Quetzal - Core ideas

Two core ideas in Quetzal

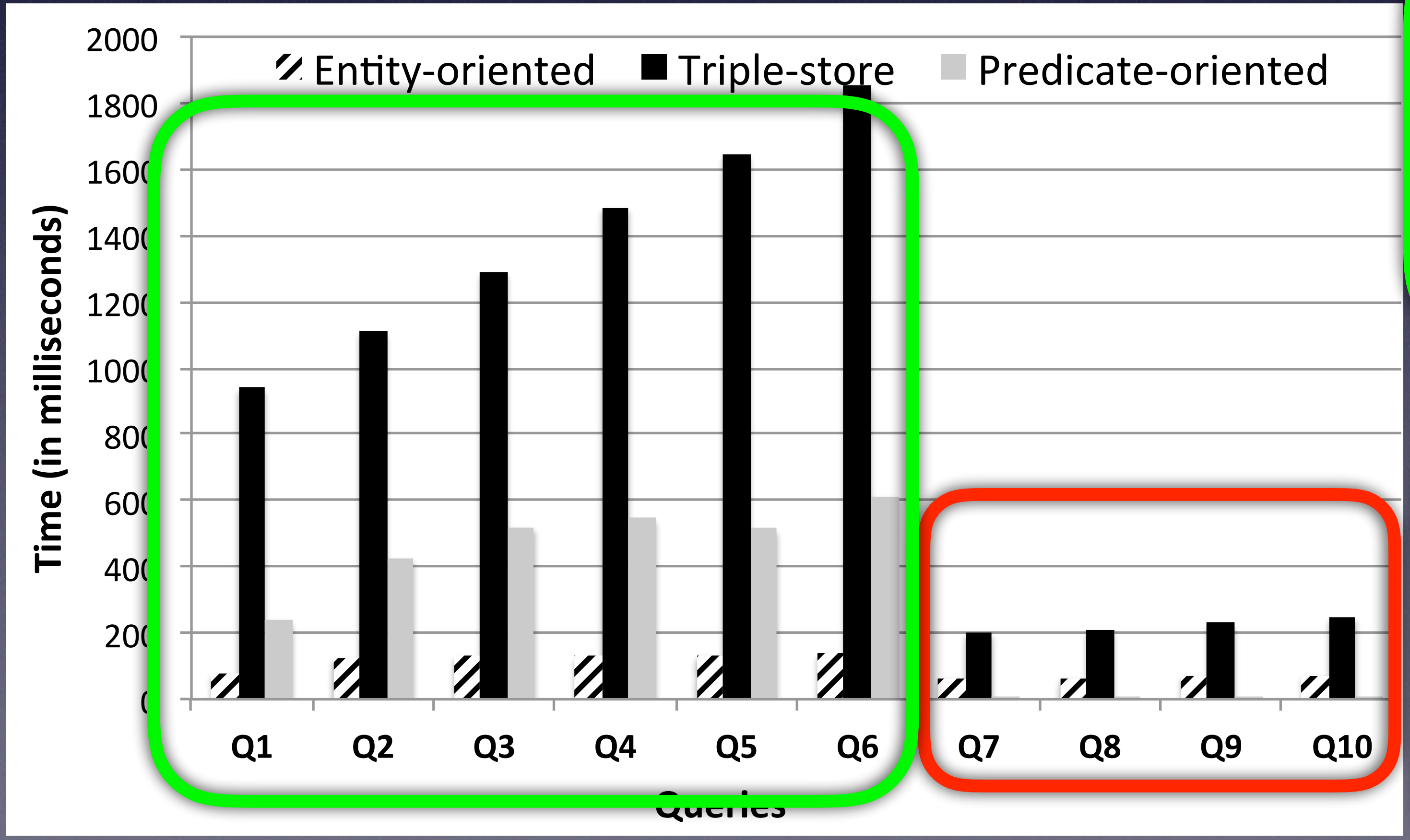
- Entity oriented layout for graph data, data duplicated in forward and backward direction
- A SPARQL to SQL translator for querying graph data (can be adapted to different schemas). Similar approach for Gremlin but inherently restricted because its not declarative like SPARQL.

Implemented on 3 different backends (DB2, Postgresql, Apache Spark) is available for experimentation in a project called **Quetzal:** github: https://github.com/Quetzal-RDF/quetzal

Why Entity Oriented Storage?

Star queries are a common component of graph queries

Micro benchmarks to manipulate selectivity of predicates when only in a star showed that entity oriented storage shows the most consistent performance



Q1-Q6 stars were selective but any given predicate in the star was not.

Q7-Q10 stars had at least 1 predicate that was highly selective. Note predicate oriented storage is better.

SIGMOD 2013

The layout

source	label	target	label	target	label	target
Flint	born	1850	died	1934	found	IBM
IBM	industry	1				

Multiple labels *hashed* to same column

source	id	label	target
IBM	1	industry	Software
IBM	1	industry	Hardware
IBM	1	industry	Services

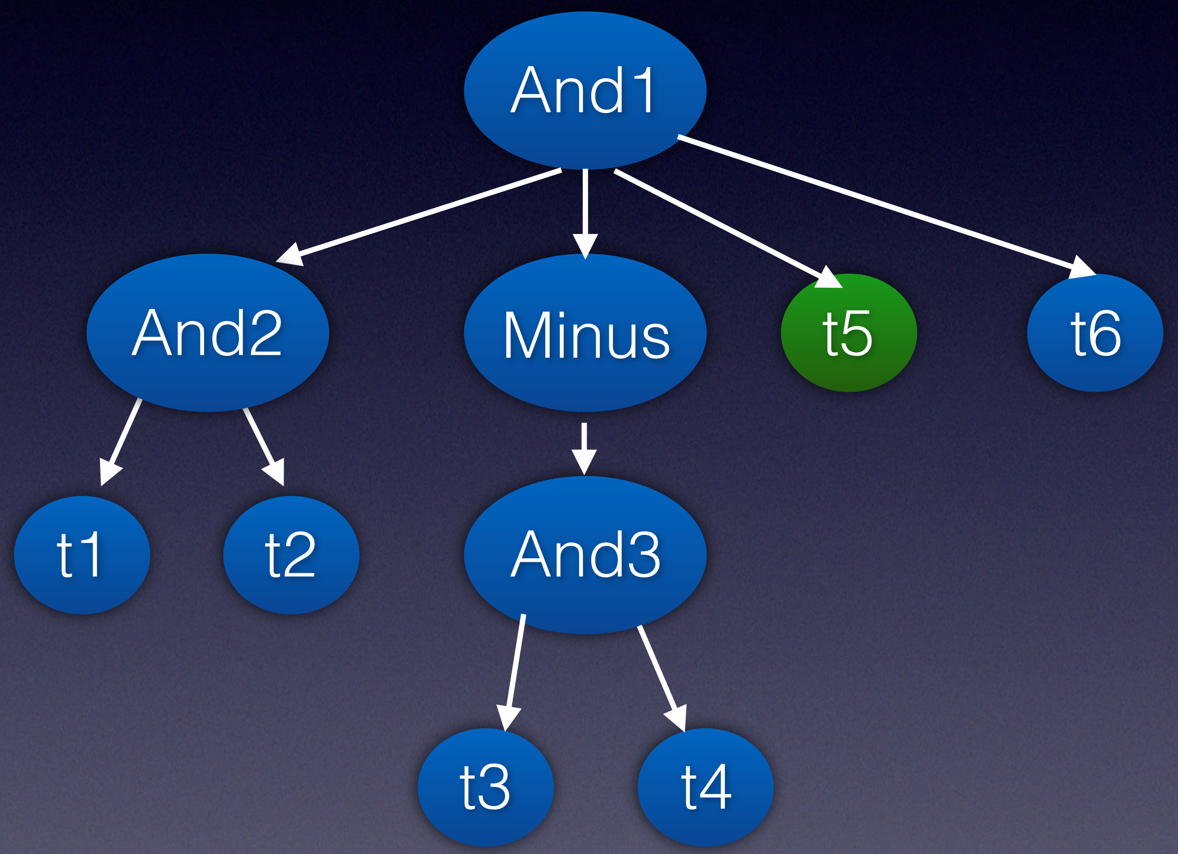
Multiple values stored in secondary table. Note the duplication of source and label here. Useful to avoid joins in cases where query only asks multi-valued labels

Index on source

Duplicate structure for backward edges

SPARQL to SQL Translator

Core idea - each query has a 'data flow', but one needs to respect the semantics of the query when following the flow.



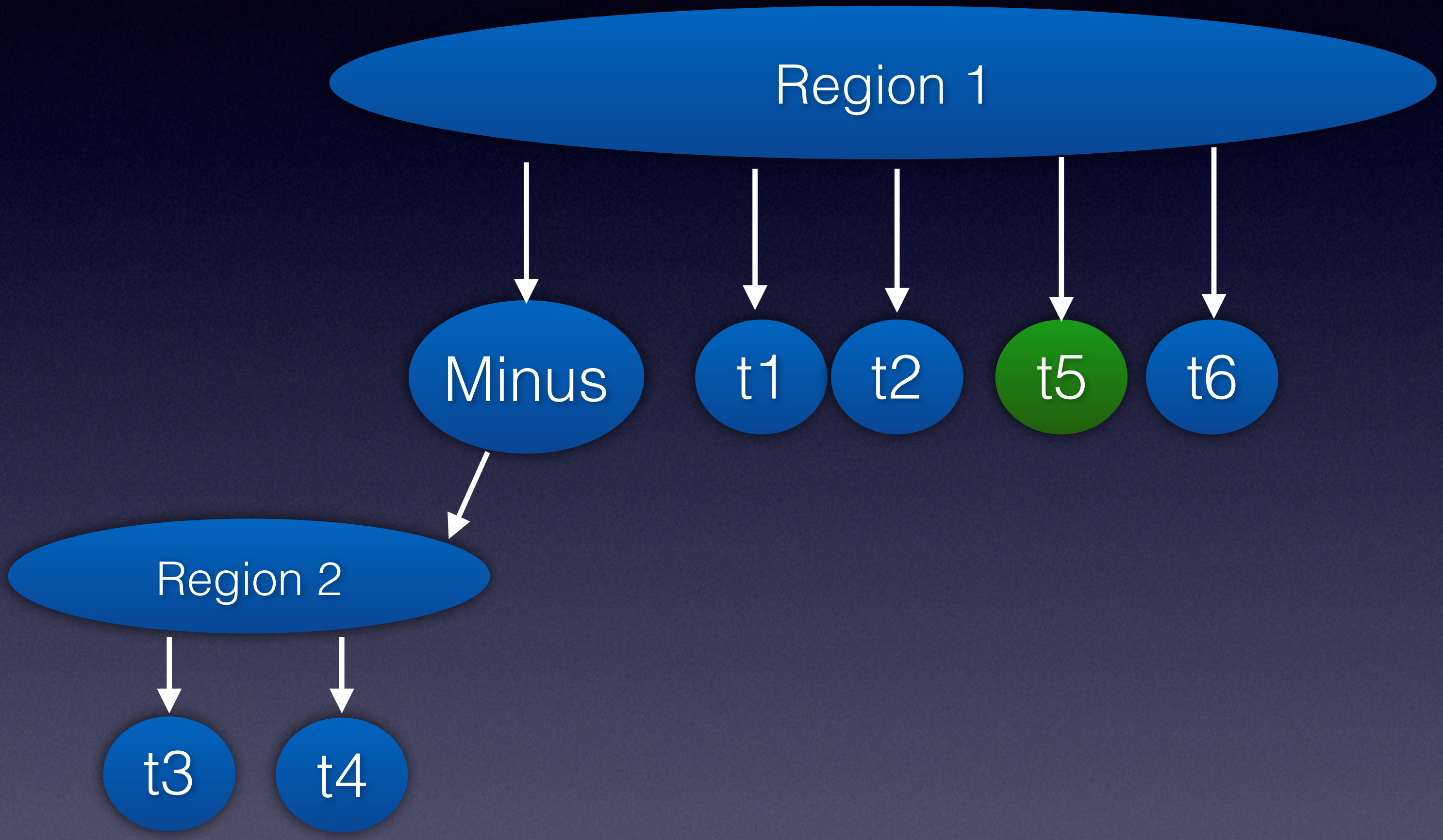
t1-6 reflect an edge in the query graph.

Executing t5 makes ?x *available*, which will make t1 "cheaper" to execute. Costs are based on whether an index can be used to execute the triple.

Query parse tree for a SPARQL query: t6 is the "cheapest" for a backward edge lookup, and needs nothing bound

Greedy planning

Plan recursively by “regions”

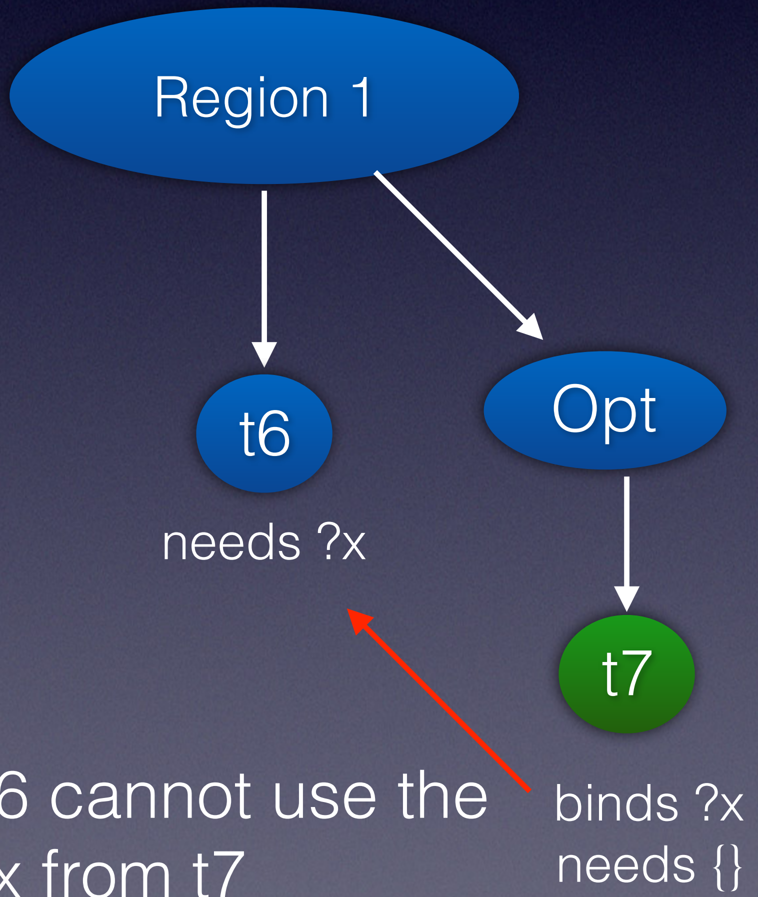


1. Gather all nodes to be ANDed into regions
2. Start with the topmost region (region 1).
3. Find the cheapest node in region (t5)
4. Execution of t5 makes some variables available.
5. Examine the next cheapest node to execute
6. Continue till no nodes are left unplanned.

The planner will recursively call plan on complex nodes such as UNION or OPTIONAL which may have their own regions to determine their cost, given the current set of bound (or available) variables.
Produces an optimal plan for 84/91 queries (WISE 2014)

Adherence to semantics

Data flow must follow query semantics



Invalid flow: t6 cannot use the bindings of ?x from t7

1. For each region, determine the subset of nodes that can be planned.
2. For optionals, an optional cannot be planned unless all of the variables that it shares with its region (?x) are available.

Semantics of query is thus respected by computing what nodes can be planned given the current set of bound variables.

Example Query

```
select ?str {
    { ?x :p ?y    T1
         ?y a Bar T2
    } MINUS {
         ?x :q ?z T3
         ?z :r ?r T4
    }
    ?x a Foo T5
    ?x :name ?str T6
}
```

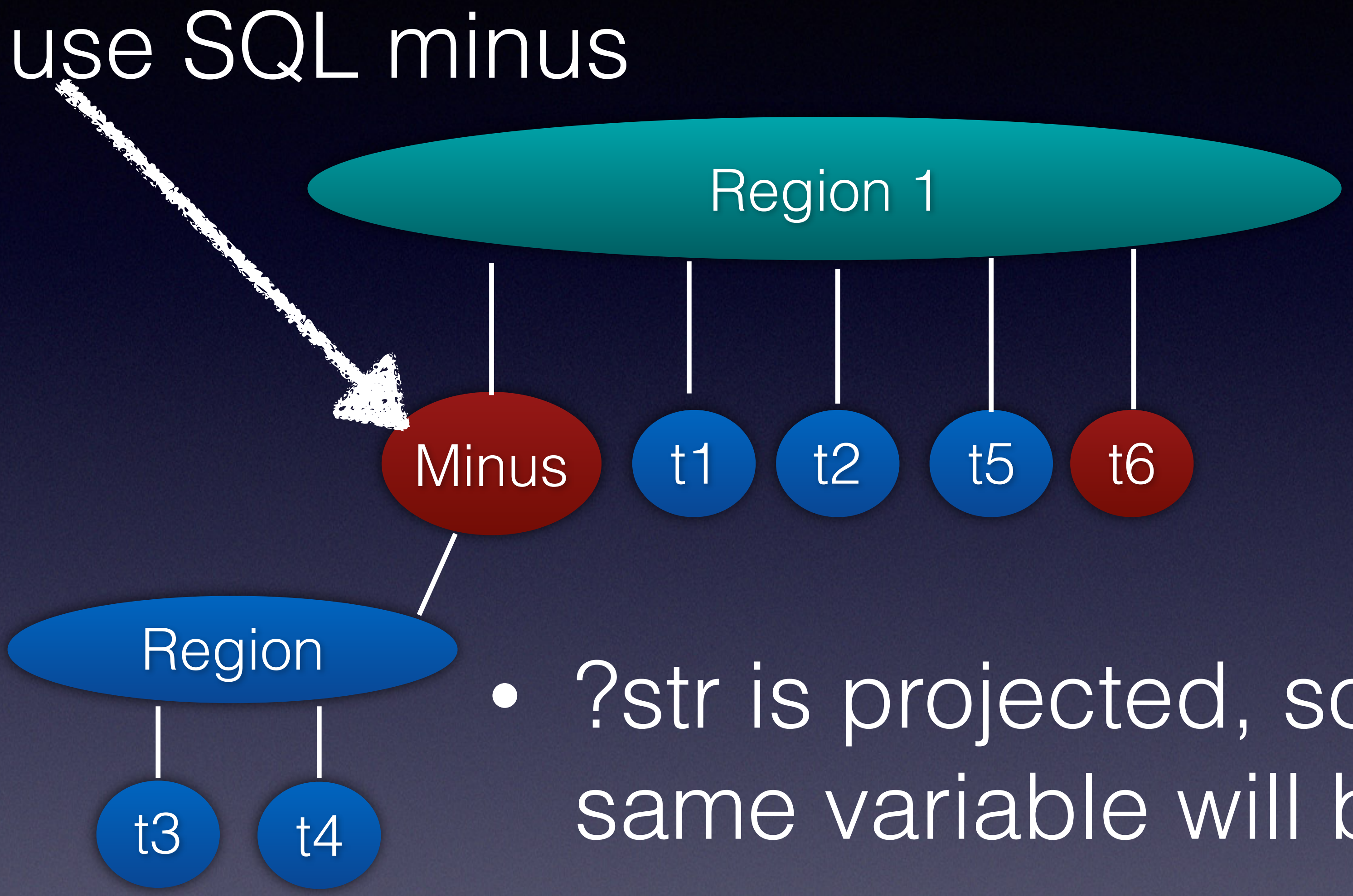
SQL Minus

- SQL Minus is an efficient set difference in RDBMs
  - much faster than joins
- Very specific requirements
  - Left and right hand sides must have the same variables
- Use SQL Minus if possible for SPARQL negation constructs

  `(optional !bound(), filter not exists, minus)`

  - Use regular joins otherwise

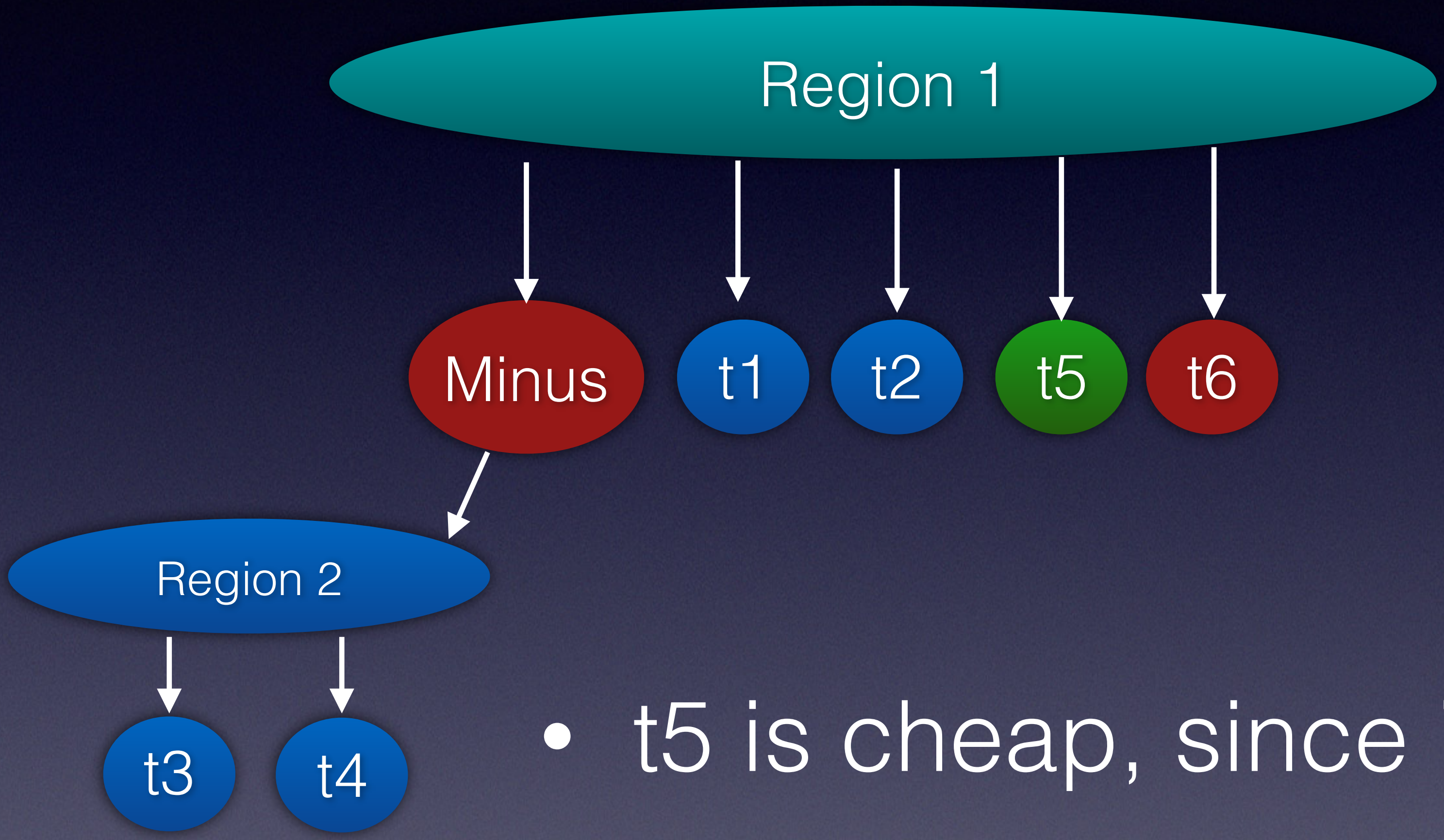
Greedy planning



Available:
Live: ?str

- ?str is projected, so it is live, meaning the same variable will be used again
- minus cannot done, since ?x not available
- t6 must be done after minus to produce ?str

Greedy planning

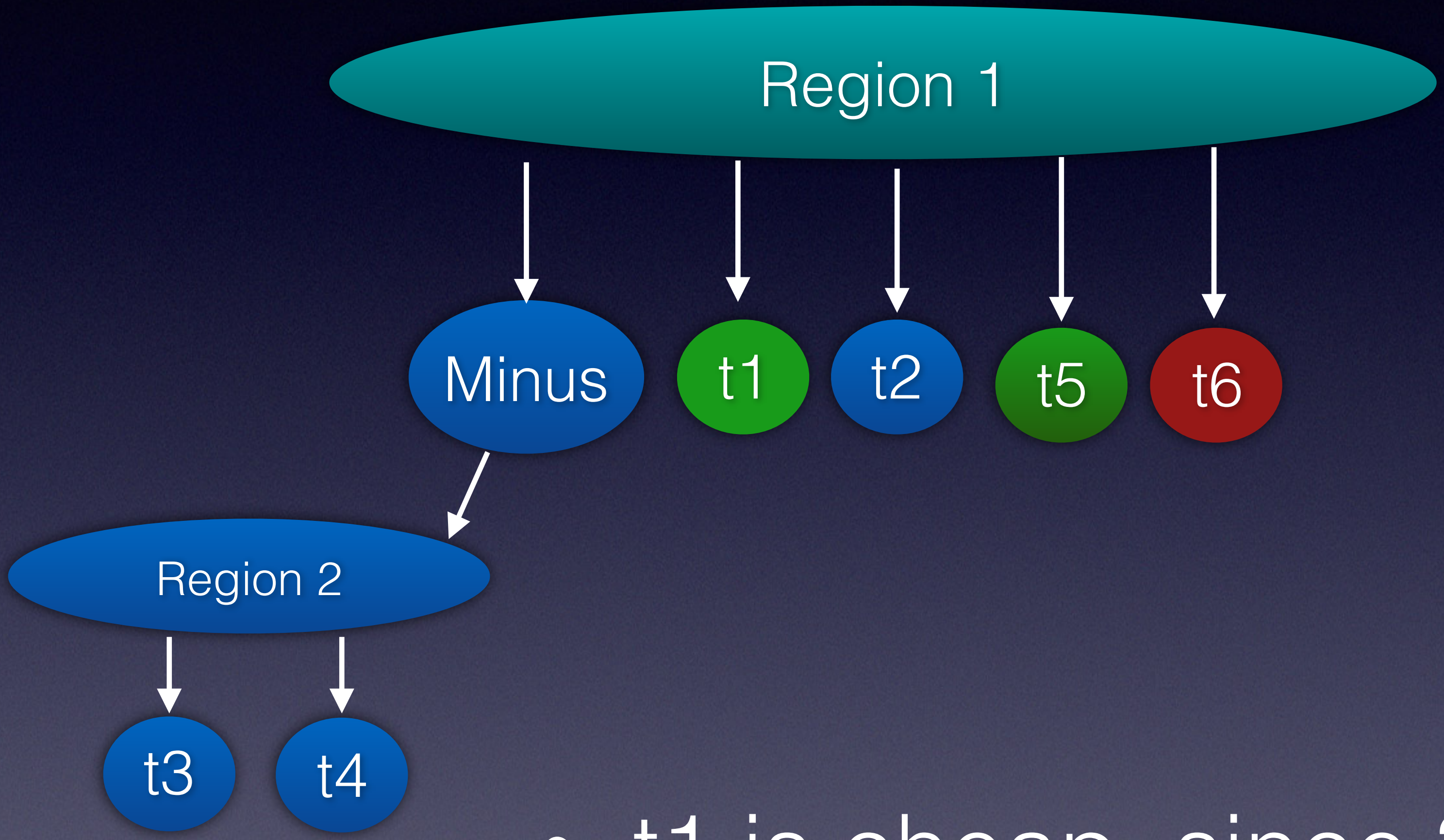


Available:
Live: ?x ?y ?str

```
?x a Foo .
```

- t5 is cheap, since it looks up a constant
- t5 is chosen to do first
- All variables from Region 1 become live

Greedy planning

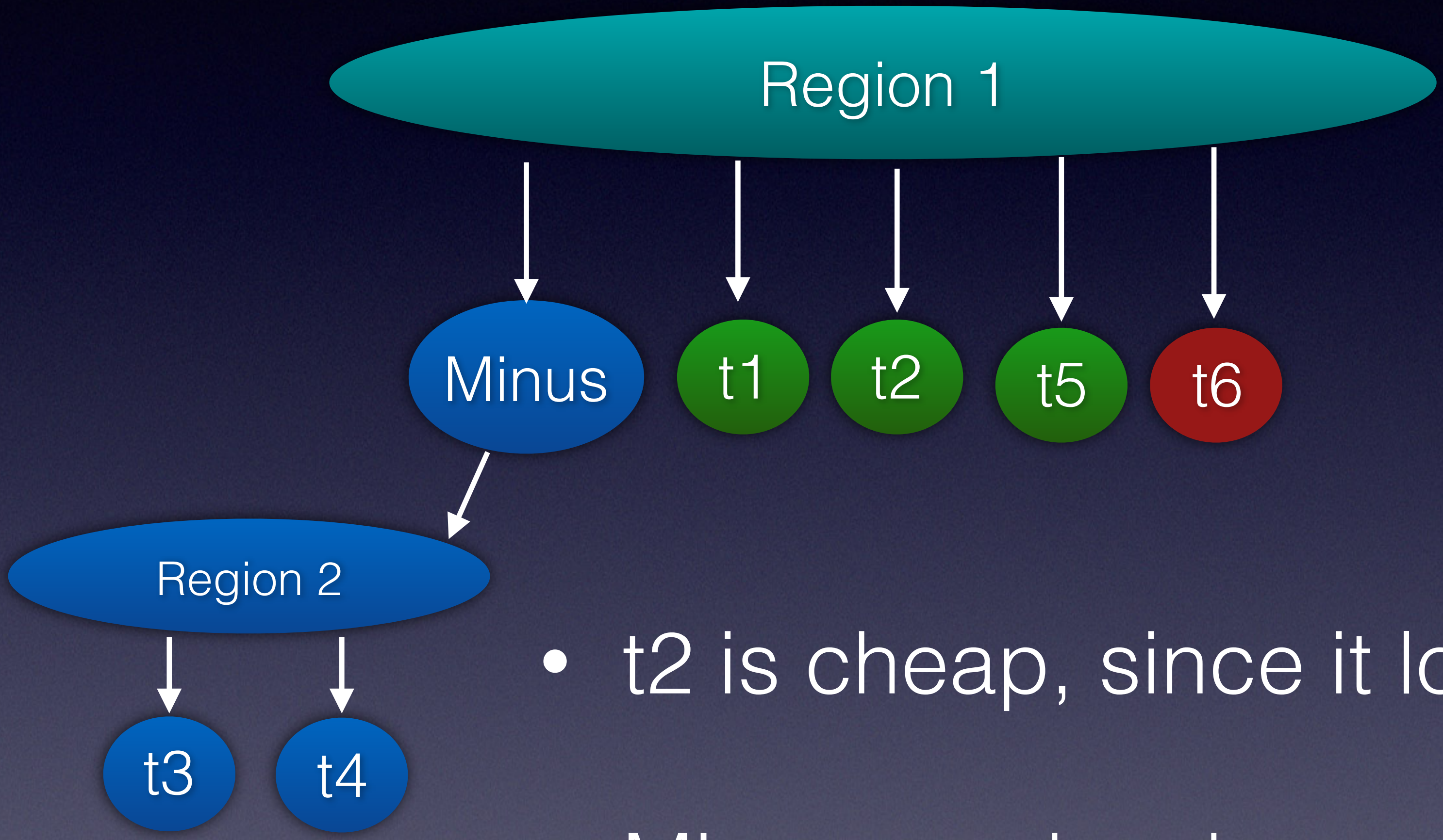


Available: ?x
Live: ?x ?y ?str

```
?x :p ?y.
```

- t1 is cheap, since ?x is available
- t1 is chosen to do next

Greedy planning

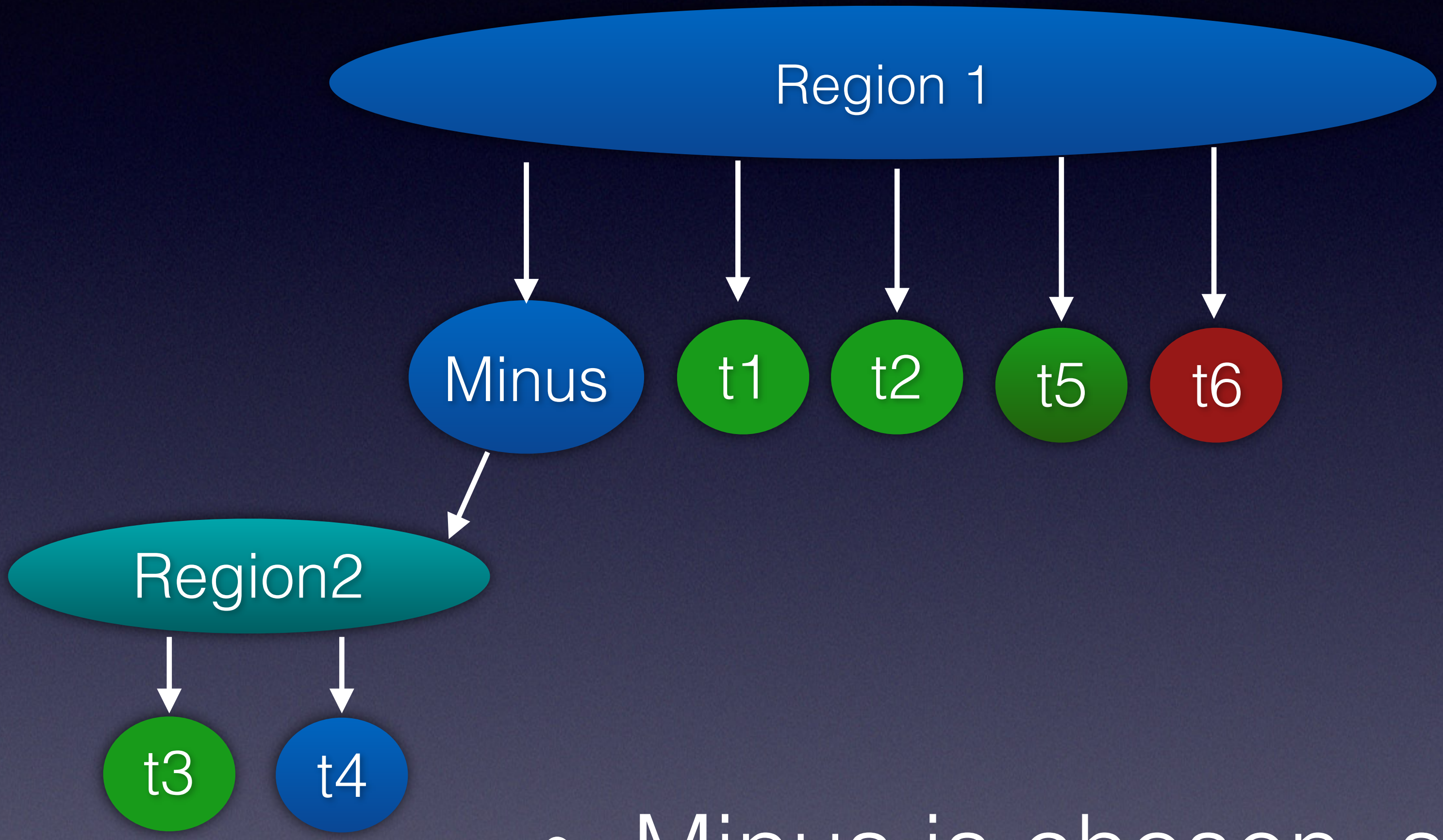


Available: ?x
Live: ?x ?str

```
?y a Bar .
```

- t2 is cheap, since it looks up a constant
- Minus can be done, as ?x and ?y available
- t2 is chosen to do next
- No more uses of ?y so it is no longer live

Greedy planning

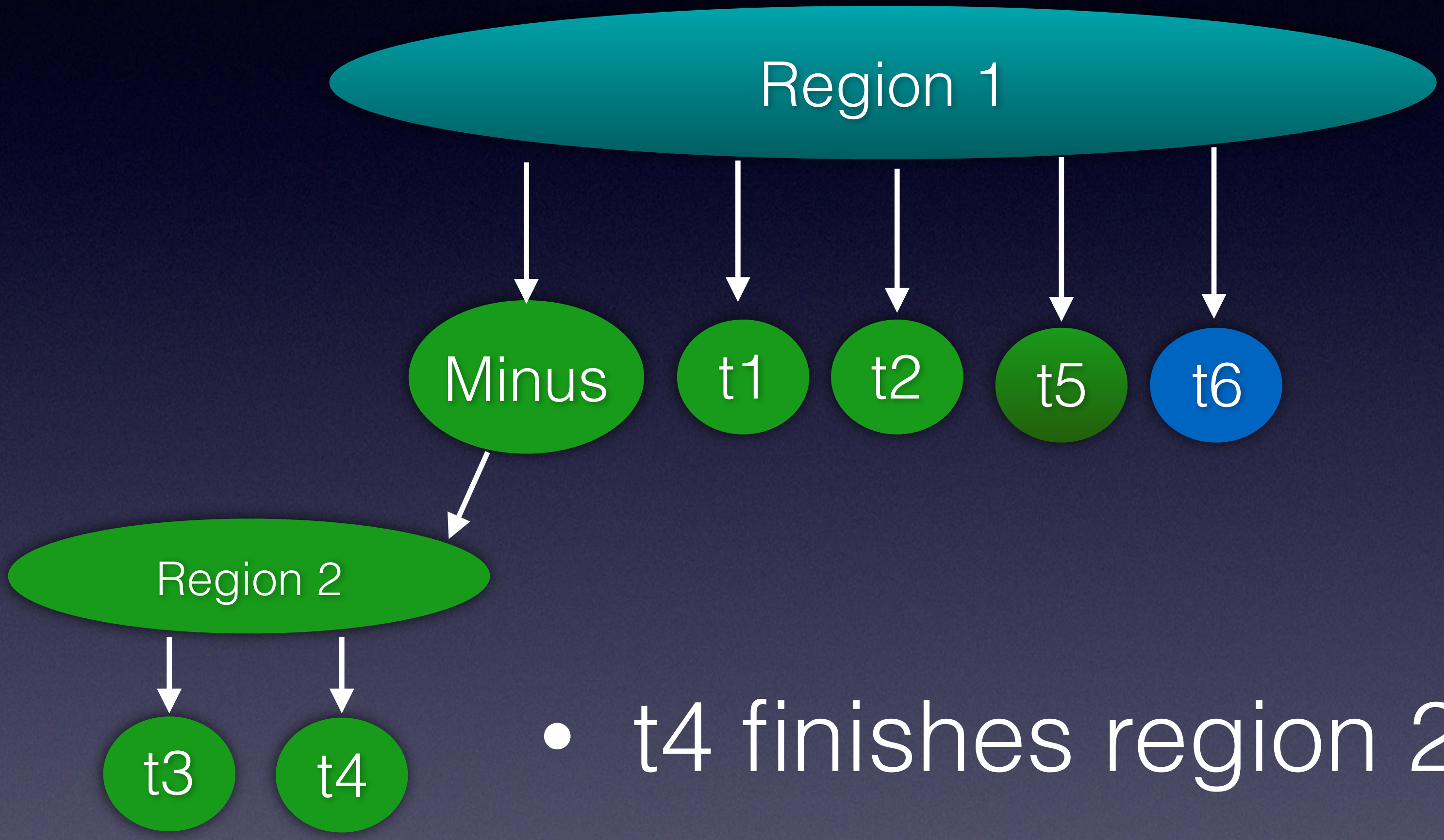


Available: ?x
Live: ?x ?z ?r ?str

```
?x :q ?z .
```

- Minus is chosen, so plan region 2
- t3 is cheaper, since ?x available

Greedy planning

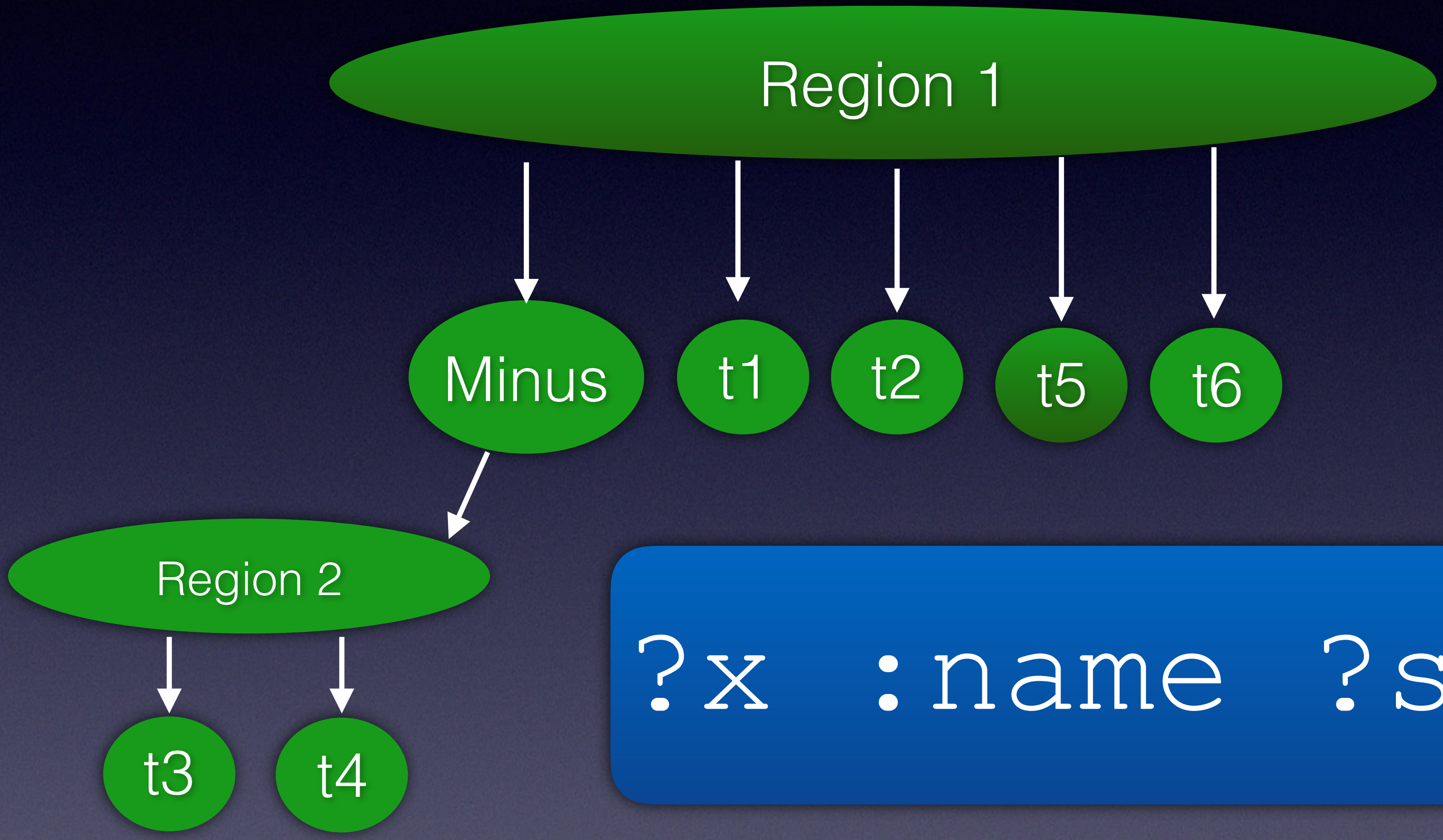


Available: ?x
Live: ?x ?str

```
?z :r ?r .
```

- t4 finishes region 2 and minus
- now t6 can be done, since minus done
- ?r and ?z are no longer live

Greedy planning



Available:?str
Live: ?str

```
?x :name ?str .
```

- finish with t6

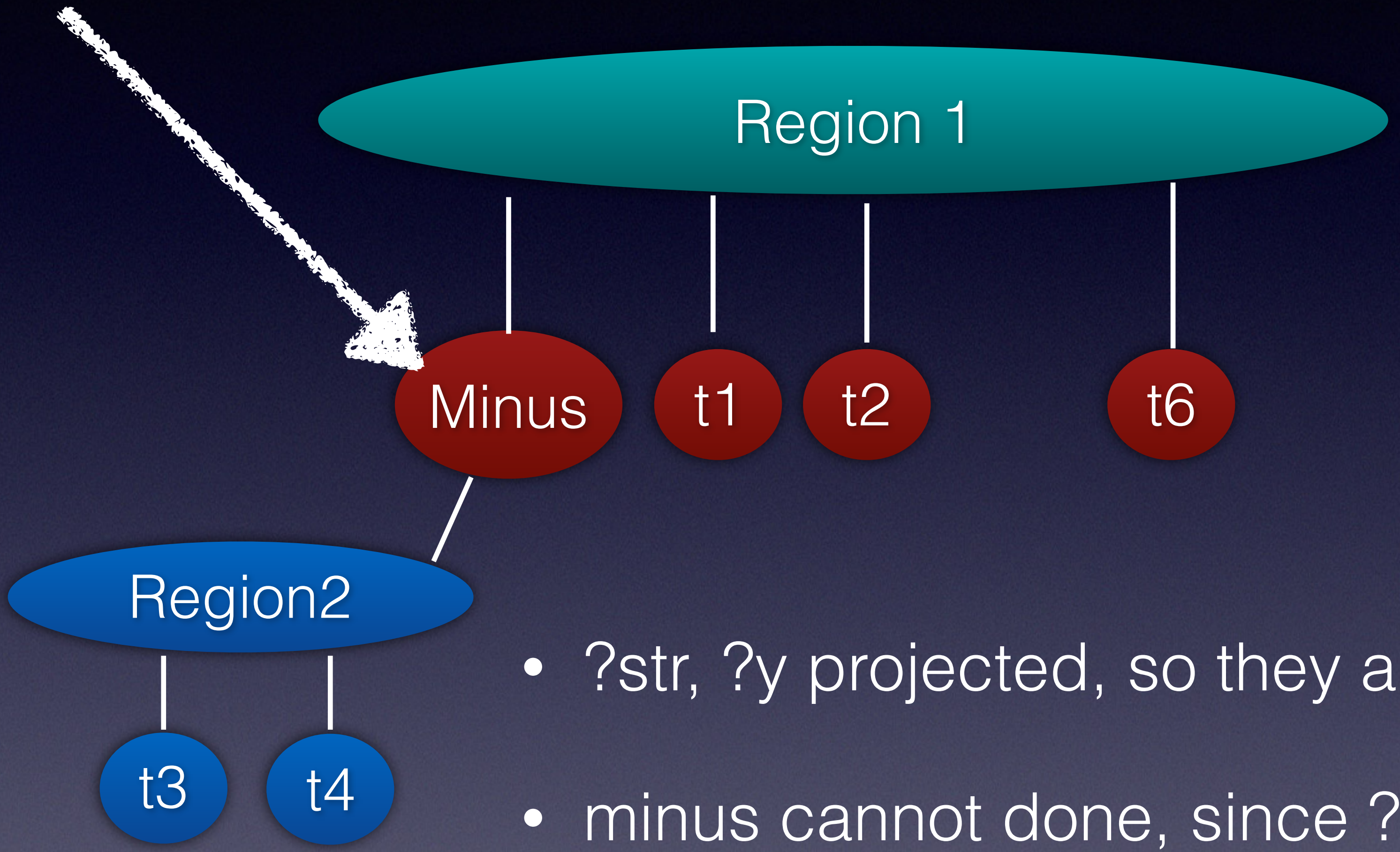
Bad Minus Example

```
select ?y ?str {
  { ?x :p ?y      T1
    ?y a Bar      T2
  } MINUS {
    ?x :q ?z      T3
    ?z :r ?r      T4
  }

  ?x :name ?str   T6
}
```

Greedy planning

cannot use SQL minus



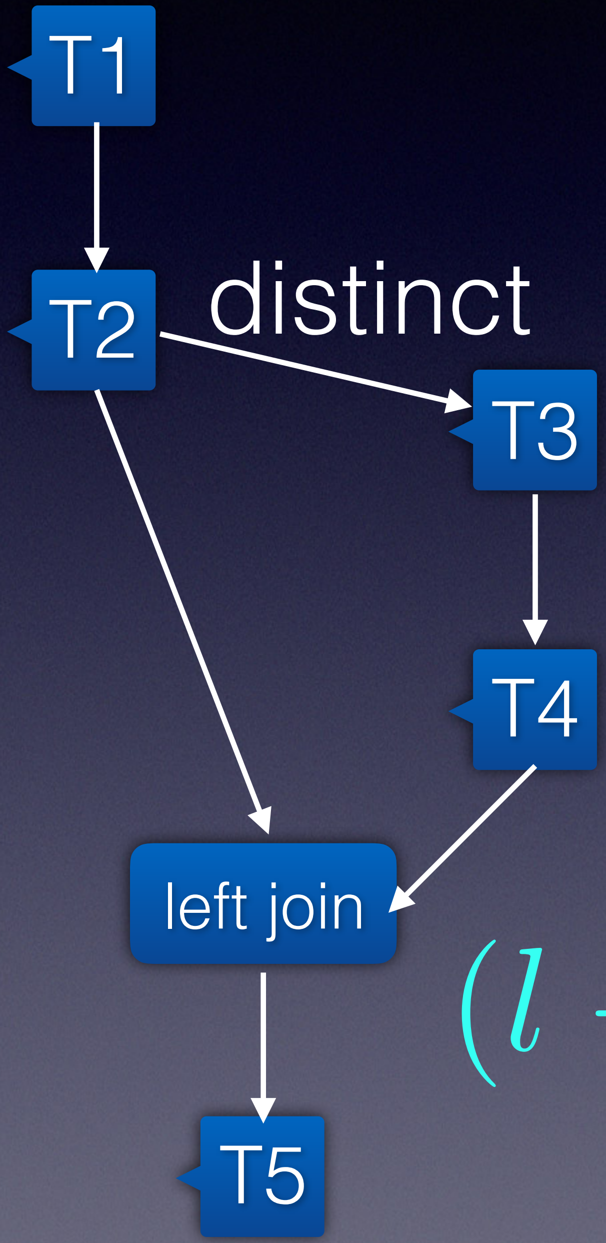
Available:
Live: ?str ?y

- ?str, ?y projected, so they are live
- minus cannot done, since ?x not available - ?x is the only variable the RHS shares with the LHS
- t6, t2, t1 must be after minus because they all make ?str, ?y available
- must use less-efficient join

Optional Example

```
select ?r ?str {
   { ?x :p ?y  T1
     ?y a Bar  T2
   } OPTIONAL {
     ?x :q ?y  T3
     ?y :r ?r  T4
   }
   ?x :name ?str  T5
}
```

Optional Compilation



Output of the left join of *l* and *r* produces the semantics of the optional in SPARQL

$$(l - r) \cup (l \cap r)$$

References

- SQLGraph: An Efficient Relational-Based Property Graph Store (SIGMOD 2015)
- An Offline Optimal SPARQL Query Planning Approach to Evaluate Online Heuristic Planners. (WISE 2014)
- Building an efficient RDF store over a relational database (SIGMOD 2013).