

COMP2400

Relational Databases

Lecture 32: Query Optimisation

Ben Lippmeier
Australian National University
Semester 2
2008

What costs to consider?

- Access to the disk
 - reading indexes and tables.
 - reading and writing temporary files.
 - sequential access is faster than random access.
- Storage cost for temporary files
 - if we need to use an external sort we must enough temp space available.
- Computation time
 - searching indexes
 - in memory sorting, merging
- Memory usage
 - in memory sort works best when the whole table is loaded...

Query Optimisation

- We have seen how to derive a query plan from an SQL expression.
- For a given query there are usually many possible plans which return the same result, but differ in resource cost.
- The query optimiser considers several possible plans, and tries to identify the one which minimizes the cost.
- The number of plans grows exponentially in the size of the query.
- For a small query it may be possible to exhaustively consider all possible plans. For large queries we must make do with a “best guess”

PostgreSQL cost constants

constant	estimated ...	default value
seq_page_cost	cost to accessing a disk page sequentially	1
random_page_cost	cost to access a random page	4
cpu_tuple_cost	cost to process a row during a query	0.01
cpu_index_tuple_cost	cost to process an index entry	0.005
cpu_operator_cost	cost to process an operator or function	0.0025
effective_cache_size	estimated size of kernel disk cache	128MB

- Cost constants give an overall weighting for each of the resources used by the DBMS.
- The constants are normalised against the average cost to sequentially read a single disk page.

Nested Loop Join Cost $R \bowtie_c S$

- Computation cost

$$|R| * |S| \text{ key comparisons}$$

- Block Reads

$$|R| / \text{bfr}_R + |S| / \text{bfr}_S$$

- Memory Cost

$$|R| * \text{rs}_R + |S| * \text{rs}_S$$

- This assumes both tables will fit in memory...

bfr = blocking factor in (blocks/row)

rs = row size in (bytes /row)

Checking the size of your tables

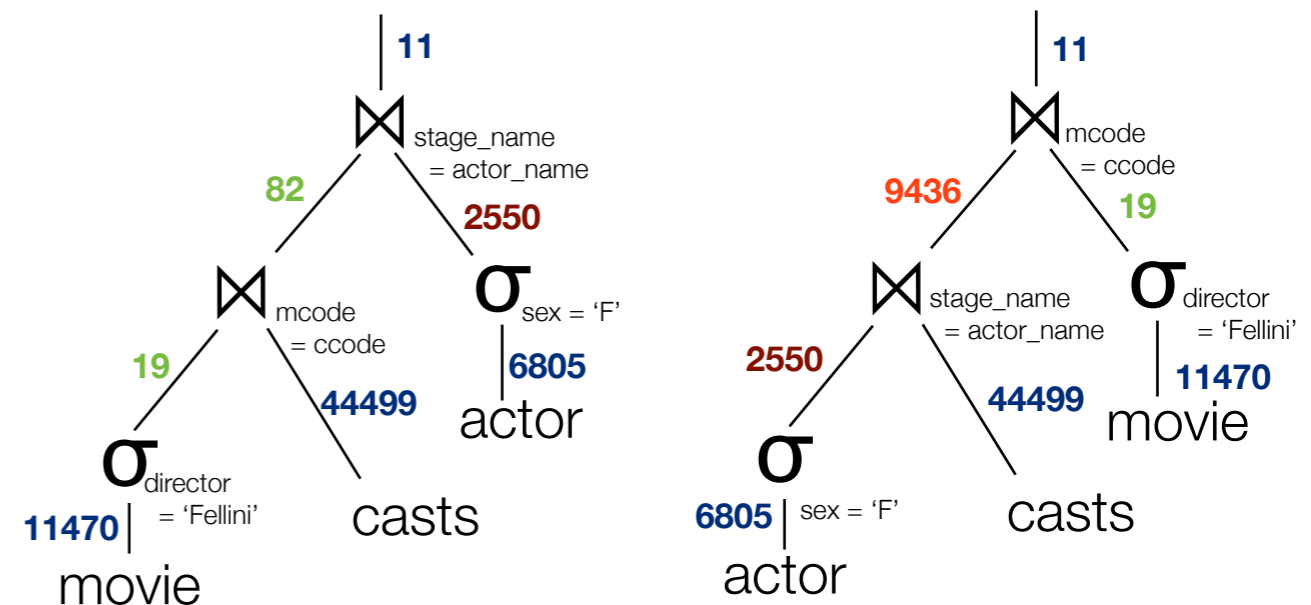
```
SELECT relname, reltuples, relpages
FROM pg_class
WHERE relname
IN ( 'movie', 'casts', 'actor'
, 'casts_code', 'movie_director', 'actor_sex');
```

relname	reltuples	relpages
actor	6805	69
casts	44499	346
movie	11471	361
casts_code	44499	122
movie_director	11471	37
actor_sex	6805	17

Merge Join Cost

- Are we doing an internal or external join?
- Are the columns already sorted on the join attribute, do we need to sort the rows, or are we doing an index scan instead?
- If we have to write temporary files, how big will they be?
- How much of the data is already in cache?
- Be wary of using a moderately complex equation to model a fiendishly complex system. We can't hope to model all the parameters accurately.
- It's probably better to use a simple cost metric, or extract a cost equation from real test data..

Intermediate results. $(M \bowtie C) \bowtie A \equiv (A \bowtie C) \bowtie M$

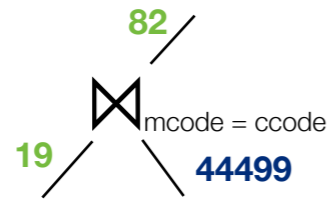


- Both of these plans yield the same result, but the second one creates an intermediate table of much greater size than the first.

Join Selectivity

- Join selectivity is a measure of the number of rows likely to result from joining two relations on a certain condition.

$$js = \frac{|R \bowtie_c S|}{|R \times S|} = \frac{|R \bowtie_c S|}{|R| * |S|}$$



$$js = 9.699 * 10^{-5}$$

- The DBMS maintains statistics on the values in each column, so it can estimate the result size of the joins in a particular query.

Viewing table statistics

```
SELECT attname, n_distinct, null_frac, avg_width
FROM pg_stats
WHERE tablename = 'actor';
```

attname	n_distinct	null_frac	avg_width
sex	2	0.0343333	2
role	-0.172226	0.00233333	10
origin	112	0.0166667	3
notes	501	0.524333	11
stage_name	-0.994122	0	13
given_name	-0.336664	0.295333	7
last_name	-0.195445	0.296667	7
birth_year	123	0.365333	4
death_year	0	1	4

- n_distinct**, when +ve is the number of distinct values in the column
when -ve is correlation between physical and logical row ordering.
- null_frac**: ratio of null values in column.

Table statistics

- To estimate join selectivity we need:
 - some idea of the number of non-null values in each column.
 - the number of unique values in each column.
- We can do better if we collect a histogram of how many occurrences of each value appear in the columns.
- Collecting table statistics can be an expensive operation. It is not usually done for each insert/update/delete.
- Use VACUUM ANALYSE to instruct PostgreSQL to collect new statistics.
- This is another useful addition to a nightly housekeeping script.

Logical vs Physical ordering correlation

- The statistical correlation between physical and logical row ordering is a measure of how well sorted a table is on a particular attribute.
- If the correlation coefficient is +1 (same order) or -1 (inverse order), an index scan on the column will be cheaper than if it is close to zero (uncorrelated)

fid	name	color	price
12	apple	red	0.80
14	banana	yellow	1.20
18	grape	green	0.05
27	durian	brown	15.80

- If the index order is in the same as the physical table order, then a linear scan of the index maps to a linear scan of the table.

An unsorted table

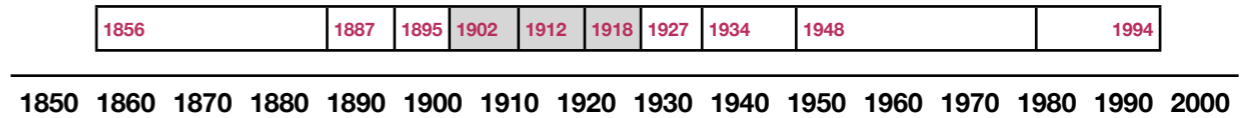
fid	name	color	price
18	grape	green	0.05
27	durian	brown	15.80
14	banana	yellow	1.20
12	apple	red	0.80

12	*
14	*
18	*
27	*

- If the rows in the table are not sorted on the same attribute as an index, then a linear scan of the index will result in many random accesses to table rows.
- The block based nature of the disk system implies that random access to large tables is less efficient than linear / sequential access.
- When performing a merge join with condition of low selectivity, it may be better to physically sort the rows in a table than to use an index scan.

Histogram buckets for actor_birth(year)

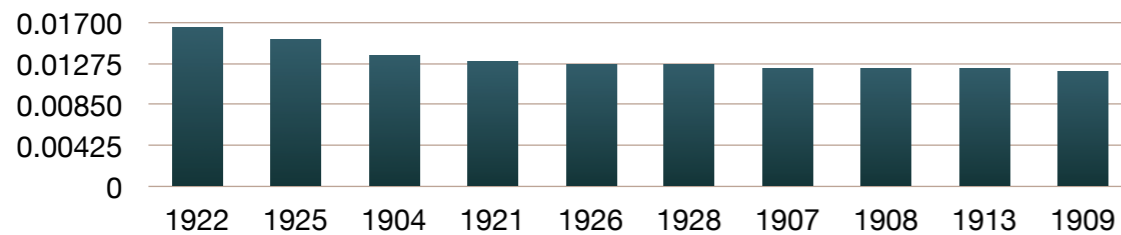
```
SELECT histogram_bounds FROM pg_stats  
WHERE tablename = 'actor'  
AND attname = 'birth_year';
```



- The histogram divides the actor birth_year column into bins containing an equal number of values
- For example, approx 30% of actors were born between 1902 and 1927.

Most common value freqs for actor(birth_year)

```
SELECT most_common_vals, most_common_freqs  
FROM pg_stats  
WHERE tablename = 'actor' AND attname = 'birth_year';
```

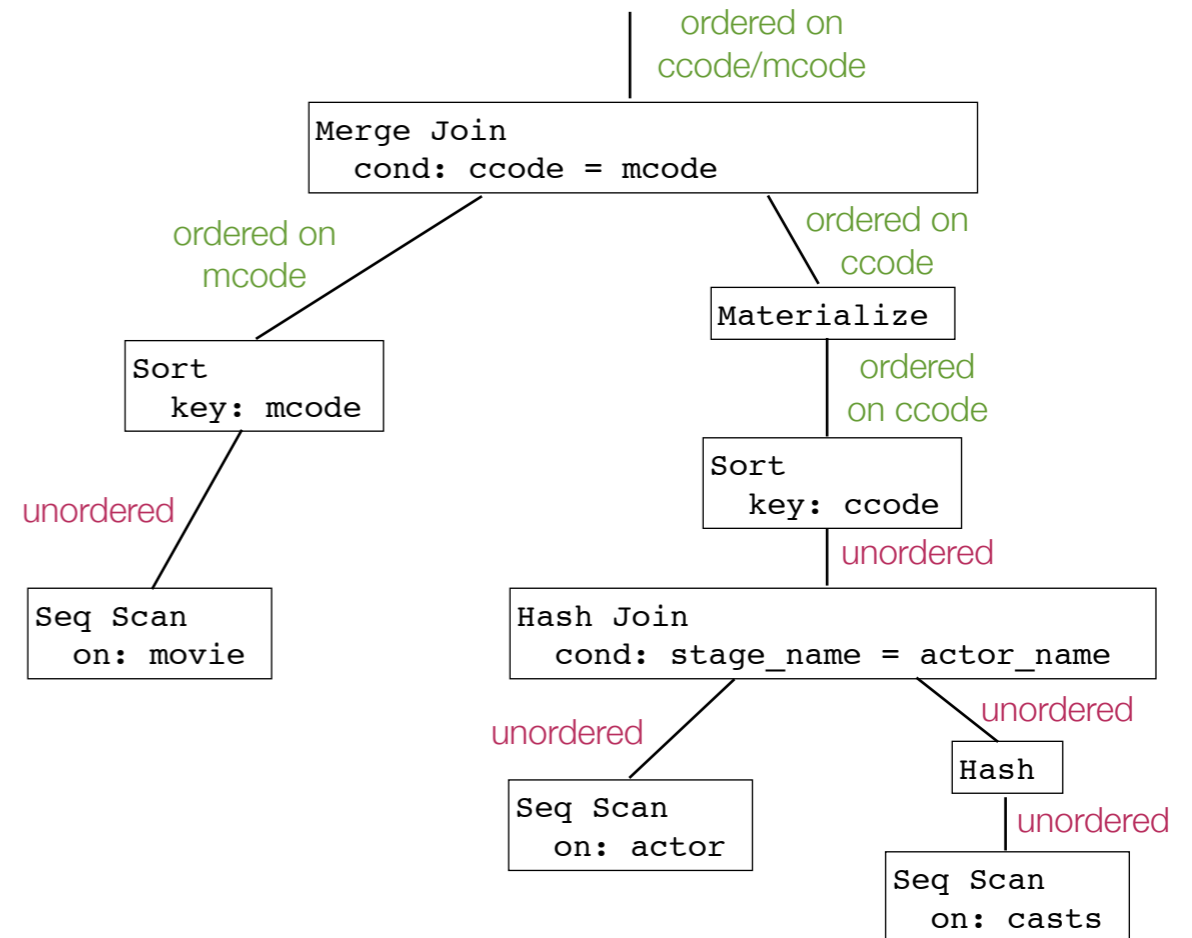
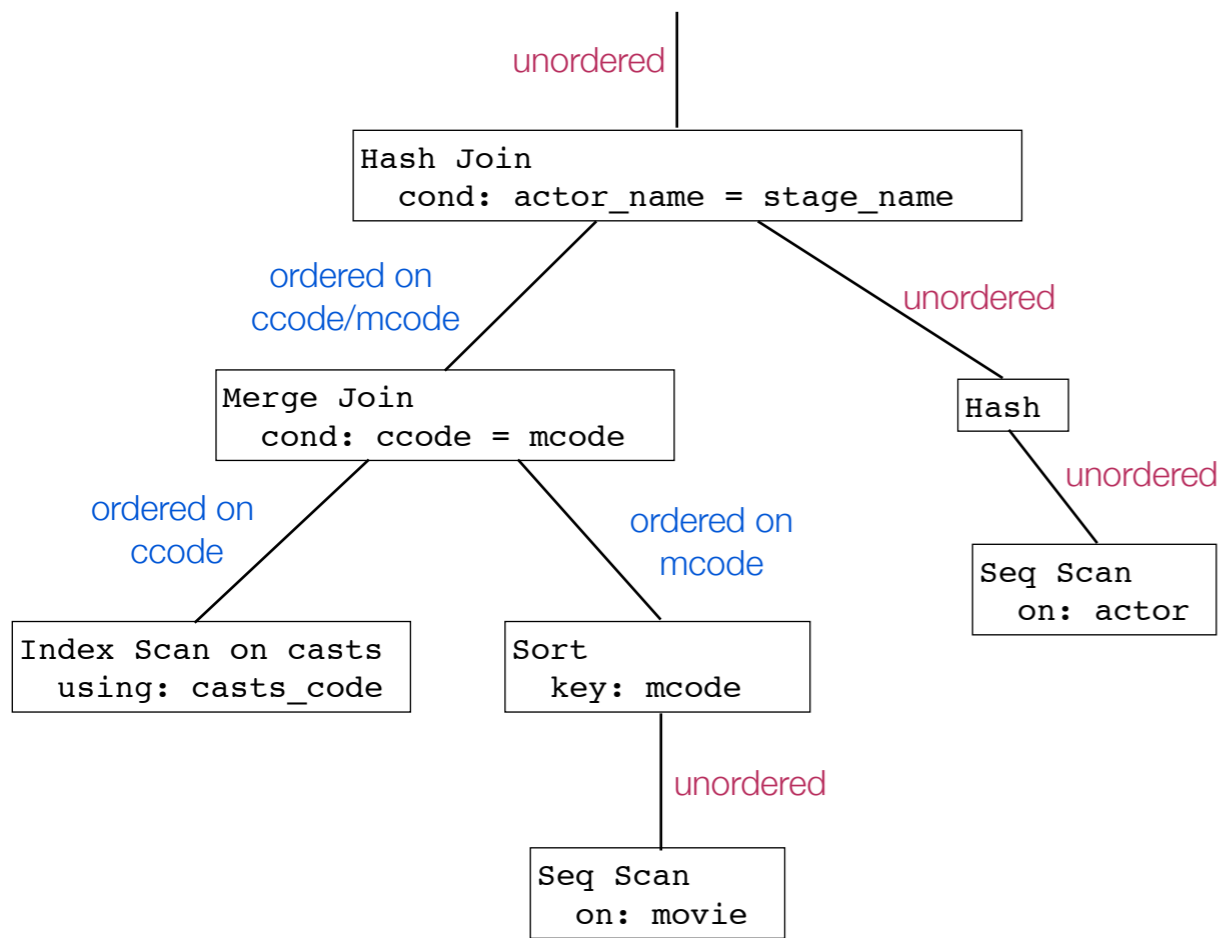


- PostgreSQL remembers the exact value frequencies for the most common values in the table. The number of frequencies remembered can be set.
- From this data we know that 1.3% of actors were born in 1926.

Interesting orders

```
SELECT movie.code, title, director,  
actor_name, birth_year  
FROM movie, actor, casts  
WHERE stage_name = actor_name  
AND movie.code = casts.code;
```

- For this query, we have not asked the output to be in any particular order.
- In this case the query planner is free to produce the result unordered, or in any order it wants to.
- However, opportunistic ordering can be exploited internally.
- For this query we already have an ordered index for casts.code...



Changing the result order affects the whole plan.

```

SELECT movie.code, title, director,
         actor_name, birth_year
FROM   movie, actor, casts
WHERE  stage_name = actor_name
AND    movie.code = casts.code
ORDER BY movie.code
  
```

- This time we have requested the output to be in a specific order.
- The unordered result of the previous plan is not suitable.
- We *could* make use of the original plan and additionally sort the result, but this could have a substantial cost if the result is large.
- It might be better to rearrange the operator tree to preserve the desired order..

The PostgreSQL Genetic Query Optimiser

- When we consider all the possible join orders, physical operators, selectivity factors, table statistics, intermediate table sizes and interesting orders, the number of possible plans becomes very large.
- When the query reaches a certain size, we cannot hope to consider all possible plans, and must fall back on heuristic search methods.
- PostgreSQL includes a genetic query optimiser, modeled after Darwinian evolution, which attempts to cross-breed plans to produce fitter ones.
- This optimiser is enabled when the number of relations in a query is more than `geqo_threshold` (default value is 12).

Final Exam (out of 70)

- SQL and the Relational Model. **[15 marks]**
- Integrity and integrity violations. **[5 marks]**
- UML Modeling and Translation **[12 marks]**
- Privacy **[3 marks]**
- Functional Dependencies and Normal Forms **[10 marks]**
- Transactions and Recovery **[5 marks]**
- Query Processing, File access and Hardware **[10 marks]**
- Relational Algebra **[10 marks]**

***Just reading lecture notes
is a terrible way to study!!!***

- .. and just watching tennis on TV is a terrible way to learn tennis.
- The material needs to pass
from the lecture notes and text book,
through your head,
and back down into your own notes.
- You can discard the notes afterwards, but the process of making them forces you to understand the material.

How to study for the exam.

- Go back and redo all of the lab exercises from scratch.

This may sound like a lot of work at first, but if you've done them all once and understood the material, the *second time around will be a lot faster.*

- Make a cheat sheets for the lecture material.

You can't take them in to the exam, but it's a good way to study.

First work out the main points of each lecture and reduce each one to a single A4 page. Once that is done reduce all these sheets to another single page.

- For each topic, imagine yourself explaining it to someone else who knows a little about computing, but nothing about databases.

You can't explain something unless you understand it yourself.