# Estimating query progress

## Theory and practice of query progress indication

Jan Urbański
j.urbanski@wulczer.org

Ducksboard

PGCon 2013, Ottawa, May 24

## For those following at home

### Getting the slides

```
$ wget http://wulczer.org/pg-progress.pdf
```

### Trying the code*

```
$ git clone git://github.com/wulczer/pg-progress.git
$ make install && psql -c 'create extension progress'
$ ./show-progress.py -c 'table tab1' dbname=progress
```

* requires modified Postgres from git://github.com/wulczer/postgres/tree/progress-rebasing

## Outline

Should I kill it yet?

## Usefulness of progress estimation

- Postgres gives no feedback for running queries
- it's hard to decide how much more time will a query take
- the decision between cancelling a heavyweight query and letting it complete is uninformed

## The devil is in the details

- while the use case is straightforward, the details are not
- what should be an estimator's output?
  - "work done"
  - percentage completed
  - time remaining
- how should the estimation be relayed to the outside
  - the backend running the query is busy... running the query

## Outline

## What makes a good estimator

| | |
|---|---|
| monotonicity | the estimator's value should always grow |
| granularity | estimates should vary across small time increments |
| low overhead | calculating progress should not be prohibitively expensive |
| hardware independence | changes in the estimates should not depend on how fast the system is |

## Outline

## Concepts inside the executor

- the executor works with a tree structure
- the executor tree mirrors the plan tree node for node
- each node has a method to produce a tuple or NULL if it's finished
- executing the plan is repeatedly fetching a tuple from the root node

## Concepts inside the executor

- the executor works with a tree structure
- the executor tree mirrors the plan tree node for node
- each node has a method to produce a tuple or NULL if it's finished
- executing the plan is repeatedly fetching a tuple from the root node

### Executing a query

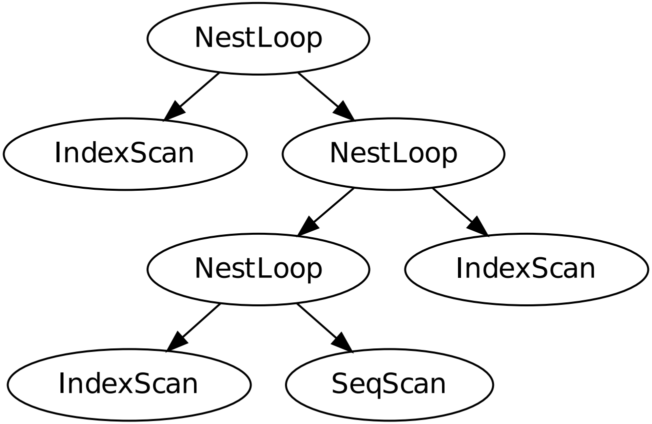
```
while (true) {
    tup = ExecProcNode(root);
    if (tup == NULL)
        break;
    emit(tup);
}
```

## Individual node execution

- each node has its own execution method
  - for Sequential Scans it fetches the next tuple from the heap
  - for Materialization it fetches and stores the next tuple from its child node
  - for Sort it stores all the tuples from its child node, sorts them and returns the top one
- nodes can carry additional instrumentation, that gets updated when they are executed
- each executor node has a link to the correspoding planner node, where estimates are kept

## Executor tree example

An example executor tree looks like this:



## Existing research

The basic idea explained here comes from a research paper by Surajit Chaudhuri and Vivek Narasayya from Microsoft Research and Ravi Ramamurthy from the University of Wisconsin.

Chaudhuri, Surajit and Narasayya, Vivek and Ramamurthy, Ravishankar.
Estimating progress of SQL queries.
*SIGMOD '04*, 803–814, 2004.

## Model of work

- query progress is expressed as the number tuples returned from all execution nodes
- that's called the GetNext() model in the paper
- a query is completed when all nodes have finished returning tuples

## Estimator definition

### GetNext() model estimator

Assume the execution plan has $m$ nodes.

For $i = 1..m$, we define $N_i$ as the number of tuples that it will return and $K_i$ as the number of tuples it has already returned.

The estimator we'll use is

$$gnm = \frac{sum\ of\ all\ tuples\ returned}{sum\ of\ all\ tuples\ to\ be\ returned}$$

## Estimator definition

### GetNext() model estimator

Assume the execution plan has $m$ nodes.

For $i = 1..m$, we define $N_i$ as the number of tuples that it will return and $K_i$ as the number of tuples it has already returned.

The estimator we'll use is

$$gnm = \frac{sum\ of\ all\ tuples\ returned}{sum\ of\ all\ tuples\ to\ be\ returned}$$

## Estimator definition

### GetNext() model estimator

Assume the execution plan has $m$ nodes.

For $i = 1..m$, we define $N_i$ as the number of tuples that it will return and $K_i$ as the number of tuples it has already returned.

The estimator we'll use is

$$gnm = \frac{sum\ of\ all\ tuples\ returned}{sum\ of\ all\ tuples\ to\ be\ returned}$$

## Estimator definition

### GetNext() model estimator

Assume the execution plan has $m$ nodes.

For $i = 1..m$, we define $N_i$ as the number of tuples that it will return and $K_i$ as the number of tuples it has already returned.

The estimator we'll use is

$$gnm = \frac{sum\ of\ all\ tuples\ returned}{sum\ of\ all\ tuples\ to\ be\ returned}$$

The basic challenge is estimating $N_i$.

## Properties of *gnm*

- kind of, sort of computable in Postgres :)
- reasonably straightforward definition
- monotonous (usually), fine-grained
- can work off planner estimates, most of the necessary info is already being collected
- but planner estimates can be wrong, especially with joins present

## Outline

## The pipeline concept

**Pipeline definition**

An execution pipeline is the maximal executor subtree that executes concurrently.

You can think of a pipeline as the set of executor nodes that will be run for the pipeline's root node to produce one tuple.

## Constructing pipelines

- execution plan leaves start new pipelines
- nodes like Sort or Hash start new pipelines
- joins combine the pipelines of their children

## Driver nodes

### Driver node definition

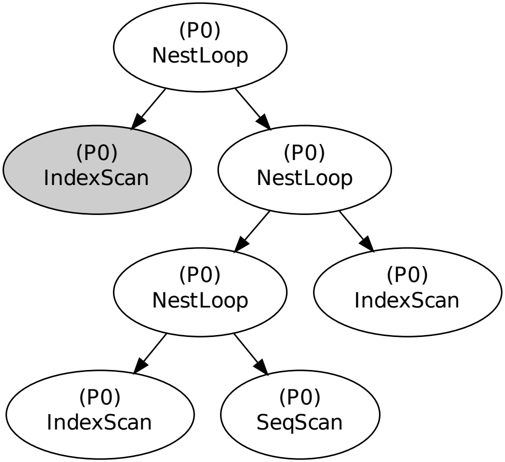
Driver nodes are leaf nodes of pipelines, except for nodes in the inner subtree of a Nested Loops join.

Conceptually, driver nodes are the ones that drive the pipeline, supplying it with tuples.

Because of that, they are better suited to estimate the progress of a pipeline than the rest of its nodes.

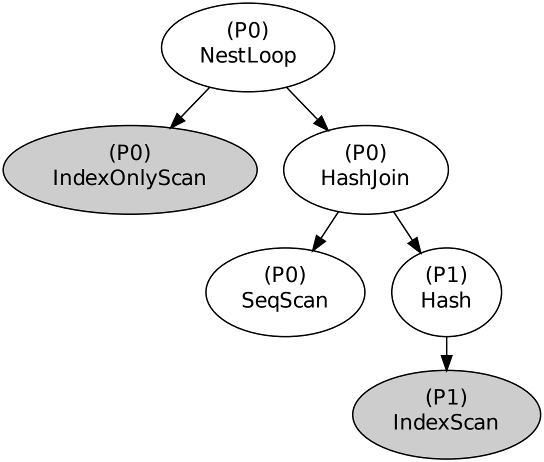
## Pipeline examples

The same executor tree annotated with pipeline identifiers:



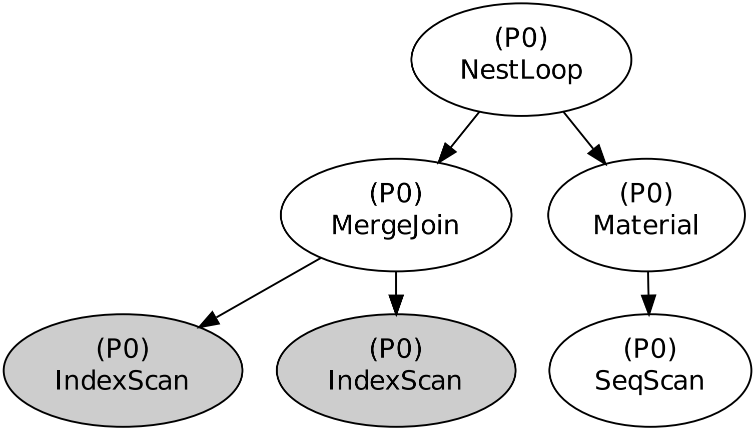
## Pipeline examples cont.

A plan with several pipelines:



## Pipeline examples cont.

Another example:



## Outline

## Driver node hypothesis

**Estimating progress of a running pipeline**

For pipelines with one driver node that are in progress, we use the driver node hypothesis, which says that

$$\frac{\textit{tuples a pipeline has returned}}{\textit{tuples to be returned}} \approx \frac{\textit{tuples the driver node has returned}}{\textit{tuples the driver node will return}}$$

## Unstarted and finished pipelines

### Estimating progress of other pipelines

For pipelines that are already finished, we know

$$\textit{tuples returned} = \textit{tuples to be returned}$$

For pipelines that are not yet started, we use

$$\textit{tuples returned} = 0, \quad \textit{tuples to be returned} = \textit{planned estimate}$$

## Final estimator

### GetNext() estimator with pipelines

For a query comprising $s$ pipelines, the final estimator definition is

$$gnm = \frac{sum\ of\ \ tuples\ already\ returned\ for\ each\ pipeline}{sum\ of\ \ tuples\ to\ be\ returned\ for\ each\ pipeline}$$

$$gnm = \frac{\sum\limits_{i \in P1} K_i + ... + \sum\limits_{i \in Ps} K_i}{\sum\limits_{i \in P1} N_i + ... + \sum\limits_{i \in Ps} N_i}$$

## Outline

## Implementation idea

The idea was to make the implementation minimally intrusive with regards to Postgres code.

1. at executor startup, determine pipelines and driver nodes
2. when asked to estimate the progress
    1. walk all pipelines
    2. calculate $K_P$ and $N_P$ for each of them
3. return the sum of $K_P$ divided by the sum of $N_P$
4. use the opportunity to draw a graph of the executor tree and annotate it with progress information for each node

## Calculating pipeline states

### Processing a pipeline

```
k, n = 0;
for node in pipeline {
    k += tuples_processed(node);
    n += tuples_estimated(node);
}
```

## Calculating pipeline states

### Processing a pipeline

```
k, n = 0;
for node in pipeline {
    k += tuples_processed(node);
    n += tuples_estimated(node);
}
if (all(pipeline->driver_nodes, FINISHED))
    return k, k;
```

## Calculating pipeline states

### Processing a pipeline

```
k, n = 0;
for node in pipeline {
    k += tuples_processed(node);
    n += tuples_estimated(node);
}
if (all(pipeline->driver_nodes, FINISHED))
    return k, k;
else if (any(pipeline->driver_nodes, STARTED))
    return k, k * dne(pipeline);
```

## Calculating pipeline states

### Processing a pipeline

```
k, n = 0;
for node in pipeline {
    k += tuples_processed(node);
    n += tuples_estimated(node);
}
if (all(pipeline->driver_nodes, FINISHED))
    return k, k;
else if (any(pipeline->driver_nodes, STARTED))
    return k, k * dne(pipeline);
else
    return 0, n;
```

## Communicating with the query backend

- needed a way to get information from a backend running the query
- not trivial, since the backend is busy producing the result
- introduce a custom signal handler based on a patch from Simon Riggs
- on next CHECK_FOR_INTERRUPTS(), calculate progress and put in shared memory
- signalling backend can then grab results from shmem

## Changes to core Postgres

From least to most controversial:

- bulk of the logic is implemented as a Postgres extension
- ability to store private data in the main executor structure
- hooks for executor node instrumentation
- a hook for running user code on SIGUSR1

## Outline

## Not everything went as smoothly...

- multiple driver nodes in a pipeline
- nodes being executed multiple times
- node with more than two children (e.g. Append)
- subselect, function scans, recursive queries...
- poor planner estimates

## Some solutions

- for a pipeline with $D$ driver nodes, the estimator is using

$$K = \textit{average amount of tuples returned by driver nodes}$$
$$N = \textit{minimum estimate of tuples to be returned}$$

- for Nested Loops, the estimated tuple counts of inner child nodes are multiplied by estimated outer child tuple count
- nodes with more than 2 children are simply assumed blocking and start their own pipeline
- some nodes are not supported at all (for instance Recursive Union)

## Refining estimates

- the original paper puts a lot of emphasis on refining estimates as the query progresses
- each node has a lower and an upper bound for its *N* value
- those bounds are refined as execution progresses:
    - Sort or Hash nodes will never increase carditality
    - when an outer child of a Nested Loops join outputs more tuples than expected, recalculate the expected number of loops in the inner subtree
    - Merge joins will stop as soon as they finish processing one of their inputs
- a missing feature for now

## Outline

## Annoyances

- correctly identifying pipelines and driver nodes (lots of different nodes types to think about)
- detecting if a pipeline has started or has finished
  - annoyingly tricky
  - nodes can be executed several times, for example in a Nested Loops outer tree
- no visibility into a Sort node progress
  - important, because Sort nodes are driver nodes
- using shared memory to relay estimates to other backends
  - haven't even considered the security implications yet...

## Rescanning nodes

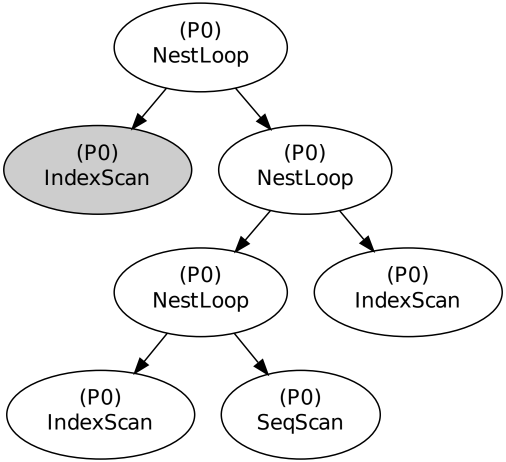
- the executor has a concept of rescanning a node
- some nodes are very cheap to rescan, for example Materialization nodes
- they will show a large number for tuples returned, but the real work was just scanning the child node once
- need to consider number of loops manually, since the planner accounts for them internally and does not expose the total estimated number of tuples
- not acconuted for in the original paper at all

## The “single outer tuple” problem

- a common situation is a Nested Loops scan estimating a single outer tuple
- if the outer child returns more than one tuple, the entire inner subtree is rescanned
- arguably a planner failure, but because the outer child can end up being the only driver node of a large pipeline, estimates are all wrong

## “Single output tuple” example

This tree has only one driver node, which leads to imprecision



## MultiExecProcNode

- some nodes are "special" and don't follow the *GetNext*() model
  - Hash nodes build the entire hashtable in one go
  - Bitmap index scans scan the entire index when building the bitmap
- if they'd be just blocking, it wouldn't have been a problem
- being outside of *GetNext*() means no visibility into their progress
- particularily bad when they're driver nodes

## Utility commands

- this model does not allow estimating the progress of utility commands in any way
- things like COPY, VACUUM, CREATE INDEX, ANALYSE
- a shame, since these are just the kinds of statements that often would benefit from estimation
- possibly a hybrid approach would work, special-casing utility statements

## Current state

- requires small changes to core Postgres, but some are controversial
- the driver node estimator looks like an interesting metric to use
- could be reimplemented as returning just raw execution data and let other tools calculate a single progress value
- graphical dumps are more a debugging tool, should probably use EXPLAIN-ish format and external tools could transform it into images
- need to implement estimate refining
- for the time being, a proof of concept

Questions?