2: ?person=http://example.org/charlotte,
?location=http://example.org/ugent,
?city=http://example.org/ghent,
?episode="9000"

Original query has 3 patterns:
P1: ?person worksat ?location
P2: ?location located ?city
P3: ?location earlyepisode ?episode

Question: What's the join order?

Option A: ((P1 ⋈ P2) ⋈ P3) <- what optimizer chooses!
Option B: ((P1 ⋈ P3) ⋈ P2)
Option C: ((P2 ⋈ P3) ⋈ P1)

Optimizer's reasoning:

1. P1 and P2 share ?location -> good to join first
2. Result of (P1 ⋈ P2) is small: 2 rows
3. P3 is also small: 2 rows
4. Final join (?P1+P2) ⋈ P3 is efficient

Cost analysis:
- P1 ⋈ P2 using ParallelJoin: cost = 4
- Result ⋈ P3 using OptimizedHashJoin: cost = 5
- Total = 9

Alternative (P1 ⋈ P3 first):
- P1 ⋈ P3 cost: 6
- Result ⋈ P2 cost: 6
- Total: 12 <- More expensive!

So Option A wins! -> ParallelJoin becomes a subtree

Hash JOIN:
Step 1 Build phase (Sequential)
For each row in LEFT:
key = extract_join_variables(row)
hash_table[key].push(row)

Step 2: PROBE phase (Sequential)
For each row in RIGHT:
key = extract_join_variables(row)
if hash_table.contains(key):
for matching_row in hash_table[key]:
output(merge(matching_row, row))

Parallelization: SIMD (sequential)
Data Structure: HashMap<VarId32, Vec<BTreeMap>>

OPTIMIZED HASH JOIN:
Step 1 Dynamic Build side selection
Choose smaller relation as build side
(Prefers hash table memory)

Step 2: PARALLEL BUILD phase
build_side.par_for_each(|row| {
field[] HashMap<var0, Vec, var1> {
// Each thread builds local hash table
acc.insert(key, row);
acc
})
.reduce(|| HashMap<var0, Vec, var2> {
// Merge all local hash tables
merge(var1, var2)
})

Step 3: PARALLEL PROBE phase
probe_side.par_for_each(
flat_map(row {
// Each thread probes independently
lookup_and_join(hash_table, row)
})

Parallelization: Rayon (fold + reduce + flat_map)
Data Structure: HashMap<VarId32, Vec<BTreeMap>>

PARALLEL JOIN (MERGE_SORT) + SIMD:
- Step 1: SORT both inputs
- Sort LEFT by join variable
- Sort RIGHT (triples) by subject, then object
Purpose: Enable efficient merge-based matching

Step 2: FILTER using SIMD
- Use predicate_id for fast ID comparison
- Filter triples that don't match predicate
#[target_feature = "avx2,fma"]
- predicate == predicate_id

Step 3: PARALLEL MERGE JOIN
- Split sorted triples into chunks
- Each thread performs merge-based matching
for each triple in chunk:
binary_search(sorted_bindings, triple.subject)
if found:
merge_and_output()

Step 4: COMBINE results
- Reduce all chunk results into final output

Parallelization: Rayon par_chunks + fold/reduce
Optimization: Sorted data + Binary search + SIMD

MERGE-SORT ALGORITHM

LEFT (results from previous join)

?person ?location ?city

0 2 4
1 3 5

RIGHT (new pattern to join): ?location earlyepisode ?episode
This is represented as a TriplePattern, not pre-computed results

Triples in database:
Subject Predicate Object
2 5 6 // kulak episode "8000"
3 5 7 // ugent episode "9000"

Pattern to match: ?location (subject) episode (predicate=5) ?episode (object)

SORT LEFT BINDINGS FOR MERGE JOIN

Input LEFT: [
[?person:0, ?location:2, ?city:4],
[?person:1, ?location:3, ?city:5]
]

Sort by join variable (?location):

After sorting:
?person ?location ?city
0 2 4
1 3 5

FILTER TRIPLES USING SIMD IDS

All triples in database:
Subject Predicate Object
0 1 3
1 1 4
2 4 5
3 4 5
2 5 6 <- Match
3 5 7 <- Match

Pattern: ?location earlyepisode ?episode
predicate_id = 5 (episode)

Filter step (SIMD-optimized ID comparison):
for triple in triples:
if triple.predicate == 5: <- Fast integer comparison!
keep(triple)

Filtered triples:
Subject Predicate Object

2 5 6 <- Subject=2
3 5 7 <- Subject=3

PARALLEL MERGE JOIN (4 threads)

Sorted LEFT bindings:
[
[?person:0, ?location:2, ?city:4],
[?person:1, ?location:3, ?city:5]
]

Sorted RIGHT triples:
[
Triple(subject=2, predicate=5, object=6),
Triple(subject=3, predicate=5, object=7)
]

Split triples into chunks for parallel processing:
Chunk 1 (Thread 1): [Triple(2,5,6)]
Chunk 2 (Thread 2): [Triple(3,5,7)]

THREAD 1: Process Triple(2,5,6)
Triple: subject=2, predicate=5, object=6

1. Binary search LEFT for ?location=2:
Search sorted_bindings for key 2
Found: [?person:0, ?location:2, ?city:4]

2. Match found! Merge:
LEFT: [?person:0, ?location:2, ?city:4]
RIGHT: [?location:2, ?episode:6] <- extracted from triple

Merged [?person:0, ?location:2, ?city:4, ?episode:6]

3. Output to thread-local results

THREAD 2: Process Triple(3,5,7)
Triple: subject=3, predicate=5, object=7

1. Binary search LEFT for ?location=3:
Search sorted_bindings for key 3
Found: [?person:1, ?location:3, ?city:5]

2. Match found! Merge:
LEFT: [?person:1, ?location:3, ?city:5]
RIGHT: [?location:3, ?episode:7]

Merged [?person:1, ?location:3, ?city:5, ?episode:7]

3. Output to thread-local results

REDUCE: Combine thread-local results
Thread 1 output: [[?person:0, ?location:2, ?city:4, ?episode:6]]
Thread 2 output: [[?person:1, ?location:3, ?city:5, ?episode:7]]

Final combined result:
[
[?person:0, ?location:2, ?city:4, ?episode:6],
[?person:1, ?location:3, ?city:5, ?episode:7]
]

NESTED-LOOP JOIN
for each left_row in LEFT:
for each right_row in RIGHT:
if can_join(left_row, right_row):
output(merge(left_row, right_row))

Complexity: O(|LEFT| * |RIGHT|)
Only used when both inputs are small (< 1000 rows)