6.830 Problem Set 2 (2014) Solutions

Assigned: Monday, Sep 22, 2014

Due: Wednesday, Oct 8, 2014, 11:59 PM
Submit to Stellar: `https://stellar.mit.edu/S/course/6/fa14/6.830/homework/`

The purpose of this problem set is to give you some practice with concepts related to schema design, query planning, and query processing. Start early as this assignment is long.

Part 1 - Query Plans and Cost Models

In this part of the problem set, you will examine query plans that PostgreSQL uses to execute queries, and try to understand why it produces the plan it does for a certain query.

The data set you will use has the same schema as the MIMIC-II dataset you used in problem set 1. Rather than running your own instance of SQLite, however, you will be connecting to our PostgreSQL server, since PostgreSQL produces more interesting query plans than SQLite.

To understand what query plan is being used, PostgreSQL includes the `EXPLAIN` command. It prints the plan for a query, including all of the physical operators and access methods being used. For example, the following SQL command displays the query plan for the `SELECT`:

```
EXPLAIN SELECT * FROM d_meditems WHERE label LIKE '%Folate%';
```

In this problem, you will find `\di` and `\d tablename` commands useful. In order to use these, you must install PostgreSQL command-line client. **Make sure you use PostgreSQL 8.3+ so that your results are consistent with the solutions.**

Athena already has version 9.3.5 installed, so you can simply `ssh` into `athena.dialup.mit.edu` and get started. In case you want to work on your own Debian/Ubuntu machine, you can install the `postgresql-client` package by running the following command in your shell.

```
sudo apt-get install postgresql-client
```

You can then connect to our PostgreSQL server by running the following command.

```
psql -h vise3.csail.mit.edu -U mimic2 -d mimic2
```

Note that we currently only allow connections from MIT IPs so you will need to connect from on campus or by ssh'ing into Athena.

To understand the output of `EXPLAIN`, you will probably want to read the performance tips chapter of the PostgreSQL documentation:

`http://www.postgresql.org/docs/9.2/static/performance-tips.html`

In general to understand the plans that are generated you may need to spend a bit of time searching the Internet.

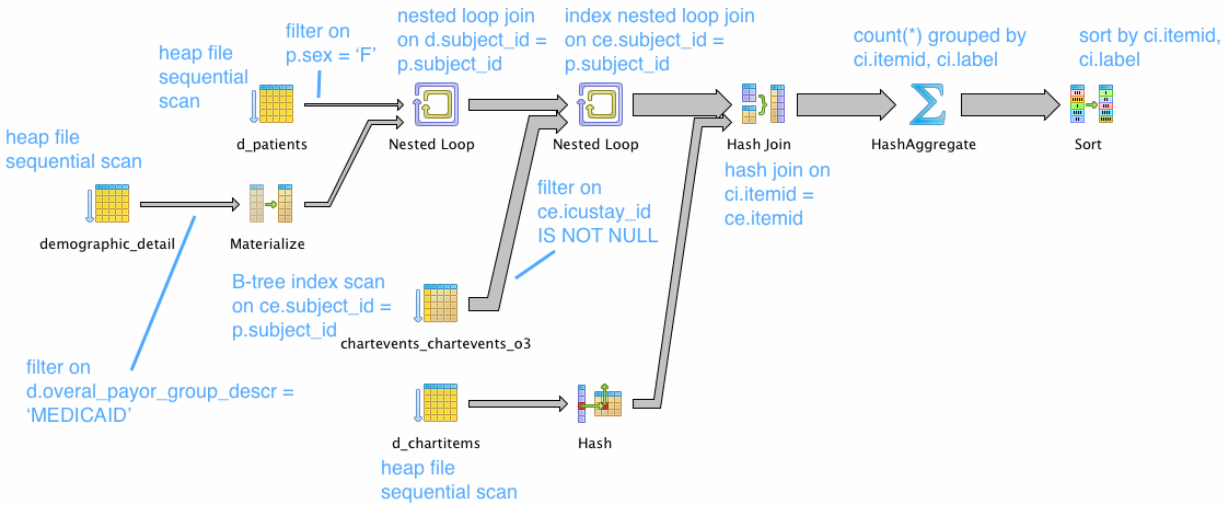
We have run `VACUUM FULL ANALYZE` on all of the tables in the database, which means that all of the statistics used by PostgreSQL server should be up to date.

1. [15 points]: *Query Plans*

Run the following query (using EXPLAIN) in PostgreSQL and answer questions (a) – (f) below:

```
EXPLAIN SELECT ci.itemid, ci.label, count(*)
FROM d_chartitems ci,
     chartevents ce,
     d_patients p,
     demographic_detail d
WHERE ci.itemid = ce.itemid
AND d.subject_id = p.subject_id
AND ce.subject_id = p.subject_id
AND d.overall_payor_group_descr = 'MEDICAID'
AND p.sex = 'F'
AND ce.icustay_id IS NOT NULL
GROUP BY ci.itemid, ci.label
ORDER BY ci.itemid, ci.label;
```

a. What physical plan does PostgreSQL use? Your answer should consist of a drawing of a query tree annotated with the access method types and join algorithms (note that you can use the pgadmin3 tool shown in class to draw plans, but will need to annotate them by hand.)



b. Why do you think PostgreSQL selected this particular plan?

Choice of access methods:

- Postgres does a sequential scan of demographic_detail, d_patients, and d_chartitems because there are no indexes on those tables.
- Postgres does an index scan on chartevents as part of an index nested loops join because it does not expect many tuples to be output from the previous join between demographic_detail and d_patients. Therefore, it expects that the small amount of random I/O required to seek for each record in the B-tree will cost less than the sequential I/O required to scan the entire table

Choice of join algorithms:

- Postgres performs a nested loops join between demographic_detail and d_patients because it expects so few rows from each table that it's not worth the overhead to build an in-memory hash table.
- See above for the reasoning behind the index nested loops join with chartevents

- Postgres performs a hash join with d_chartitems because there are too many rows for a nested loops join to be practical.

Join ordering:

- Postgres chose this join ordering because it's left-deep, it avoids any cross-products, and the first tables joined are most selective.

c. What does PostgreSQL estimate the size of the result set to be?

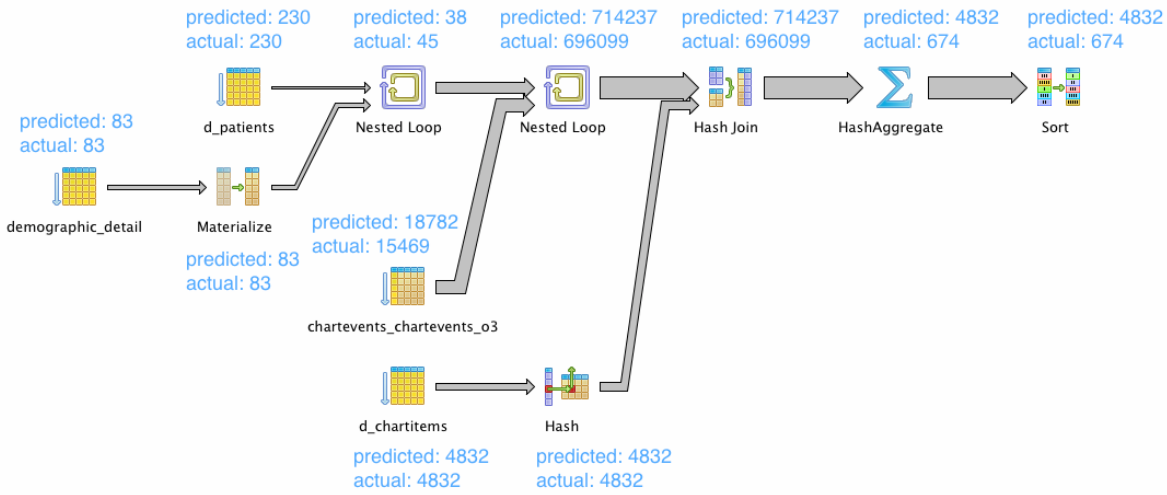
Postgres estimates the size of the result set to be 4832 rows, with an average width of 17 bytes.

d. When you actually run the query, how big is the result set?

It actually returns 674 rows.

e. Run some queries to compute the sizes of the intermediate results in the query. Where do PostgreSQL's estimates differ from the actual intermediate result cardinalities?

Postgres accurately predicts the number of rows for each sequential scan, but otherwise its estimates are incorrect:



f. Given the tables and indices we have created, do you think PostgreSQL selected the best plan? You can list all indices with `\di`, or list the indices for a particular table with `\d tablename`. If not, what would be a better plan? Why?

Even though Postgres' statistics were not always accurate, they were close enough (within an order of magnitude). The access methods, join ordering, and choice of join algorithms still make sense for the same reasons as described in part b.

2. [15 points]: *Estimating Cost of Query Plans*

Use `EXPLAIN` to find the query plans for the following two queries and answer question (a).

1. ```
   EXPLAIN SELECT m.label
   FROM d_chartitems AS c,
        chartevents AS ce,
        medevents AS me,
        d_meditems AS m
   WHERE c.itemid=ce.itemid
   AND ce.subject_id=me.subject_id
   AND me.itemid=m.itemid
   AND ce.subject_id > 100
   AND m.itemid>370;
   ```
2. ```
   EXPLAIN SELECT m.label
   FROM d_chartitems AS c,
        chartevents AS ce,
        medevents AS me,
        d_meditems AS m
   WHERE c.itemid=ce.itemid
   AND ce.subject_id=me.subject_id
   AND me.itemid=m.itemid
   AND ce.subject_id > 27000
   AND m.itemid>370;
   ```

a. Notice that the query plans for the two queries above are different, even though they have the same general form. Explain the difference in the query plan that PostgreSQL chooses, and explain why you think the plans are different.

The first query has a much less selective filter condition on subject_id than the second query. As a result, Postgres produces different plans for each query.

Access methods: Since Postgres predicts it will scan many rows in the first query, it does not perform any index scans. The second query scans many fewer rows, however, so it makes sense to perform an index scan on chartevents and medevents.

Join algorithms: The first query only uses hash joins and merge joins, since any other algorithm would be way too expensive. The second query uses an index nested loops join since it does not expect many tuples to be returned from the join between d_chartitems and chartevents.

Join order: The second query has a traditional left-deep plan with the most selective join first. The first query has a "bushy" plan, which Postgres determined through complex cost analysis to be the best plan.

Now use `EXPLAIN` to find the query plan for the following query, and answer questions (b) – (g).

3. ```
   EXPLAIN SELECT m.label
   FROM d_chartitems AS c,
        chartevents AS ce,
        medevents AS me,
        d_meditems AS m
   WHERE c.itemid=ce.itemid
   AND ce.subject_id=me.subject_id
   AND me.itemid=m.itemid
   AND ce.subject_id > 26300
   AND m.itemid>370;
   ```
   ```

b. What is PostgreSQL doing for this query? How is it different from the previous two plans that are generated? You may find it useful to draw out the plans (or use pgadmin3) to get a visual representation of the differences, though you are not required to submit drawings of the plans in your answer.

This query is has a slightly less selective filter condition than query 2, but still much more selective than query 1. The only difference between this plan and the plan for query 1 is that there is an index scan on chartevents rather than a sequential scan. This is due to the fact that there are many fewer expected rows from chartevents after filtering.

c. Run some more EXPLAIN commands using this query, sweeping the constant after the '>' sign from 100 to 27000. For what value of $subject\_id$ does PostgreSQL switch plans?
Note: There might be additional plans other than these three. If you find this to be the case, please write the estimated last value of subject_id for which PostgreSQL switches query plans.

Switch 1: 14854 $\rightarrow$ 14855

Switch 2: 26319 $\rightarrow$ 26320

d. Why do you think it decides to switch plans at the values you found? Please be as quantitative as possible in your explanation.

Switch 1: According to EXPLAIN, Postgres expects 5971620 rows to be returned from scanning chartevents with ce.subject_id $>$ 14854. With ce.subject_id $>$ 14855, it expects 5930164 rows. Postgres likely estimates that the cost of random I/O for 5971620 seeks is higher than the cost to scan the entire table, while the cost of random I/O for 5930164 seeks is less than the cost to scan the entire table.

Switch 2: When ce.subject_id $>$ 26319, Postgres expects 56224 rows to be returned from scanning chartevents. When ce.subject_id $>$ 26320, however, Postgres only expects 1 row to be returned from scanning chartevents. This drastic difference in expected rows explains the drastic difference between the plans.

e. Suppose the crossover point (from the previous question) between queries 2 and 3 is $subject\_id_{23}$. Compare the actual running times corresponding to the two alternative query plans at the crossover point. How much do they differ? Do the same for all switch points.
Inside `psql`, you can measure the query time by using the `\timing` command to turn timing on for all queries. To get accurate timing, you may also wish to redirect output to `/dev/null`, using the command `\o /dev/null`; later on you can stop redirection just by issuing the `\o` command without any file name. You may also wish to run each query several times, throwing out the first time (which may be longer if the data is not resident in the buffer pool) and averaging the remaining runs.

Switch 1: Before the crossover point, the query takes about 65.3 ms. After, it takes about 64.9 ms.

Switch 2: Before the crossover point, the query takes about 63.2 ms. After, it takes about 1.6 ms.

f. Based on your answers to the previous two problems, are those switch points actually the best place to switch plans, or are they overestimate/underestimate of the best crossover points? If they are not the actual crossover point, can you estimate the actual best crossover points without running queries against the database? State assumptions underlying your answer, if any.

Based on the runtimes calculated in part e, the first switch point seems like a good place to switch. Since the times are nearly equal, Postgres must have accurately calculated the tradeoff between random I/O in the index scan and sequential I/O in the sequential scan. Based on the answer to part d, the second switch point also seems like a good place to switch since the number of expected rows is so drastically different.

Part 2 – Query Plans and Access Methods

In this problem, your goal is to estimate the cost of different query plans and think about the best physical query plan for a SQL expression.

Suppose you are running a web service, "sickerthanyou.com", where users can upload their genome data, and that uses genome processing algorithms to provide reports to users about their genetic disorders.

Data is uploaded as files, with metadata about those files stored in a database table. Each file is passed through one or more fixed processing pipelines, which produce information about the disorders a patient has. These results are stored in another database table.

The database contains the following tables:

```
// user u_uid with a name, gender, age, and race
CREATE TABLE users(u_uid int PRIMARY KEY,
                   u_name char(50),
                   u_gender char,
                   u_age int,
                   u_race int)

// relatives table records familial relationships between users
// relationships are not symmetric, so e.g., if A is B's child, then
//  there will be two entries: (A,B,child) and (B,A,parent)
CREATE TABLE relatives(rl_uid1 int,
         rl_uid2 int,
         rel_type int,
       PRIMARY KEY (rl_uid1, rl_uid2));

// data set collected about a specific user on a specific
//  genome sequencing platform,
//  with results stored in a specified file
CREATE TABLE datasets(d_did int PRIMARY KEY,
d_time timestamp,
d_platform char(20),
d_uid int REFERENCES users(u_uid),
d_file char(100));

// results produced by processing pipeline
// each disorder column represents a specific condition
//  (e.g., disorder1 might be diabetes, disorder2 might be a type of cancer, etc.)
CREATE TABLE results(r_rid int PRIMARY KEY,
r_did int REFERENCES datasets(d_did),
        r_uid int REFERENCES users(u_uid),
r_pipeline_version int,
r_timestamp int,
r_disorder1probability float,
...
r_disorder50probability float)
```

In this database, `int`, `timestamp`, and `float` values are 8 bytes each and characters are 1 byte. All tuples have an additional 8 byte header. This means, that, for example, the size of a single `results` record is $8 + 8 \times 5 + 8 \times 50 = 448$ bytes.

You create these tables in a row-oriented database. The system supports heap files and B+-trees (clustered and unclustered). B+-tree leaf pages point to records in the heap file. Assume that you can have at most one clustered index per file, and that

the database system has up-to-date statistics on the cardinality of the tables, and can accurately estimate the selectivity of every predicate. Assume B+-tree pages are 50% full, on average. Assume disk seeks take 10 ms, and the disk can sequentially read 100 MB/sec. In your calculations, you can assume that I/O time dominates CPU time (i.e., you do not need to account for CPU time.)

For the queries below, you are given the following statistics:

Statistic	Value
Number of users	10^6
Number of datasets	10^7
Number of results	10^8
Number of relatives	10^7

In the absence of other information, assume that attribute values are uniformly distributed (e.g., that there are approximately the same number of relatives per user, datasets per user, results per dataset, etc).

Suppose you are running the query `SELECT COUNT(*) FROM results WHERE d_uid = 1`. Answer the following:

3. [1 points]: In one sentence, what would be the best plan for the DBMS to use assuming no indexes? Approximately how long would this plan take to run, using the costs and table sizes given above?

The best plan is a sequential scan of results. $\frac{448 \text{ bytes} \times 10^8}{10^8 \text{ bytes scanned/sec}} = 448$ seconds

4. [2 points]: In one sentence, what would be the best plan for the DBMS to use assuming a clustered B+tree index on `d_uid`? Approximately how long would this plan take?

Assuming that there is one user per `d_uid`, the best plan is a index lookup on the B+Tree. The cost will be the cost to read the inner pages of the B+Tree, the time to read the leaf page of the B+Tree, and the time to read the record from the heap file. Assuming 4K disk pages, and 8 byte pointers and 8 byte keys, we can fit 4096/16 = 256 entries per B+Tree page. Since pages are half full, we have approximately 128 entries per page. $log_{128}10^8 = 3.80$, so we will need to read 4 B+Tree pages, plus the heap page, for a total of 50 ms. The total time to scan the record will be negligible.

5. [2 points]: In one sentence, what would be the best plan for the DBMS to use assuming an unclustered B+tree index on `d_uid`? Approximately how long would this plan take?

Because each `d_uid` is unique, there will be no difference between a clustered an unclustered index in this case.

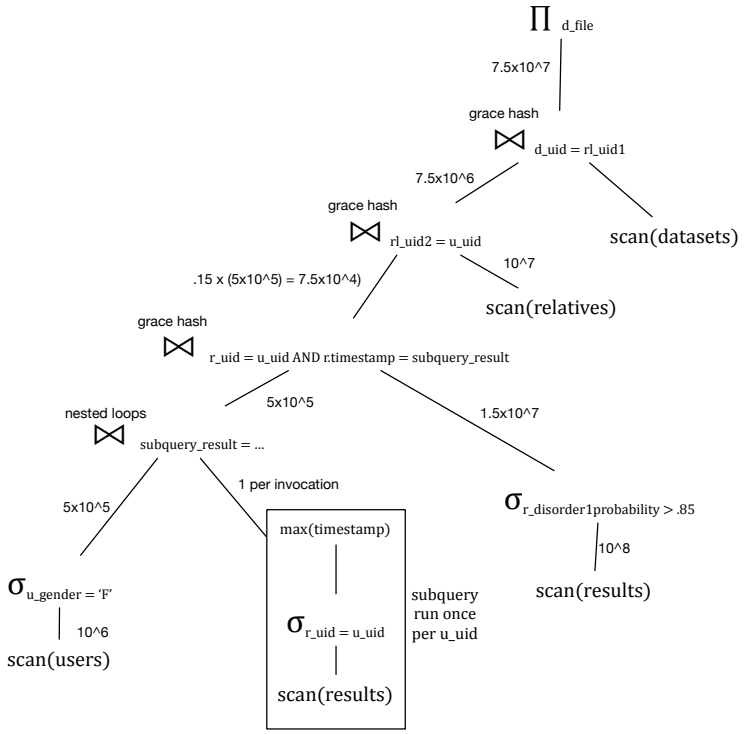
Now consider the following query that finds genome files of people who have female relatives with a high likelihood of having a particular disorder:

```
SELECT d_file
FROM users as u, relatives as rl, datasets as d, results as r
WHERE d_uid = rl_uid1
AND rl_uid2 = u_uid
AND r_uid = u_uid
AND u_gender = 'F'
AND r_disorder1probability > .85
AND r_timestamp = (SELECT MAX(r_timestamp) FROM results WHERE r_uid = u_uid)
```

Note that most database systems will execute the subquery as a join node where the right hand (inner) side of the join is the aggregate query, parameterized by `u_uid`, and the left hand is a table or intermediate result that contains `u_uid`.

6. [3 points]: Suppose only heap files are available (i.e., there are no indexes), and that the system has grace (hybrid) hash, merge join, and nested loops joins. For each node in your query plan indicate (on the drawing, if you wish), the approximate output cardinality (number of tuples produced.)

The key to this question is understanding what the query computes. Note that the two predicates on the `results` table will only allow tuples that are BOTH the maximum timestamp record and where the `disorder1probability` is $> .85$. The predicate on `disorder1probability` cannot be pushed into the subquery. The best plan is shown below:



Since we don't have information about how much memory the system has, it is likely that the best option is to just use grace hash for each join, assuming sufficient memory to do it in 3 passes over the two relations. We have to do a nested loops join for the subquery because we need to evaluate it once per `u_uid`.

Since users is the smallest table, it makes sense to filter it first, and then perform the subquery on it (to minimize the number of subquery invocations, as other joins will increase the number of results output.) The join with results won't increase the number of records output at all, because each user joins with at most one result (actually only 15% of the users will join, due to the filter on `disorder1probability`.) The join will relatives increases the output size by a factor of 100, since each user appears to have 100 relatives (on average). The join with `datasets` further increases the output by a factor of 10. This is a bit surprising because the total number of outputs is almost equal to the number of datasets – in fact it is quite likely that there are a number of duplicate datasets here because relatives are duplicated across users.

Note that there is a semantically equivalent rewrite of the query that would likely result in much better performance because it eliminates the correlated subquery that has to be run once per `uid`:

```
SELECT d_file
FROM relatives as rl, datasets as d, results as r,
(SELECT u_uid, MAX(r_timestamp) as timestamp FROM users, results
  WHERE u_uid = r_uid AND u_gender = 'F') as t
WHERE d_uid = rl_uid1
```

```
AND rl_uid2 = t.u_uid
AND r_uid = t.u_uid
AND r_disorder1probability > .85
AND r_timestamp = t.timestamp
```

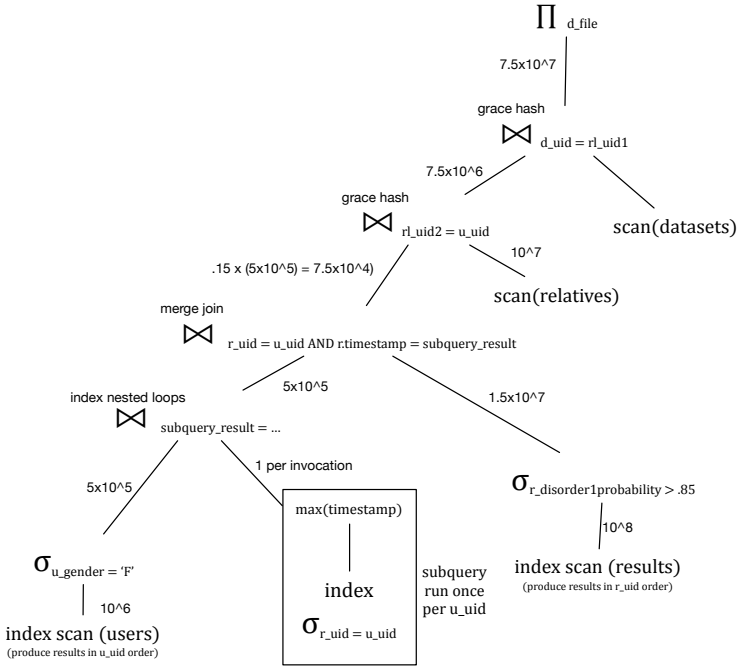
It's unlikely the database system will be able to find this, but we would accept answers that assumed this transformation was applied first. Note that in this case the filter on `disorder1probability` still can't be pushed into the subquery because we don't want to consider any results that aren't the most recent for a user.

7. [2 points]: Estimate the runtime of the plan you drew, in seconds.

The runtime will be dominated by the time to repeatedly scan `results` as a part of the subquery (once per user). Since a scan of `results` takes 448 seconds, the entire plan will be $10^5 x 448 = 4.48 \times 10^7$ seconds. None of the other joins will require anything within a factor of 1000 of this, so they are negligible.

8. [2 points]: Now, suppose that there are clustered B+Trees on `u_uid, d_uid, rl_uid1, r_uid`, and an unclustered B+Tree on `rl_uid2`. Draw the new plan you think the database would use and estimate its runtime methods available to it.

The index on `r_uid` really helps, since we can use it in the repeated invocations of the subquery to eliminate the sequential scans. There will be several timestamps per result, but they should all fit in one page. We can also replace the first grace hash join with a merge join, because we can use the indexes `results` and `users` to scan the tables in `uid` order. This index scan is (hopefully) fast since the indexes are clustered. The plan is shown below:



The overall runtime is no longer dominated by the subquery. The breakdown is as follows:

- Assuming each subquery invocation takes 50 ms, we have $.05 \times 5\ times 10^5 = 2500$ seconds for the first join with the subquery.
- The scan of `users` takes 83 bytes per user $\times\ 10^6$users$/100$ MB/sec = $.83$ seconds; we don't need to go through the index at all since the table is clustered in u_uid order.

- The scan of `results` takes 448 seconds. The join takes no additional I/O since both inputs are already sorted.
- A scan of `relatives` takes 32 bytes per record $\times 10^7$records$/100$ MB/sec $= 3.2$ seconds. Since we are doing grace hash we also have to write out the left input, which is 7.5×10^4 records. We have omitted projections in the plan, but we only need `u_uid` at this point so records are only 16 bytes (including header), so the total time to write out will be $7.5 \times 10^4 \times 16/100$ MB/sec $= .012 seconds$. We have to do 3 passes over `relatives` (one to read in, one to write out hashes, and one to read hashes back in), and two over the left input (just one to write out hashes and one to read back in). So the total time is $3.2 \times 3 + .012 \times 2 = 9.6$ seconds. (Note that since the left input is so small we might well have been able to fit it into memory, allow using to use a plain hash join!)
- Finally, we have to do the grace hash with `datasets`. We have 100 times as many records on the left input, so writing these out / reading in will now take 1.2 seconds each. Each pass of `datasets` will take 145 bytes per record $\times 10^7$datasets$/100$ MB/sec $= 14.5$ seconds. The total time is $14.5 \times 3 + 1.2 \times 2 = 45.9$ seconds.

The total is $2500 + .83 + 448 + 9.6 + 45.9 = 3004.33$ seconds.

Note that you might be able to do slightly better by scanning `relatives` in `rl_uid1` order, if you could preserve that order through the grace hash on `rl_uid2 = u_uid`, since you could then use the index on `d_uid` to do a merge join for the last join, avoiding the extra 45.9 seconds of I/O.

9. [2 points]: Suppose you could choose your own indexes (assuming at most one clustered index per table) for this plan; what would you choose, and why? Justify your answer quantitatively.

Clustering `users` on `u_gender` or `results` on `r_disorder1probability` probably won't help, since we have to pay the extra cost of doing the grace hash in this case. A clustered index on `rl_uid2` would allow us to replace the `rl_uid2 = u_uid` with a merge join without sorting, avoiding 9.6 seconds of I/O. This would only be a good idea if the optimization to scan `relatives` in `rl_uid1` order mentioned above was not applied.

10. [3 points]: Now suppose you load the same data into a column-store. Suppose there are no unclustered indexes, but that you can sort each table in some specific key, and that the database can quickly lookup records according to the sort key (e.g., if you sort `results` on `r_disorderprobability1`, you can quickly find all `r_disorderprobability1` records with value $> .85$. Further assume that the column-store has 1 byte of overhead per column (instead of the 8 bytes of overhead per record in the row-store.) For each table, list the sort key you would choose, and estimate the runtime of the above query in the column store (you don't need to draw out the query plan – just estimate the I/O costs.)

The basic approach is going to be pretty similar to that given above. We can sort `users` on `u_uid` order and `results` on `r_uid` to make the 2nd join a merge join (as above.) The subquery is still going to be painfully slow, but this sorting should make each subquery require just a few I/Os each. The exact implementation will affect the performance here – one option is to perform binary search, which will give performance that scales with $\log_2$ of the number of pages required to store the `r_uid` column. If each entry requires 9 bytes, and pages are 4 KB, this is $log_2(9 \times 10^8/4096) = 18$ I/Os, plus 1 to read the corresponding page of the `timestamp` column (more efficient implementations are possible). Sorting relatives in `rl_uid1` order will allow us to make the last join into a merge join as well, if we also sort datasets on `d_uid`. So the only joins we have to do I/O for are the first one (the subquery) and the 3rd one `rl_uid2 = u_uid`. We also have to scan the needed columns of the datasets. The costs are as follows:

- Time to scan `gender` and `u_uid`: 18 bytes per user $\times 10^6$users$/100$ MB/sec = $.18$ seconds.
- Time to perform 5×10^5 subqueries: $.190$ seconds per subquery $\times 5 \times 10^5 = 9500$ seconds
- Time to perform `rl_uid2 = u_uid` join: 9.6 seconds (as above)
- Time to scan `r_disorder1probability`, `r_uid` and `r_timestamp` columns from `results` table: 27 bytes per results $\times 10^8$results$/100$ MB/sec = 27 seconds.

- Time to scan `rl_uid1` and `rl_uid2` columns from `relatives` table: 18 bytes per relative$\times 10^7$relatives$/100$ MB/sec = 1.8 seconds.
- Time to scan `d_uid` and `d_file` columns from `datasets` table: 110 bytes per dataset$\times 10^7$datasets$/100$ MB/sec = 11.1 seconds.

So the total in this case is 9540 seconds, dominated by the time to perform binary search as a part of the subquery.

Part 3 – Schema Design and Query Execution

An online-shopping company has its own delivery system including several warehouses. Your job in this problem set is to design a schema for keeping track of the warehouses, orders and the logistics.

Specifically, you will need to keep track of:

1. The address, city, state, manager (unique), employees of every warehouse
2. The name, phone-number, salary of every employee and manager
3. The order_ID, product_ID, custumer_ID of every order
4. The shipping information of an order, which may include several records about shipping from one warehouse to another, with shipping time, delivery time, and processors information (one at each end)
5. The maximum numbers of orders that can be delivered between each two warehouses everyday

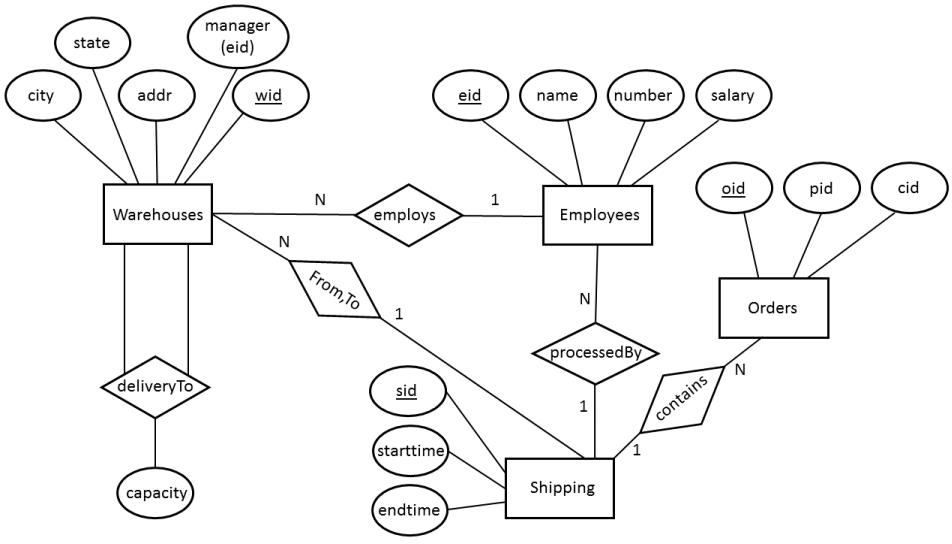
11. [2 points]: Write out a list of functional dependencies for this schema. Not every fact listed above may be expressible as a functional dependency.

Solution:

```
(wid) --> (addr, city, state, manager)
(eid) --> (name, number, salary)
(oid) --> (pid, cid)
(sid) --> (wid1, wid2, starttime, endtime, eid)
(wid1, wid2) --> (capacity)
```

12. [3 points]: Draw an ER diagram representing your database. Include a few sentences of justification for why you drew it the way you did.

Solution:



13. [2 points]: Write out a schema for your database in BCNF. You may include views. Include a few sentences of justification for why you chose the tables you did.

Solution:

```
Warehouses(wid, addr, city, state, manager references Employees.eid)
Employees(eid, name, number, salary)
Orders(oid, pid, cid)
Shipping(sid, wid1 references Warehouses.wid, wid2 references Warehouses.wid, starttime, endti
DeliveryTo(wid1 references Warehouses.wid, wid2 references Warehouses.wid, capacity)
```

14. [2 points]: Is you schema redundancy and anomaly free? Justify your answer.

Solution:

```
Yes. This schema is redundancy free and anomaly free.
For every functional dependency X->A, the X is a superkey in the schema.
```

15. [3 points]: Suppose you wanted to ensure that on each day, the number of orders that are shipped is less than 95% of the maximum delivering capability between two warehouses. Can you think of a way to enforce that via the schema? (i.e., by creating a table or modifying one of your existing tables?) How else might you enforce this?

Solution:

There's no way to enforce this with the schema.

One solution would be to use an assertion or trigger that many database systems provide.

The DBMS will verify this constraint after a each modification transaction. Here is an example of the `CREATE ASSERTION` syntax that could be used to provide this (we didn't necessarily expect you to provide this level of detail in your answer):

```
CREATE ASSERTION assert
CHECK (NOT EXISTS (
    SELECT s.wid1, s.wid2, julianday(s.starttime) as day, COUNT(*) as ct
    FROM Shipping AS s
    GROUP BY s.wid1, s.wid2, day
    HAVING ct >= 0.95* (SELECT capacity
                        FROM DeliveryTo AS dt
                        WHERE s.wid1 = dt.wid1 AND s.wid2 = dt.wid2)
                        ));
```

You could also create a helper table `ShippingCollectiveByDay(wid1, wid2, day, count)` and use triggers to update this helper table, checking the integrity constraint after inserting or updating the `Shipping`.