Replacing GEQO

Join ordering via Simulated Annealing

Jan Urbański
j.urbanski@wulczer.org

University of Warsaw / Flumotion

May 21, 2010

For those following at home

Getting the slides

```
$ wget http://wulczer.org/saio.pdf
```

Trying it out

```
$ git clone git://wulczer.org/saio.git
$ cd saio && make
$ psql
=# load '.../saio.so';
```

Do not try to compile against an assert-enabled build. Do not be scared by lots of ugly code.

Outline

Getting the optimal join order

- part of of planning a query is determining the order in which relations are joined
- it is not unusual to have queries that join lots of relations
- JOIN and subquery flattening contributes to the number or relations to join
- automatically generated queries can involve very large joins

Problems with join ordering

- finding the optimal join order is an NP-hard problem
- considering all possible ways to do a join can exhaust available memory
- not all join orders are valid, because of:
  - outer joins enforcing a certain join order
  - IN and EXISTS clauses that get converted to joins
- joins with restriction clauses are preferable to Cartesian joins

Outline

Randomisation helps

- PostgreSQL switches from exhaustive search to a randomised algorithm after a certain limit
- GEQO starts by joining the relations in any order
- and then proceeds to randomly change the join order
- genetic algorithm techniques are used to choose the cheapest join order

Problems with GEQO

- has lots of dead/experimental code
- there is a TODO item to remove it
- nobody really cares about it
- is an adaptation of an algorithm to solve TSP, not necessarily best suited to join ordering
- requires some cooperation from the planner, which violates abstractions

Outline

Previous work

- numerous papers on optimising join order have been written
- Adriano Lange implemented a prototype using a variation of Simulated Annealing
- other people discussed the issue on `-hackers`

What is Simulated Annealing

Annealing (...) is a process that produces conditions by heating to above the re-crystallisation temperature and maintaining a suitable temperature, and then cooling.

– *Wikipedia*

The SA Algorithm cont.

- the system starts with an initial temperature and a random state
- uphill moves are accepted with probability that depends on the current temperature

probability of accepting an uphill move

$$p = e^{\frac{cost_{prev} - cost_{new}}{temperature}}$$

- moves are made until equilibrium is reached
- temperature is gradually lowered
- once the system is frozen, the algorithm ends

The SA algorithm

Simulated Annealing

The SA algorithm

Simulated Annealing

```
state = random_state()
```

The SA algorithm

Simulated Annealing

```
state = random_state()


      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
```

The SA algorithm

Simulated Annealing

```
state = random_state()

    do {
      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
    }
    while (!equilibrium())
```

The SA algorithm

Simulated Annealing

```
state = random_state()

    do {
      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
    }
    while (!equilibrium())
    reduce_temperature()
```

The SA algorithm

Simulated Annealing

```
state = random_state()
do {
    do {
      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
    }
    while (!equilibrium())
    reduce_temperature()
}
while (!frozen())
return state
```

The SA Algorithm cont.

Implementing Simulated Annealing means solving the following problems:

- finding an initial state
- generating subsequent states
- defining an acceptance function
- determining the equilibrium condition
- suitably lowering the temperature
- determining the freeze conditions

The SA algorithm

Simulated Annealing

```
state = random_state()
do {
    do {
      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
    }
    while (!equilibrium())
    reduce_temperature()
}
while (!frozen())
return state
```

The SA algorithm

Simulated Annealing

```
state = random_state()
do {
    do {
      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
    }
    while (!equilibrium())
    reduce_temperature()
}
while (!frozen())
return state
```

The SA algorithm

Simulated Annealing

```
state = random_state()
do {
    do {
      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
    }
    while (!equilibrium())
    reduce_temperature()
}
while (!frozen())
return state
```

The SA algorithm

Simulated Annealing

```
state = random_state()
do {
    do {
      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
    }
    while (!equilibrium())
    reduce_temperature()
}
while (!frozen())
return state
```

The SA algorithm

Simulated Annealing

```
state = random_state()
do {
    do {
      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
    }
    while (!equilibrium())
    reduce_temperature()
}
while (!frozen())
return state
```

The SA algorithm

Simulated Annealing

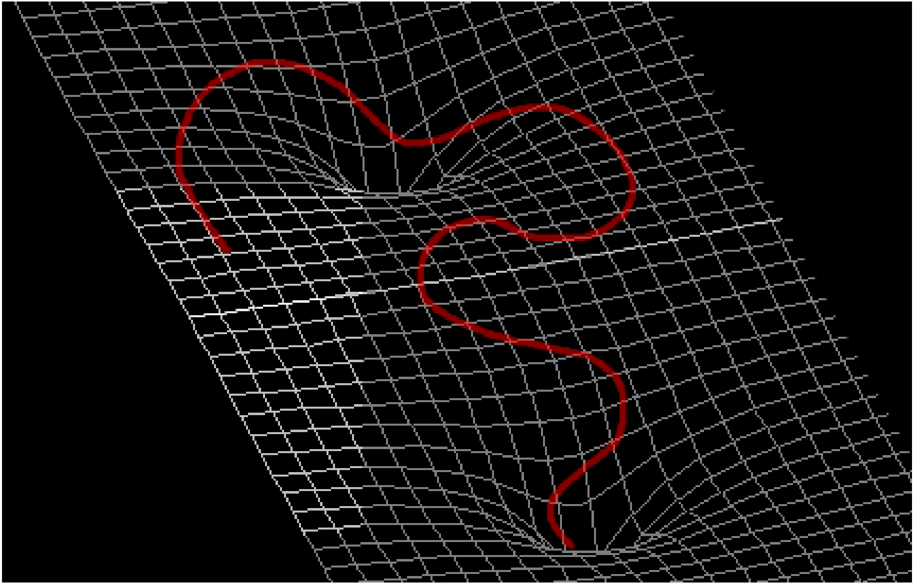
```
state = random_state()
do {
    do {
      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
    }
    while (!equilibrium())
    reduce_temperature()
}
while (!frozen())
return state
```

The SA algorithm

Simulated Annealing

```
state = random_state()
do {
    do {
      new_state = random_move()
      if (acceptable(new_state))
        state = new_state
    }
    while (!equilibrium())
    reduce_temperature()
}
while (!frozen())
return state
```

A visual example



Outline

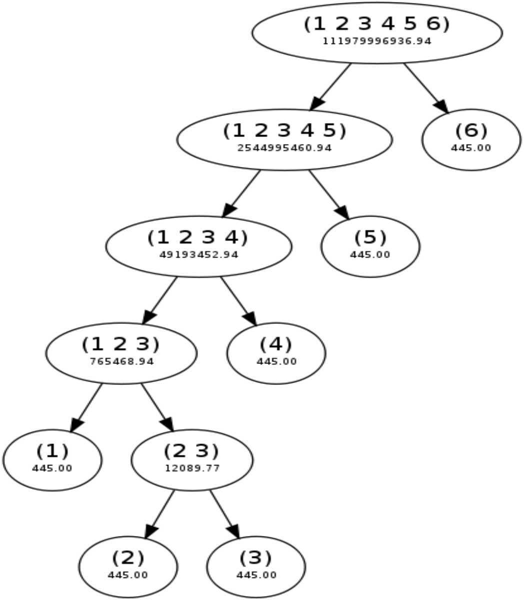
Differences from the original algorithm

- PostgreSQL always considers all possible paths for a relation
- `make_join_rel` is symmetrical
- you can have join order constraints (duh)
- the planner is keeping a list of all relations...
- ... and sometimes turns it into a hash

Join order representation

- SAIO represents joins as query trees
- chosen to mimic the original algorithm more closely
- each state is a query tree
- leaves are basic relations
- internal nodes are joinrels
- the joinrel in the root of the tree is the current result

Query trees

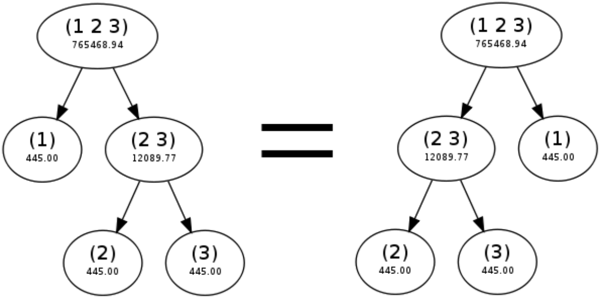


Example query tree for a six relation join.

Query trees cont.

Some useful query tree properties:

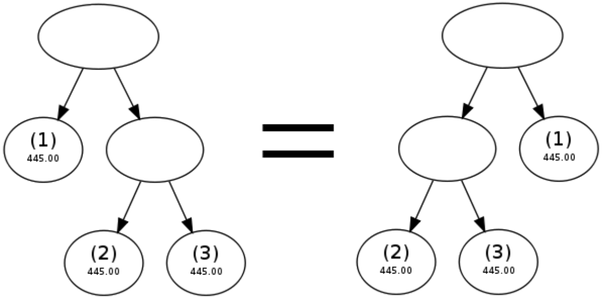
- symmetrical (no difference between left and right child)



Query trees cont.

Some useful query tree properties:

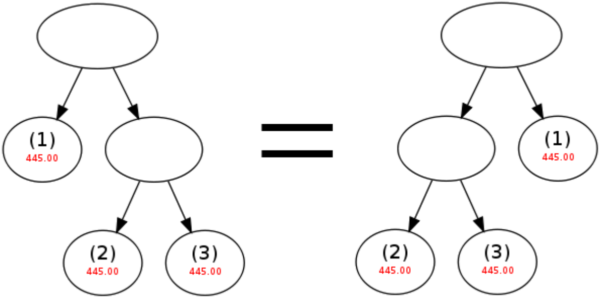
- symmetrical (no difference between left and right child)
- fully determined by the tree structure and relations in leaves



Query trees cont.

Some useful query tree properties:

- symmetrical (no difference between left and right child)
- fully determined by the tree structure and relations in leaves
- each node has a cost



Outline

Implementing Simulated Annealing means solving the following problems:

- finding an initial state
- generating subsequent states
- defining an acceptance function
- determining the equilibrium condition
- suitably lowering the temperature
- determining the freeze conditions

Implementing Simulated Annealing means solving the following problems:

- finding an initial state
- generating subsequent states
- defining an acceptance function
- determining the equilibrium condition
- suitably lowering the temperature
- determining the freeze conditions

Some are easy

Implementing Simulated Annealing means solving the following problems:

- finding an initial state
- generating subsequent states
- defining an acceptance function
- determining the equilibrium condition
- suitably lowering the temperature
- determining the freeze conditions

Some are easy, some are hard

The easy problems - initial state

Finding an initial state

Make base relations into one-node trees, keep merging them on joins with restriction clauses, forcefully merge the remaining ones using Cartesian joins. Results in a query tree that is as left-deep as possible.

This is exactly what GEQO does.

The easy problems - temperature

The acceptance function

A uphill move is accepted with the probability that depends on the current temperature.

$$P(accepted) = e^{\frac{cost_{prev} - cost_{new}}{temperature}}$$

Lowering the temperature

The initial temperature depends on the number of initial relations and drops geometrically.

$$initial\_temperature = I * initial\_rels$$

$$new\_temperature = temperature * K$$

where

$$0 < K < 1$$

The easy problems - equilibrium and freezing

Equilibrium condition

Equilibrium is reached after a fixed number of moves that depend on the number of initial relations.

$$moves\_to\_equilibrium = N * initial\_rels$$

Freezing condition

The system freezes if temperature falls below 1 and a fixed number of consecutive moves has failed.

Move types

The difficult part seems to be generating subsequent states.

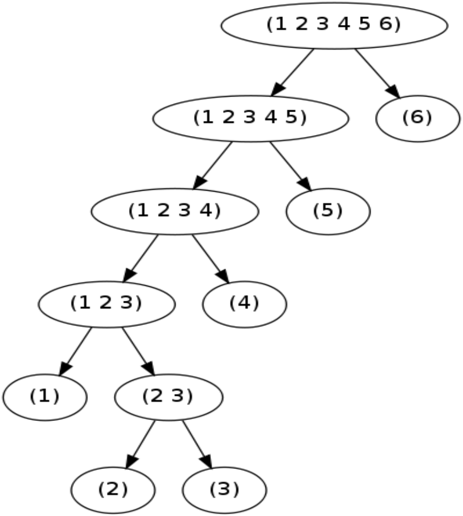
- a number of move generating approaches can be taken
- the most costly operations is creating a joinrel, especially computing paths
- need to free memory between steps, otherwise risk overrunning
- need to deal with planner scribbling on its structures when creating joinrels
- how to efficiently sample the solution space?

SAIO `move`

- randomly choose two nodes from the query tree
- swap the subtrees around
- recalculate the whole query tree
- if it cannot be done, the move fails
- check if the cost of the new tree is acceptable
- if not, the move fails

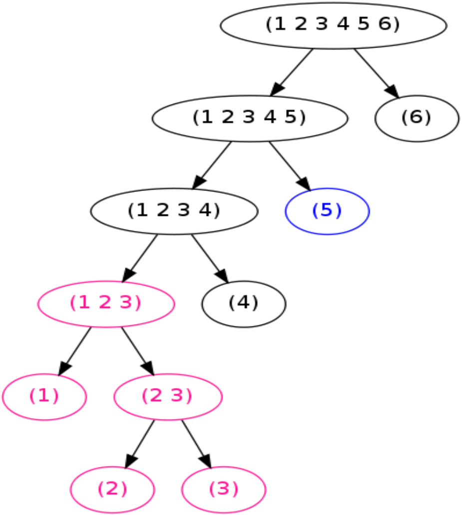
SAIO move example

Take a tree,



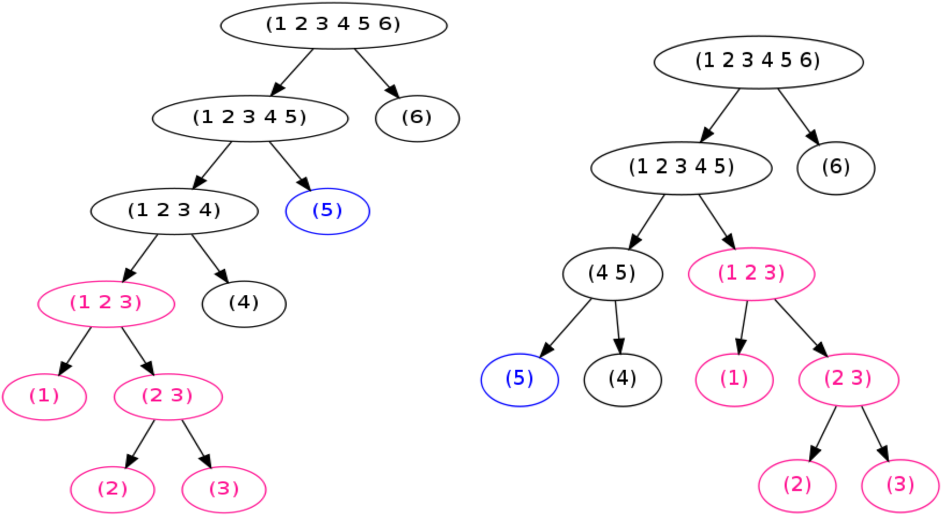
SAIO move example

Take a tree, choose two nodes,



SAIO move example

Take a tree, choose two nodes, swap them around



SAIO `move` problems

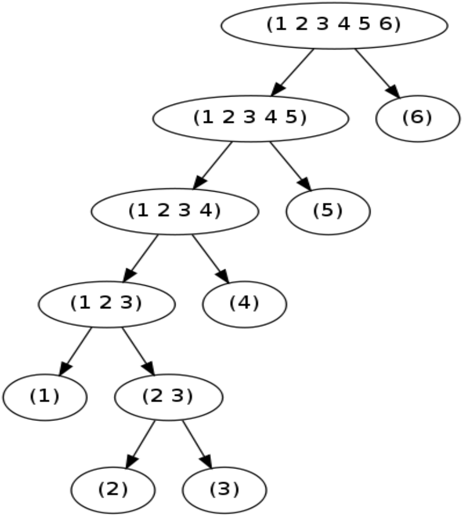
- choosing a node narrows down the possible choices for the second node
  - can't choose descendant node (how would that work?)
  - can't choose ancestor node (for the same reason)
  - can't choose sibling node (because of symmetry)
- the changes to the tree are big, the algorithm takes "large steps"
- if the resulting query tree is invalid, lots of work is thrown away
- any join failure results in the whole move failing, so it doesn't explore the solution space very deeply

SAIO pivot

- change `(A join B) join C` into `A join (B join C)`
- in practise, choose a node at random
- swap the subtrees of one of its children and the sibling's
- continue trying such pivots until all nodes have been tried

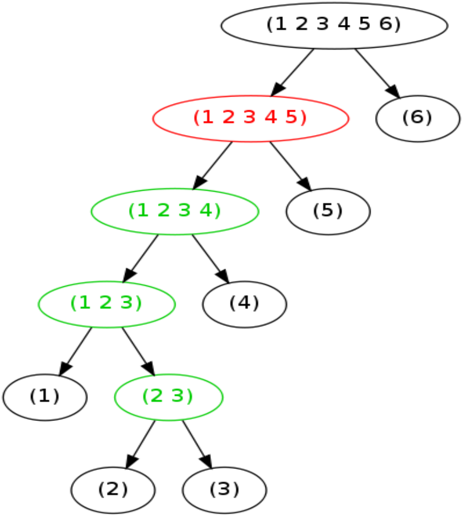
SAIO pivot example

Take a tree,



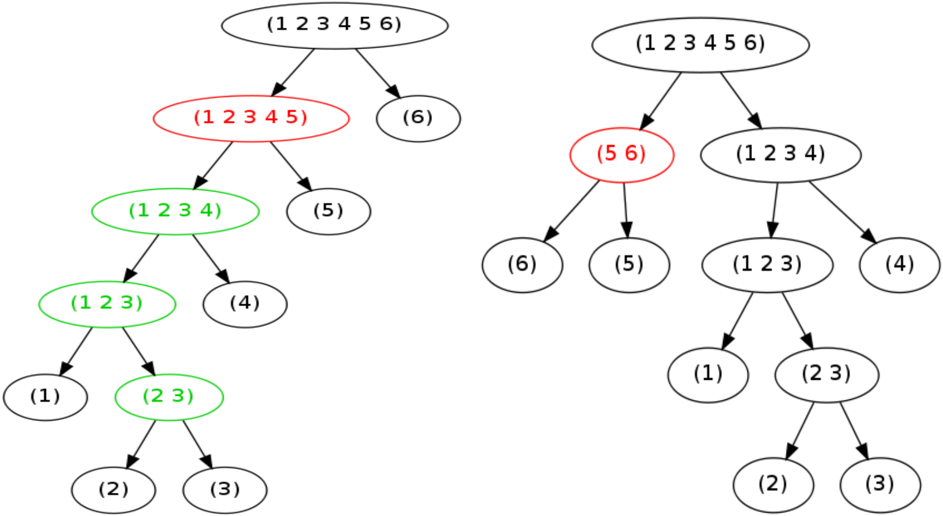
SAIO pivot example

Take a tree, choose a node,



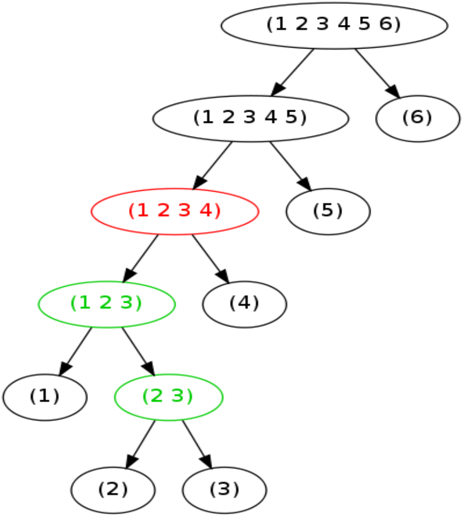
SAIO pivot example

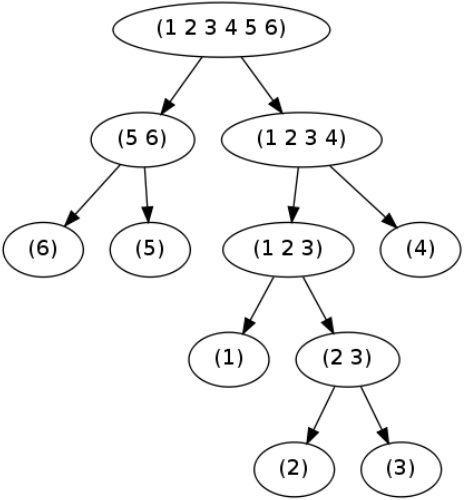
Take a tree, choose a node, pivot,



SAIO pivot example

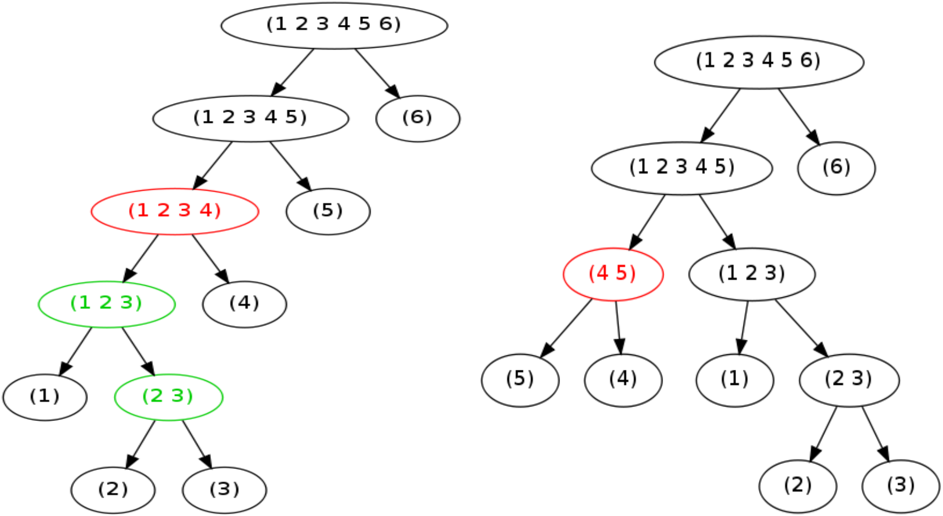
Take a tree, choose a node, pivot, choose another,





SAIO pivot example

Take a tree, choose a node, pivot, choose another, pivot ...



SAIO `pivot` problems

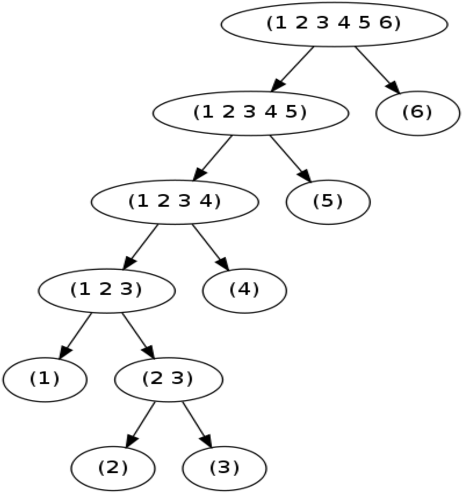
- each move explores a lot of possibilities, but requires lots of computation
- does not introduce big changes, which sometimes are needed to break pessimal joins
  - actually, it's not obvious that the solution space is smooth wrt costs
  - small changes in the structure may result it gigantic changes in costs
  - might want to augment the cost assessment function (number of non-cross joins?)
- the same join might be recalculated many times in each step

SAIO `recalc`

- essentially the same as `move`
- but keeps all the relations built between moves
- recalculate the joins from the chosen nodes up to the common ancestor
- if it succeeded, recalculate the nodes from the common ancestor up to the root node
- avoids pointless recalculations when joins fail
- each query tree node has its own memory context
- needs to remove individual joinrels from the planner (nasty!)

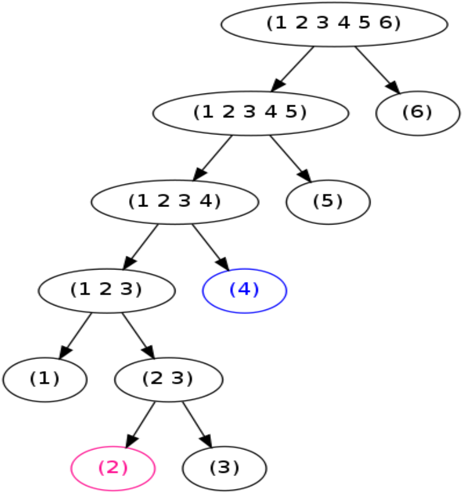
SAIO `recalc` example

Take a tree,



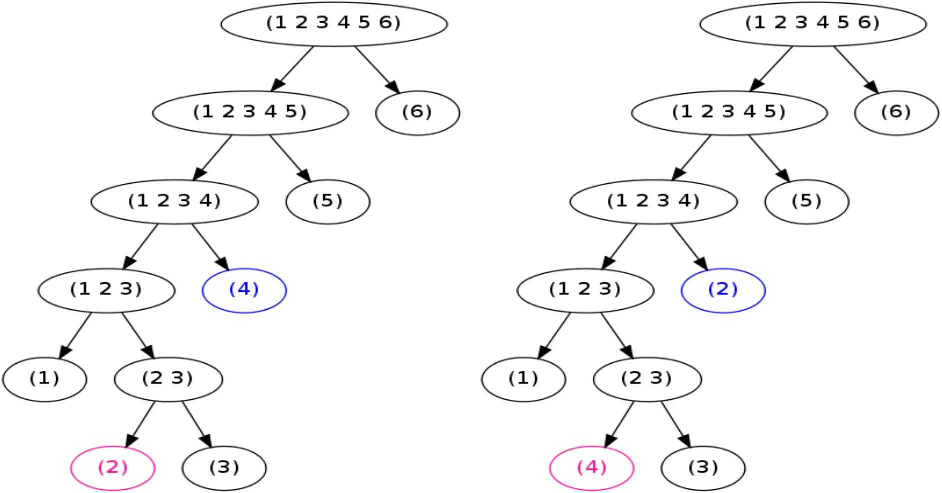
SAIO `recalc` example

Take a tree, choose two nodes,



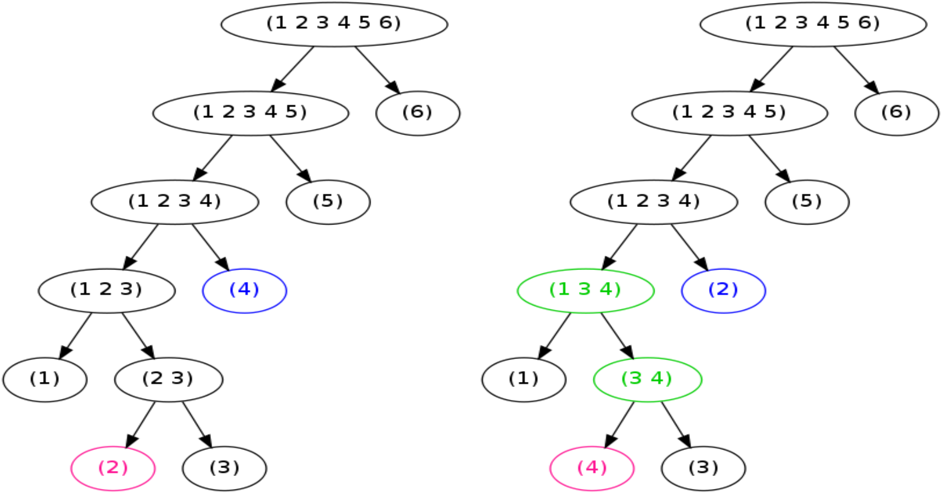
SAIO recalc example

Take a tree, choose two nodes, swap them around,



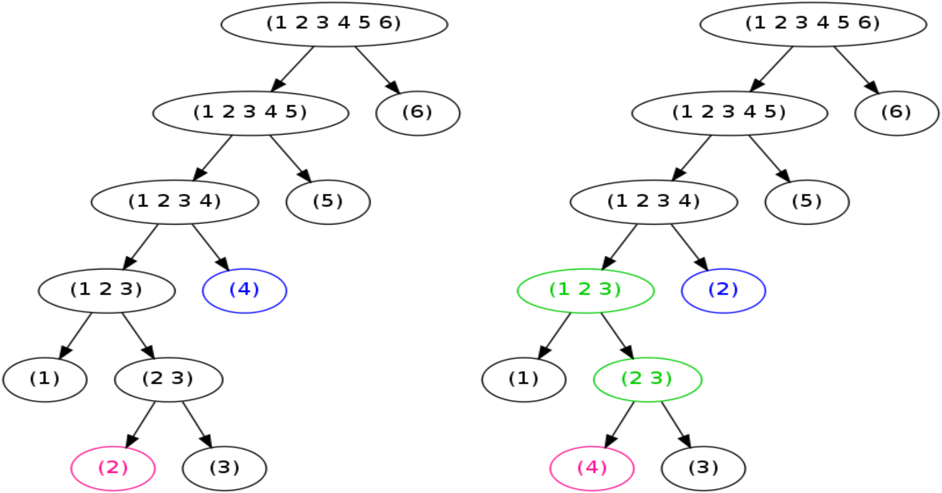
SAIO `recalc` example

Take a tree, choose two nodes, swap them around, recalculate up to the common ancestor,



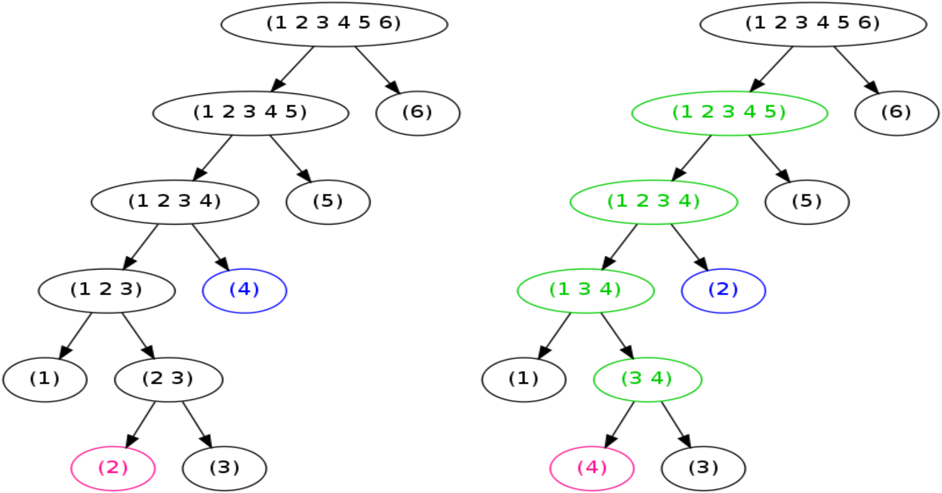
SAIO `recalc` example

Take a tree, choose two nodes, swap them around, recalculate up to the common ancestor, just discard the built joinrels if failure,



SAIO `recalc` example

Take a tree, choose two nodes, swap them around, recalculate up to the common ancestor, just discard the built joinrels if failure, recalculate the rest of the relations



SAIO `recalc` problems

- does really nasty hacks
- does not speed things up as much as it should
- does not solve the problem of failing, it just makes failures cheaper
- because the trees are usually left-deep, the benefits from partial recalculation are not as big

Outline

Moderately big query

Collapse limits set to 100. Move algorithm used is `recalc`.

algorithm	equilibrium loops	temp reduction	avg cost	avg time
GEQO	n/a	n/a	1601.540000	0.54379
SAIO	4	0.6	1623.874000	0.42599
SAIO	6	0.9	1617.341000	3.69639
SAIO	8	0.7	1618.838000	1.44605
SAIO	12	0.4	1627.873000	0.87609

Very big query

Collapse limits set to 100. Move algorithm used is `recalc`.

algorithm	equilibrium loops	temp reduction	avg cost	avg time
GEQO	n/a	n/a	22417.210000	777.20033
SAIO	4	0.6	21130.063000	145.27570
SAIO	6	0.4	21218.134000	125.08545
SAIO	6	0.6	21131.333000	240.70522
SAIO	8	0.4	21250.160000	179.34261

Outline

What the future brings

- MSc thesis :o)
- smarter tree transformation methods
- less useless computation
- perhaps some support from the core infrastructure
- faster and higher quality results than GEQO
- `git://wulczer.org/saio.git`

Acknowledgements

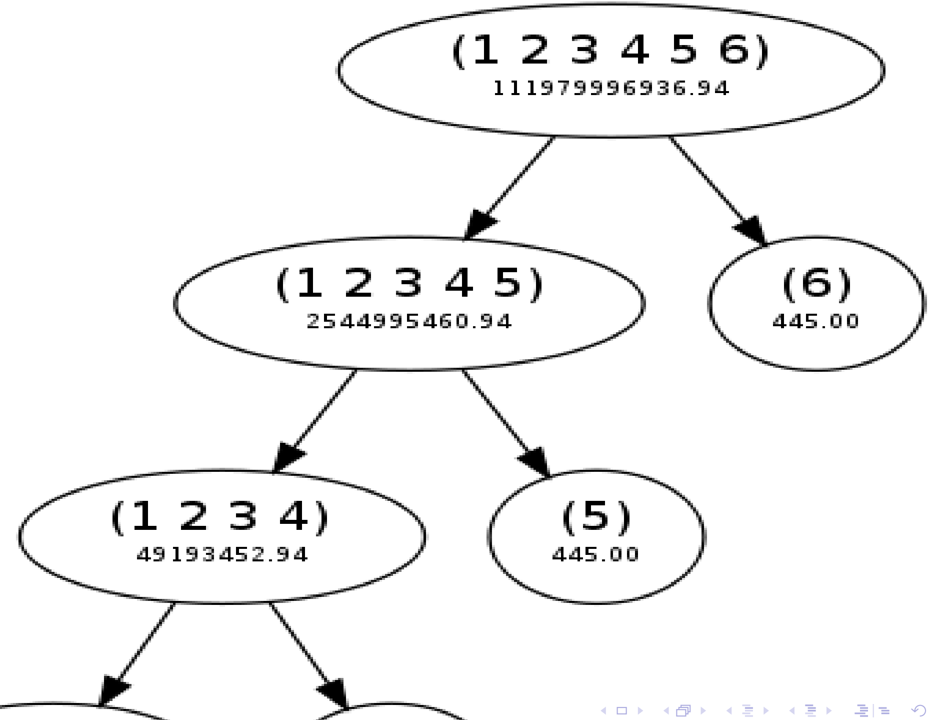
- Adriano Lange, the author of the TWOPO implementation
- Andres Freund, for providing the craziest test query ever
- Robert Haas, for providing a slightly less crazy test query
- dr Krzysztof Stencel, for help and guidance
- Flumotion Services, for letting me mess with the PG planner instead of doing work

Further reading

Yannis E. Ioannidis and Eugene Wong.
Query optimization by simulated annealing.
SIGMOD Rec., 16(3):9–22, 1987.

Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper.
Heuristic and randomized optimization for the join ordering problem.
The VLDB Journal, 6(3):191–208, 1997.

Questions?


(1 2 3 4 5 6)
111979996936.94
(1 2 3 4 5)
2544995460.94
(6)
445.00
(1 2 3 4)
49193452.94
(5)
445.00