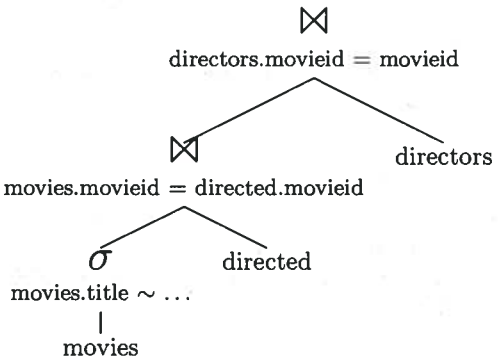
138+1/140

6.830
Database Systems
2007/10/10

Dan Ports
drkp@mit.edu
Collaborators: {amdragon}

Problem Set 2

Problem 1:



Postgres's optimizer chose this plan because it had the lowest estimated cost of all possible join orderings for left-deep query plans. This plan had a low cost because it was able to use indexes for all joins.

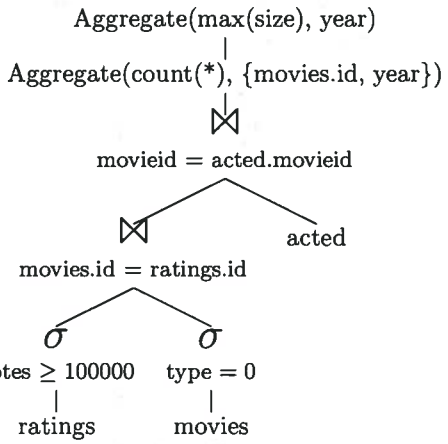
5 ✓

The estimated result size is 2. The actual result size is 10. ↗ 3, I think

Postgres estimates that the regex match on `movies` will have 2 rows; it actually has 12. It's not too surprising that this isn't accurate, since it's not clear how this could be estimated well.

This seems like the best plan for this fairly straightforward query; it joins the tables in a logical order and uses a sequential scan for the regex match and index scans for the other tables.

Problem 2:



5+5 ✓

Postgres's optimizer selected the plan with lowest estimated cost, using index joins for all joins and keeping the number of intermediate results low.

The estimated result size is 69 and the actual result size is 32.

The estimated size for the subquery is fairly accurate (69 rows, vs. an actual value of 60). On the other hand, some of the intermediate result size estimates were pretty far off: the scan on `ratings`

returned 60 rows instead of 12, and the join with `movies` returned 60 instead of 5. The join of the result with `acted` gives 4096 results instead of 69.

Although many of the estimated sizes differed from the actual results, the query plan Postgres chose still seems to be optimal in terms of keeping the size of the intermediate result sets small.

Problem 3:

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=5.69..5.69 rows=1 width=0)
   ->  Index Scan using movies_pkey on movies  (cost=0.00..5.43 rows=106 width=0)
         Index Cond: (id < 100)
(3 rows)
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Aggregate  (cost=25856.31..25856.31 rows=1 width=0)
   ->  Seq Scan on movies  (cost=0.00..23376.72 rows=991835 width=0)
         Filter: (id < 1000000)
(3 rows)
```

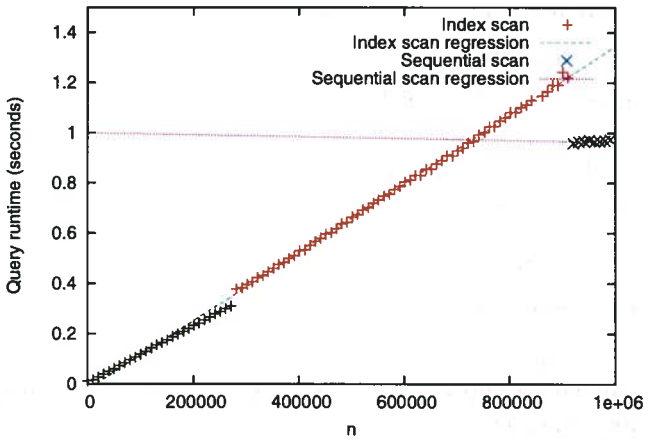
The first query scans the index to find movies with id less than 100; the second query scans the index directly under the (correct) assumption that most movies will have id less than 1000000. This can be more efficient, assuming that nearly all tuples really do match, because it eliminates the overhead of accessing the index (*e.g.* scanning the internal nodes of the tree, assuming it's a B-tree index).

good ✓

The cutoff point appears to be 911857. This is the point at which the estimated cost of the index scan exceeds 23376.16, the estimated cost of the sequential scan.

At the cutoff point, the index scan takes 1207 ms, and the sequential scan takes 941, so the cutoff point is clearly too high. To determine a better cutoff point, the graph below shows the runtimes for the index and sequential scans for varying values of n, and a linear regression. Assuming the sequential scan runtimes continue to follow the same trend, which seems reasonable, we expect the correct cutoff point to be at their intersection, or $n \approx 728749$.

10



Problem 4:

A heap file is the obvious choice for a workload consisting only of sequential scans, since the heap file has no index overhead and there wouldn't be any benefit to having the data ordered in any way.

Problem 5:

To perform this join, we'll need to use a nested loops join. Using an index on the inner relation will keep us from needing to iterate over all the tuples in the inner relation, though it needs to be a B$^+$-tree so that we can perform a range query. Using an index NL join, we need to perform one random I/O per tuple in the outer table, so we'll choose A, the smaller table, as the outer one. It should be represented as a heap file, since it will be scanned sequentially, and B should be represented as a dense-packed clustered B$^+$-tree. It needs to be clustered to eliminate the need for an additional random access per tuple, and dense-packed to minimize overhead; since there are no insertions, there's no downside to either of these.

Problem 6:

As above, we'll want to use index nested loop joins; since the queries are now equi-joins, a clustered hash file is a viable option for the inner table and more efficient since it avoids the B$^+$-tree overhead. Again, we'd like the outer table to be the smaller one, so A should be the outer table and represented by a heap file; B and C should be represented by clustered hash files indexed on f_1 .

Problem 7:

This is the same as the preceding workload except that the joins of A with C are now on f_2. We can use the same configuration (index nested loop join with A as the outer, A represented as a heap file, and B and C represented as clustered hash files), except that C should now be indexed on f_2.

Problem 8:

Since the majority of the workload is constant inequality queries on A, we'll want to represent A as a B$^+$-tree, which can handle these efficiently. It should be dense and clustered for the same reasons as (5). For the joins with B, we should use A as the outer, and B as the inner, represented as a clustered hash file indexed on f_1, as in the previous problem. There is slightly more overhead involved in the join since the outer relation is represented as a B$^+$-tree, but it should not be unreasonable.

Problem 9:

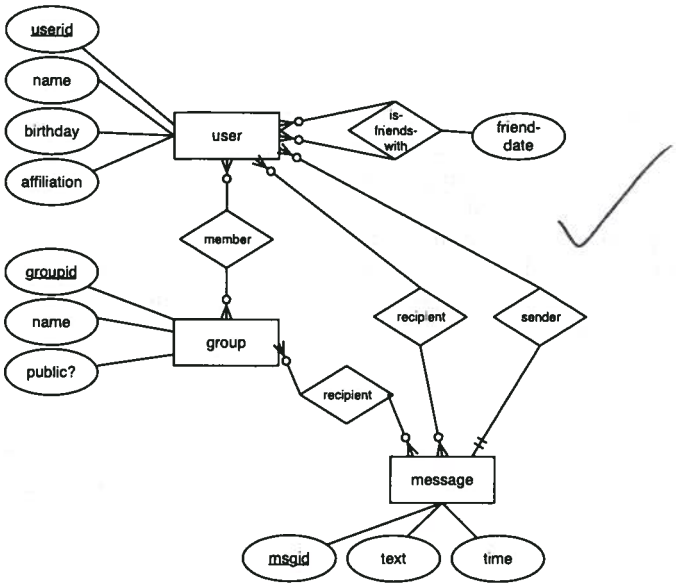
This is essentially the same workload as before, except with insertions in the mix. We choose the same representation (A as a clustered B$^+$-tree, B as a clustered hash file), except that the B$^+$-tree should no longer be dense-packed. Keeping the tree dense-packed would make insertions quite expensive.

Problem 10:

There are many update and deletion anomalies. One update anomaly is that a user's affiliation is stored in multiple tuples in the `friends` table (in either the `affiliation1` or `affiliation2` column, making it even worse). If the user's affiliation is changed, it needs to be changed in all of these locations. A deletion anomaly occurs due to the same problem. If a user's last friend is

removed, his or her affiliation, birthday, etc. will be lost.

Problem 11:



Problem 12:

The schema is as follows:

`user`	:	{`userid`, `name`, `birthday`, `affiliation`}
`friend`	:	{`friendid1`, `friendid2`, `frienddate`}
`group`	:	{`groupid`, `name`, `public`}
`group-member`	:	{`groupid`, `userid`}
`message`	:	{`messageid`, `text`, `time`, `senderid`}
`message-user-recipient`	:	{`messageid`, `userid`}
`message-group-recipient`	:	{`messageid`, `groupid`}

Problem 13:

The functional dependencies are listed below:

{`userid`} $\mapsto$ {`name`, `birthday`, `affiliation`}
{`friendid1`, `friendid2`} $\mapsto$ {`frienddate`}
{`groupid`} $\mapsto$ {`name`, `public`}
{`messageid`} $\mapsto$ {`text`, `time`, `senderid`}

Problem 14:

The resulting schema is the same as that derived from the ER diagram:

`user`	:	{`userid`, `name`, `birthday`, `affiliation`}
`friend`	:	{`friendid1`, `friendid2`, `frienddate`}
`group`	:	{`groupid`, `name`, `public`}
`group-member`	:	{`groupid`, `userid`}
`message`	:	{`messageid`, `text`, `time`, `senderid`}
`message-user-recipient`	:	{`messageid`, `userid`}
`message-group-recipient`	:	{`messageid`, `groupid`}

Problem 15:

The two resulting schemata are the same. This isn't too surprising, since the 3NF schema resulting from the ER model is also in BCNF.

Problem 16:

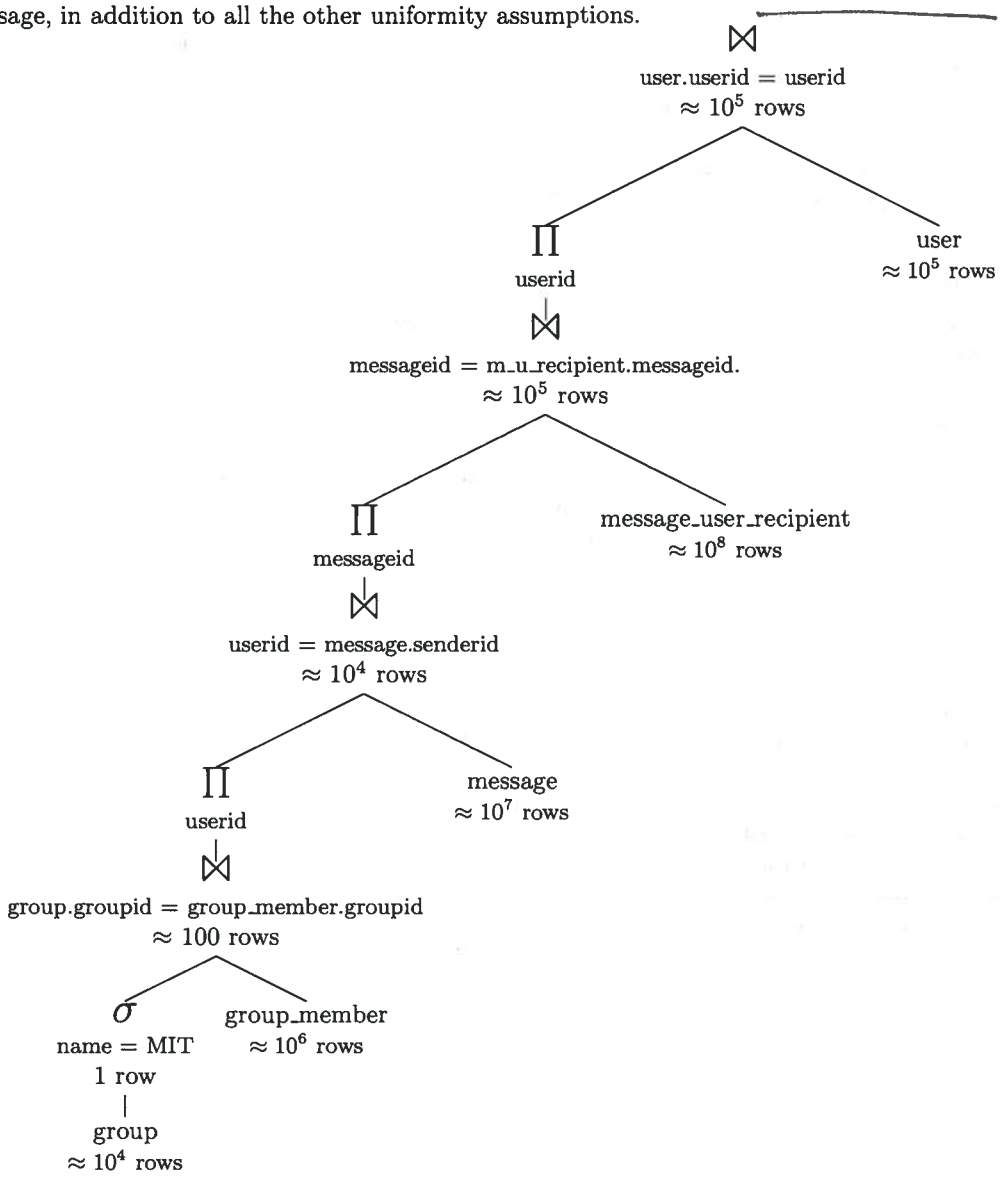
Yes. The schema preserves all functional dependencies (trivially so, since it was derived from the functional dependencies!). It avoids the update and deletion anomalies mentioned above, since information related to a user (affiliation, birthday, etc.) is stored only in a single place in the `user` table.

Problem 17:

```
SELECT user.name
FROM user, message_user_recipient, message,
     group_member, group
WHERE user.userid = message_user_recipient.userid
AND message.messageid = message_user_recipient.userid
AND message.senderid = group_member.userid
AND group.groupid = group_member.groupid
AND group.name = 'MIT';
```

Problem 18:

The optimal query tree is below. Note that we are additionally assuming an average of 10 recipients per message, in addition to all the other uniformity assumptions.



A naïve approach to executing this query would use nested loop joins for all the joins. This would incur a fairly horrific cost. However, observing that if we perform a projection after each join to keep only the key we need for the next join, we can fit all of the intermediate result sets in memory: the largest intermediate result sets are around 10^5 rows, and the keys are probably not larger than 64 bit integers, so this fits comfortably in our memory. This means that we can perform all of the joins as in-memory hash joins. As a result, the number of I/Os required is simply determined by the cost of scanning each of the tables involved. This is $10^4 + 10^6 + 10^7 + 10^8 + 10^5 \approx 10^8$ rows, or approximately 1.1×10^7 page I/Os, assuming 10 tuples per page. This is a lot, but is optimal; without indexes, we clearly can't do any better because we need to read the tables in their entirety.

Problem 19:

Assuming that we're optimizing specifically for this query, we can make it as efficient as possible by adding clustered hash indexes on all the join predicates: on `group.name`, `group_member.groupid`, `message.senderid`, `message_user_recipient.messageid`, and `user.userid`. This means that the selection can be performed using 1 I/O, and all the joins can be performed as index nested loop joins using 1 I/O per row of the outer table. This gives us a total of

$$1 + 1 + 100 + 10^4 + 10^5 \approx 1.1 \times 10^5 \text{ I/Os}$$

14

list which one is which