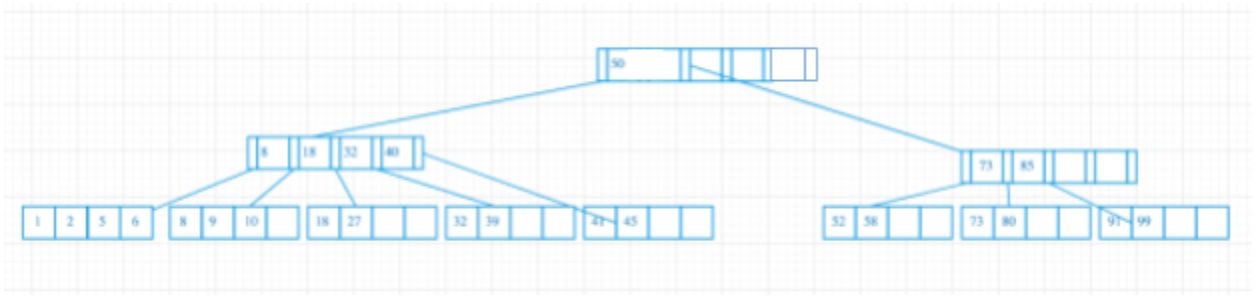
Problem Set 2

Manesha Ramesh, Bokang Jia

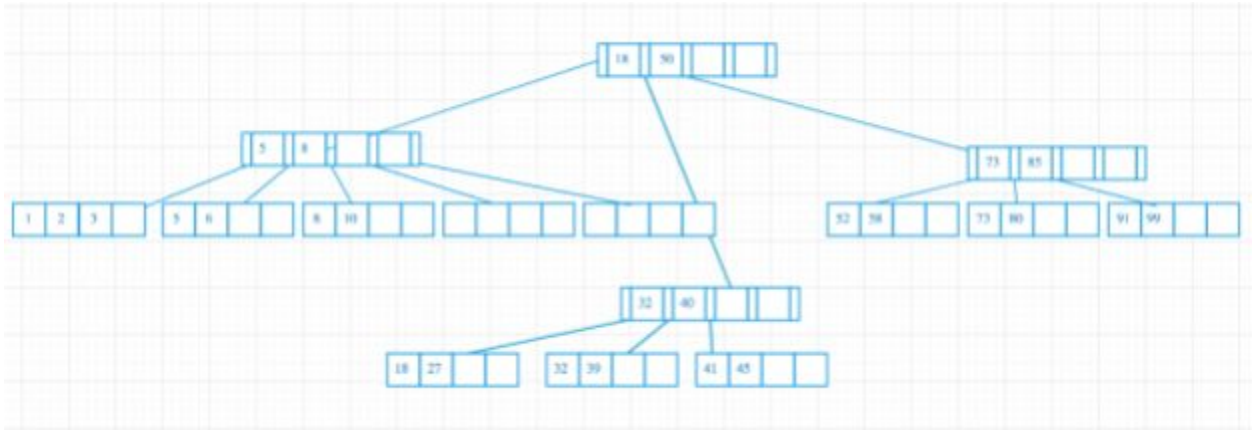
1) B+ trees deep dive

Exercise 10.1:

1)

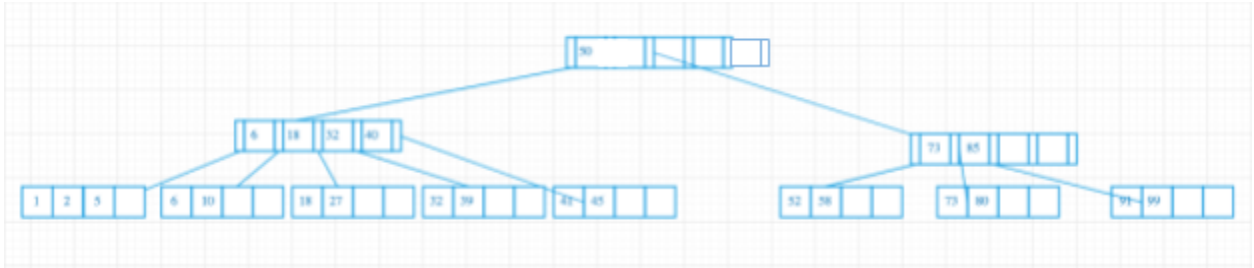


2)

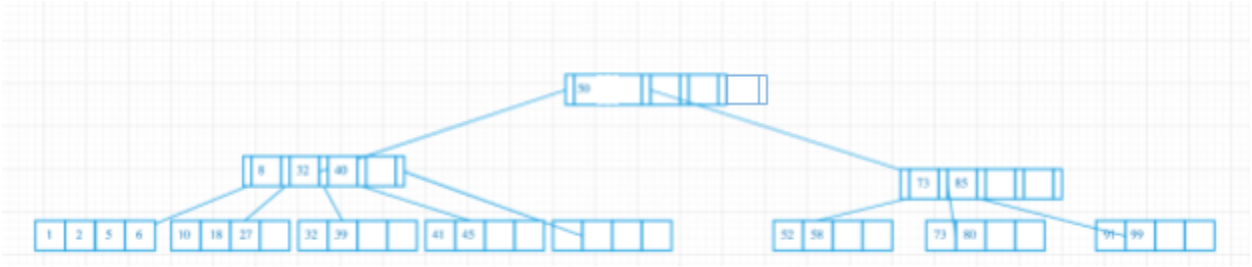


5 page writes, 4 page reads and 2 new pages

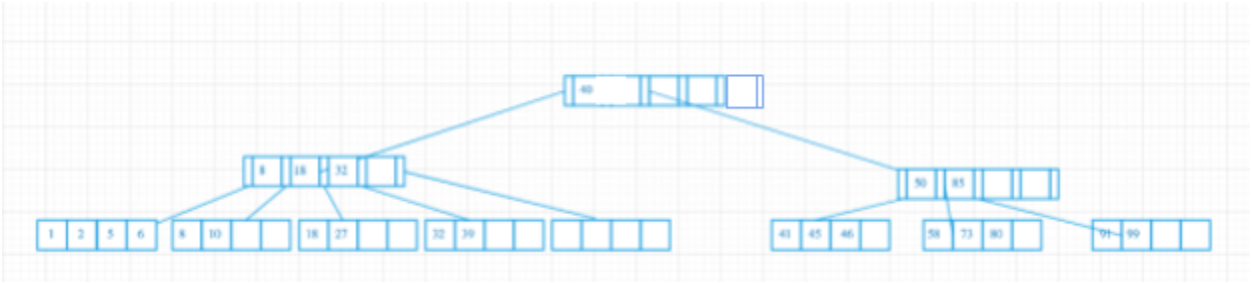
3)



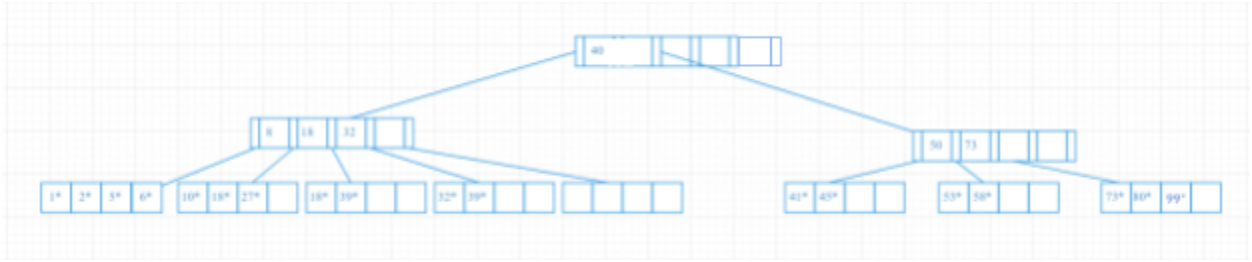
4)


50
8
32
40
73
85
1
2
5
6
10
18
27
32
39
41
45
52
58
73
80
91
99

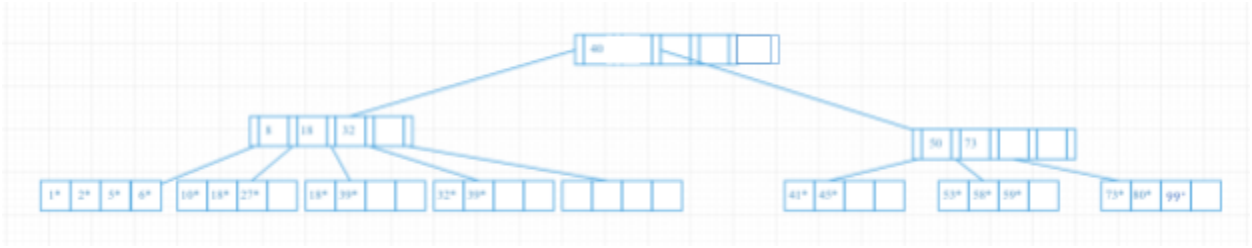
5)


40
8
18
32
50
85
1
2
5
6
8
10
18
27
32
39
41
45
46
58
73
80
91
99

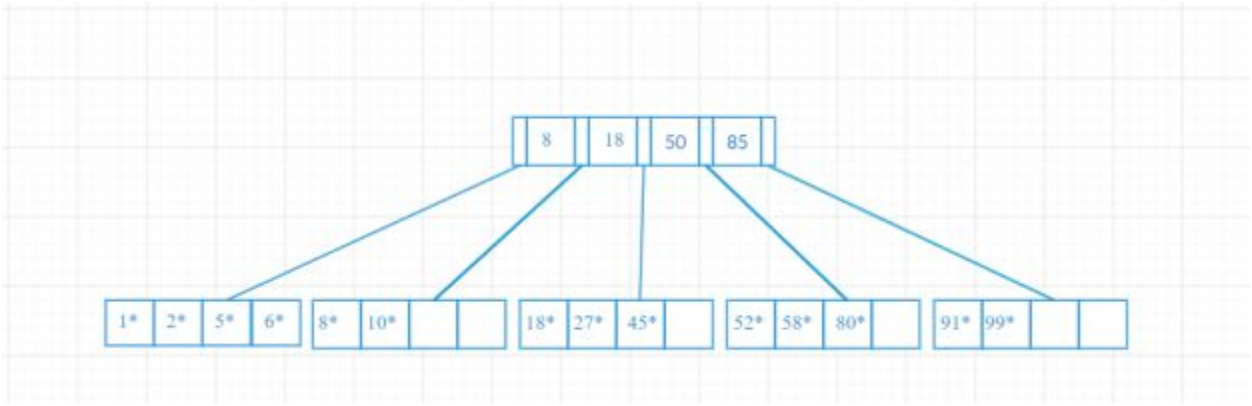
6)


40
8
18
32
50
73
1*
2*
5*
6*
10*
18*
27*
18*
39*
32*
39*
41*
45*
53*
58*
73*
80*
99*

7)


40
8
18
32
50
73
1*
2*
5*
6*
10*
18*
27*
18*
39*
32*
39*
41*
45*
53*
58*
59*
73*
80*
99*

8)

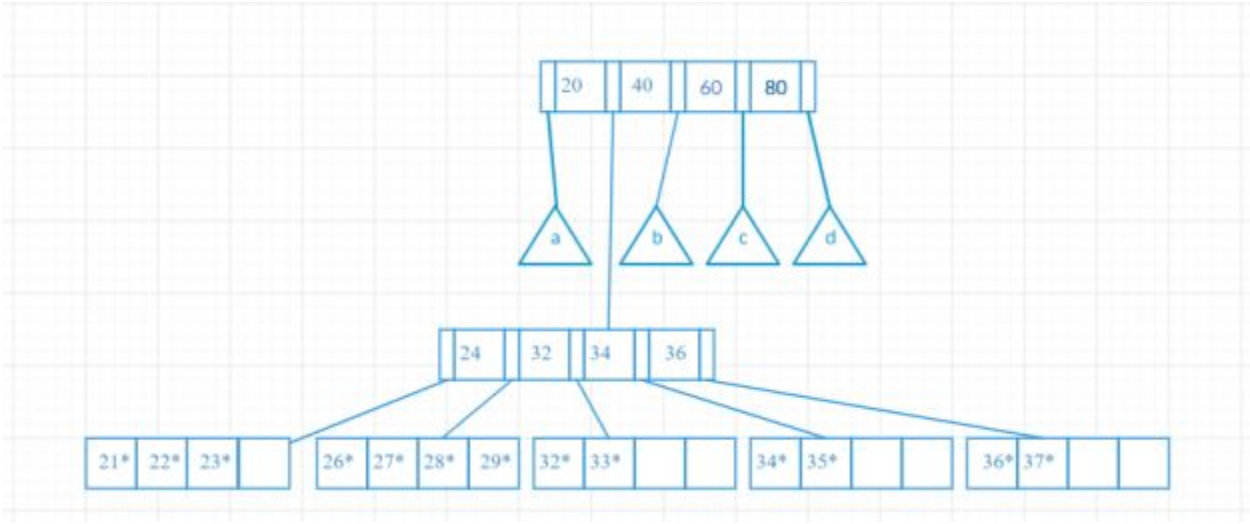


(redistribution with left sibling)

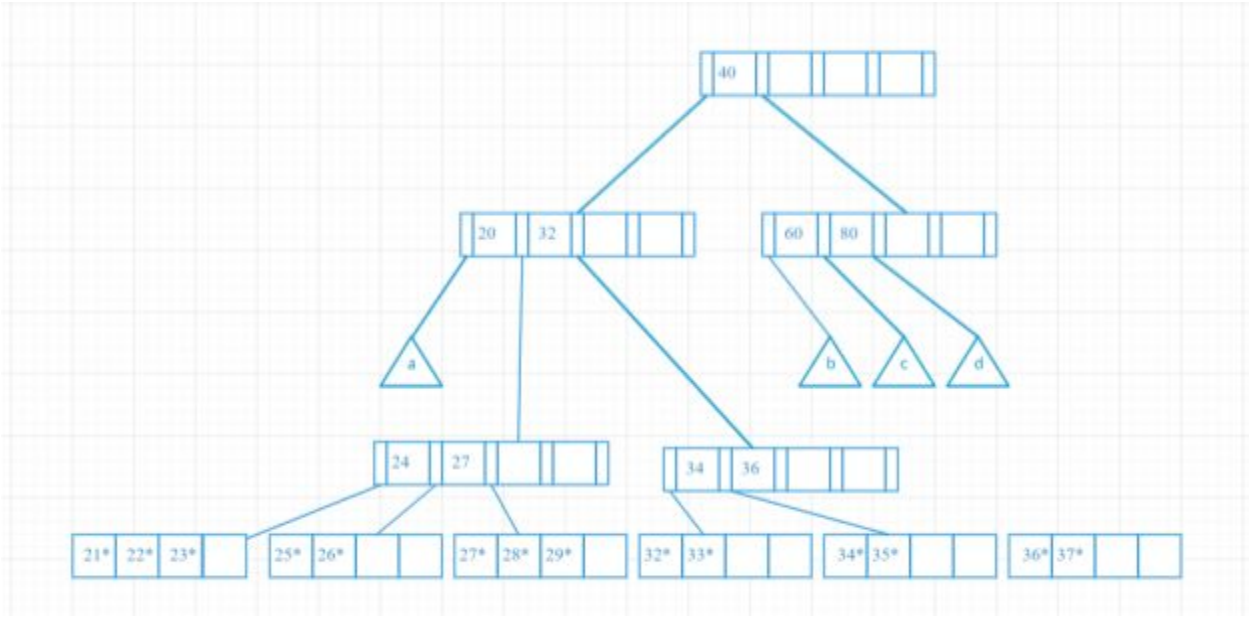
Exercise 10.4:

1)

a) Before insertion:

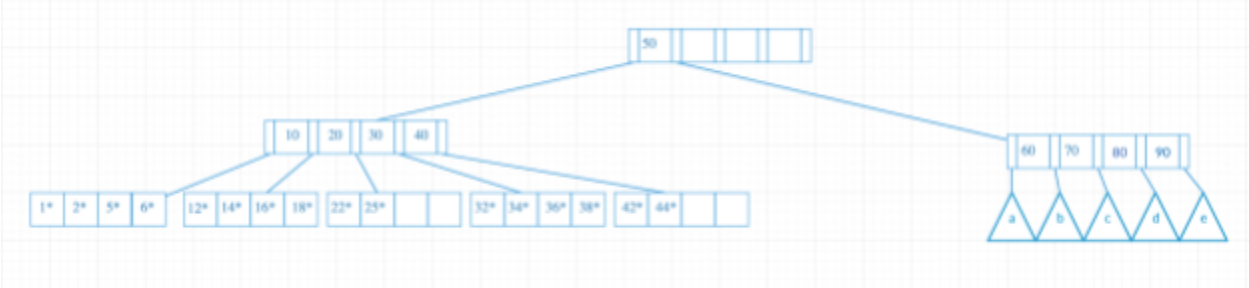


b) After insertion:

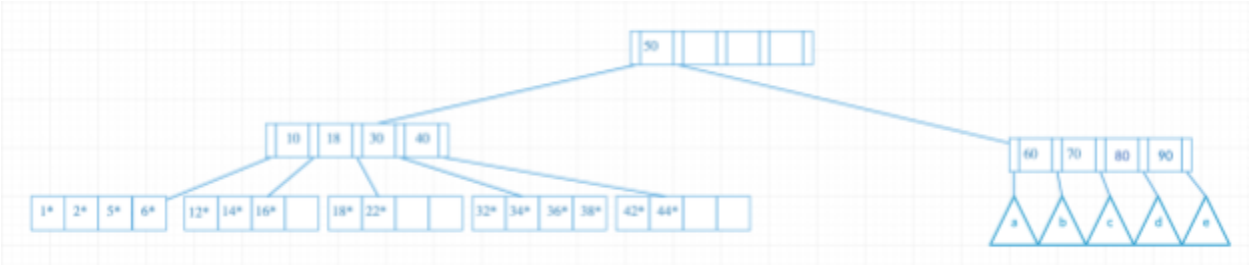


2)

a) Before deletion

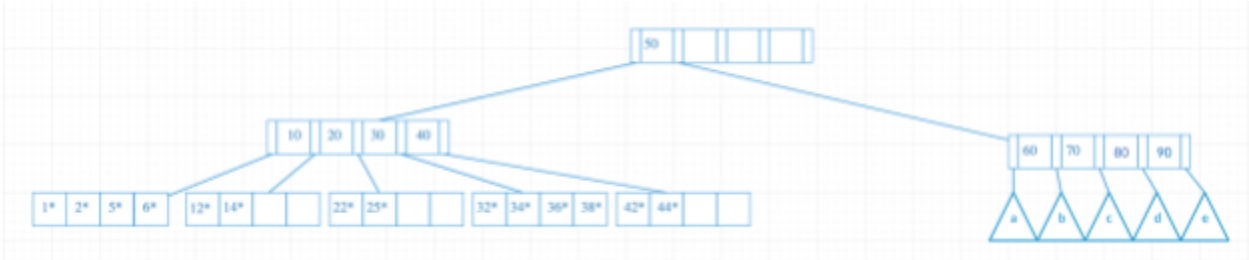


b) After deletion

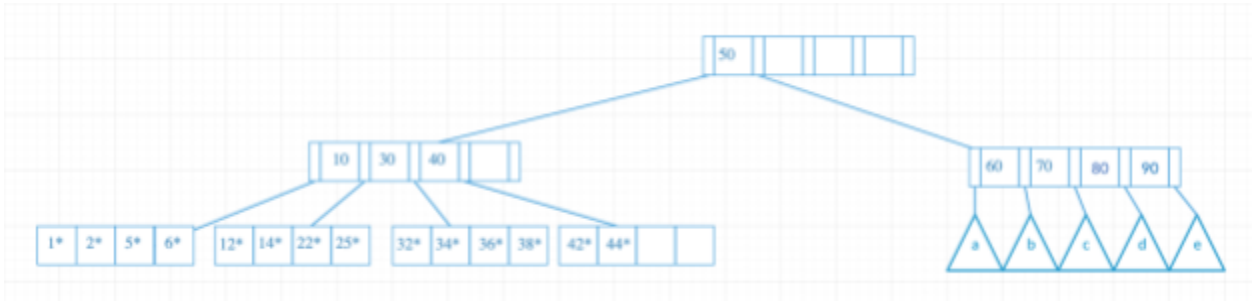


3)

a) Before Deletion



b) After deletion



Exercise 10.7

1) Inserting record by record using the B++ tree insertion procedure might be very expensive due higher I/O Costs for splitting and latching. The performance head is therefore very high. Moreover, each time a node splits, the data is redistributed evenly resulting in a 50% utilization. Hence, the space utilization is very low.
2) In bulk loading, the algorithm is built bottom-up. The performance is much better compared to repeated inserts because there are fewer I/Os. Moreover, in this technique you can control the fill factor and aim for higher storage utilization.

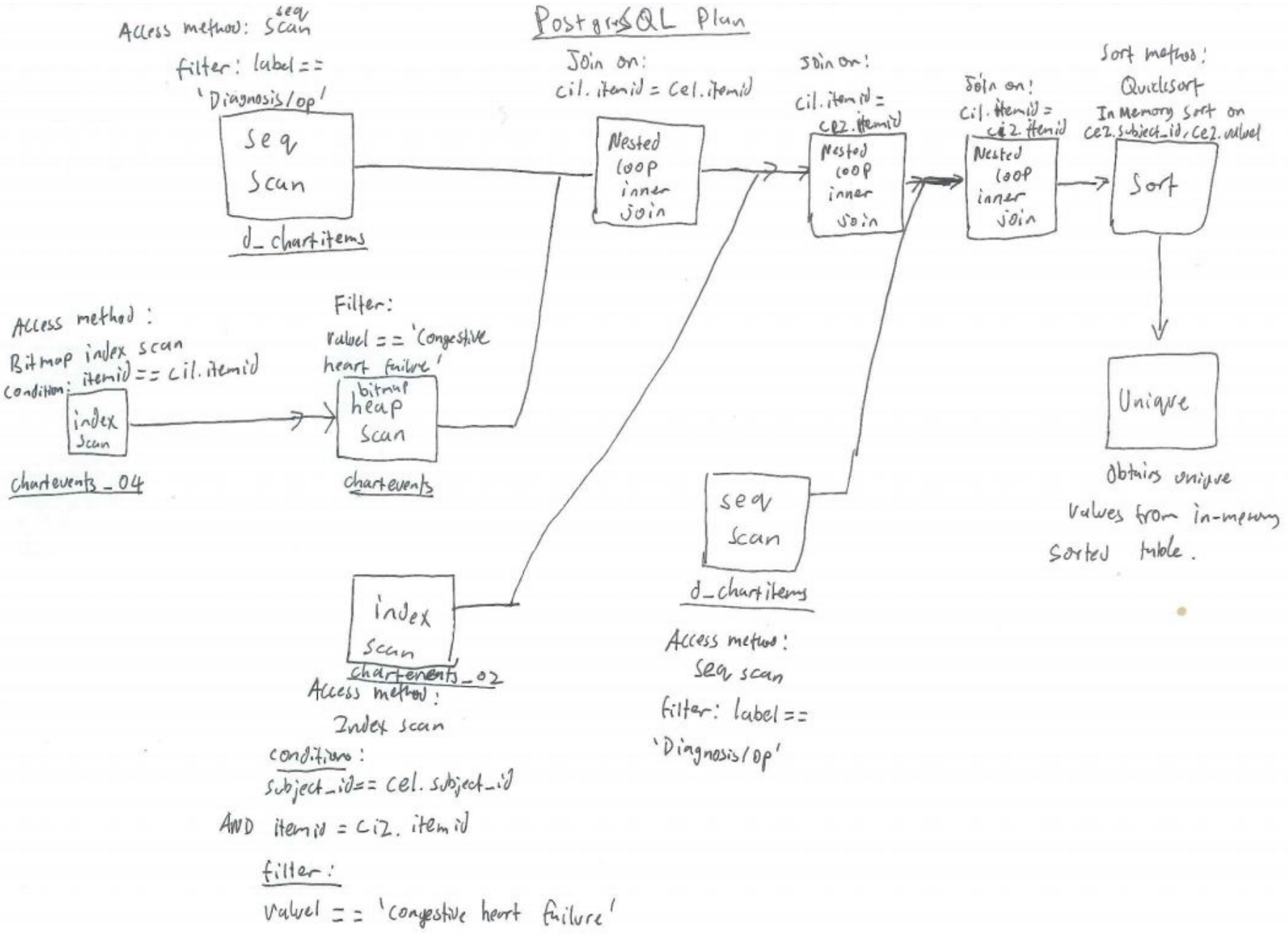
2) Query plans

2.1) **Query one:**

```
SELECT DISTINCT ce2.subject_id, ce2.value1
FROM chartevents AS ce1, d_chartitems AS ci1,
chartevents AS ce2, d_chartitems AS ci2
WHERE ce1.itemid=ci1.itemid
AND ce2.itemid=ci2.itemid
AND ce1.subject_id=ce2.subject_id
AND ci1.label='Diagnosis/op'
AND ce1.value1='CONGESTIVE HEART FAILURE'
AND ci2.label='Diagnosis/op'
AND ce2.value1!='CONGESTIVE HEART FAILURE';
```

a) Physical plan that PostgreSQL uses:

PostgreSQL Plan



b) Why do you think PostgreSQL selected this particular plan?
PostgreSQL In general follows a few heuristics and rules that minimizes cost. One such rule, for example, is that predicates are pushed down to the leaf nodes. We can see, for example, in the seq scan access of d_chartitems, a filter of label=='Diagnosis/op' is immediately applied while retrieving the rows, before joining more columns onto the row. This is done because it minimizes the number of rows that need to be passed up the plan.

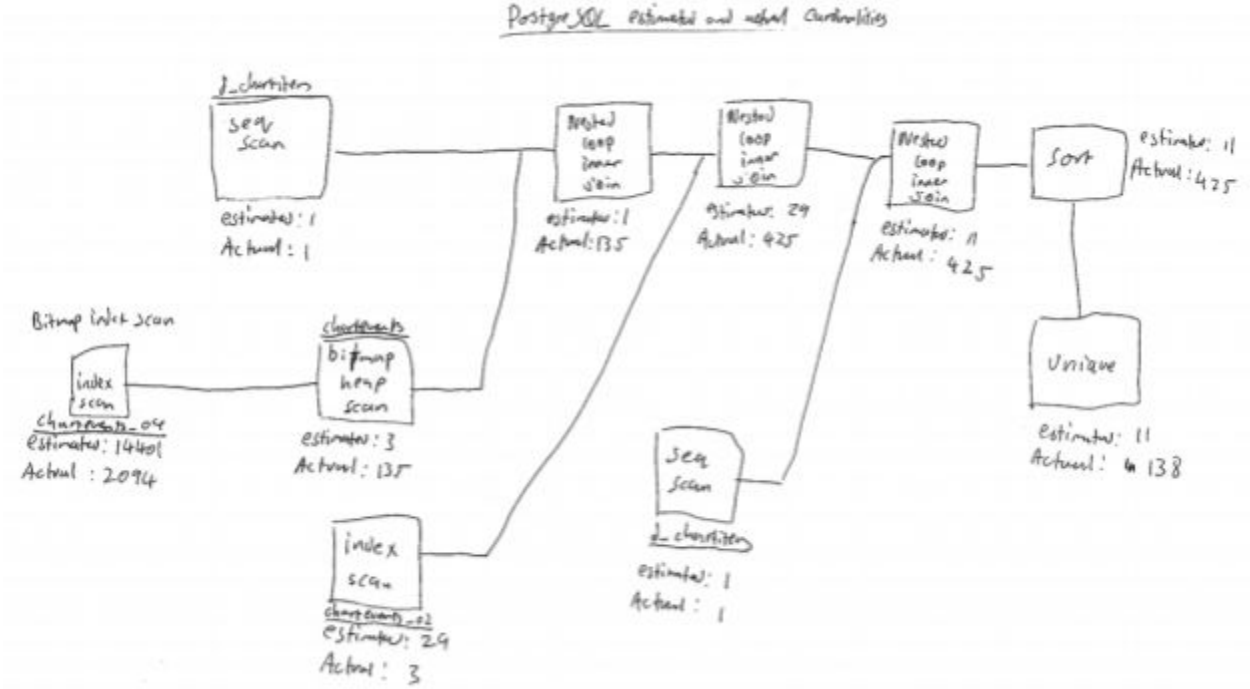
Other choices, for example the access method for retrieving chartevents_chartevents_o4 uses a bitmap index scan. This is chosen because the cardinality of chartevents itemid is low relative to the total number of records. There are 1133 unique item ids but 9548444 chartevents. Therefore, using a bitmap scan on the index of itemid (index number 4) will quickly allow us to discard invalid itemid rows without having to read the full tuple from disk.

The nested loop inner joins are also chosen instead of, for example, hash joins because based on heuristics such as the rejection ratio of filtering by strings. The plan assumes that the inputs are small (here we have cardinality 3 and 1). This case, a nested loop would be efficient as we do not need to allocate space in memory for a hash-join.

c) What does PostgreSQL estimate the size of the result set to be?
The estimated result set is size of 11 rows with a width of 11 bytes.

d) When you actually run the query, how big is the result set?
Actual size is 138 rows (width 11 bytes).

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?



The estimated differences can be seen in the above diagram. These estimates are carry-forward i.e estimates are based on the estimated previous outputs. For example, the bitmap index scan expected to find 14401 rows, but actually only found 2094 rows.

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?

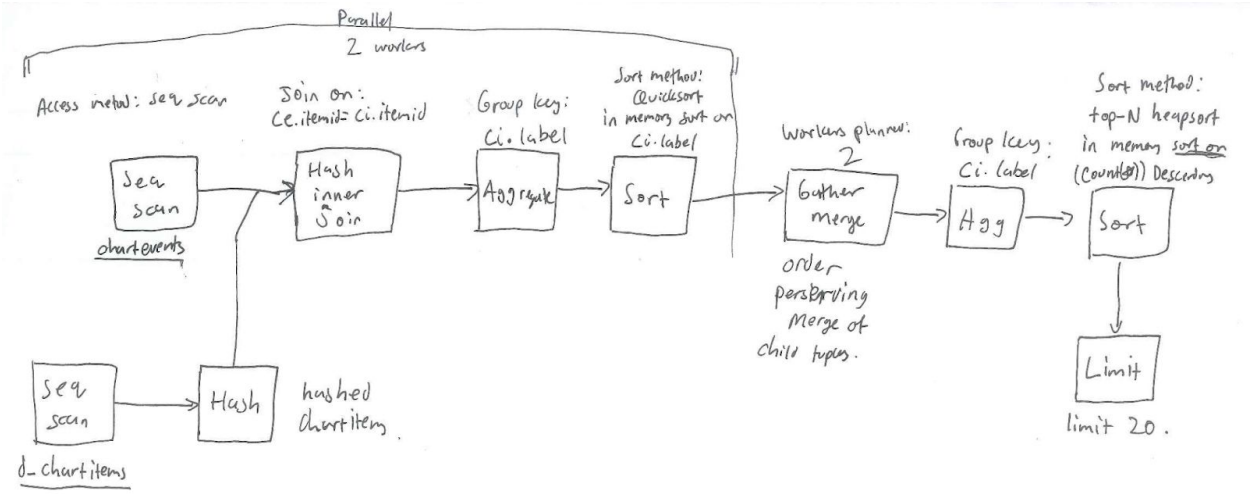
Indexes:
"chartevents_chartevents_o1" btree (cuid)
"chartevents_chartevents_o2" btree (subject_id, itemid)
"chartevents_chartevents_o3" btree (subject_id)
"chartevents_chartevents_o4" btree (itemid)
"chartevents_chartevents_o5" btree (icustay_id)
"chartevents_chartevents_o6" btree (charttime)
"chartevents_chartevents_o7" btree (cgid)

The plan successfully uses the two indexes on itemid and on subject_id which are what we expect for maximum access and join performance. Because of the low cardinality the join selection was good. Quicksort was also adequately selected because the table could fit in memory.

2.1) **Query two:**

```
SELECT ci.label, COUNT(*) AS freq
FROM d_chartitems AS ci, chartevents AS ce
WHERE ci.itemid=ce.itemid
GROUP BY ci.label
ORDER BY freq DESC
LIMIT 20;
```

a) Physical plan that PostgreSQL uses:



b) Why do you think PostgreSQL selected this particular plan?
Heuristics were applied in a number of cases to optimize the cost of the query.

For example, postgreSQL chose to use a hash inner join on the two tables chartevents and d_chartitems. It did this because it expected a high cardinality from both tables (due to the limited amount of predicates).

It also chose to use the output from the d_chartitems seq scan as the inner hashed table in the hash join. It chose this because this is the smaller relation of the two and so will not only fit in memory, but also consume less memory.

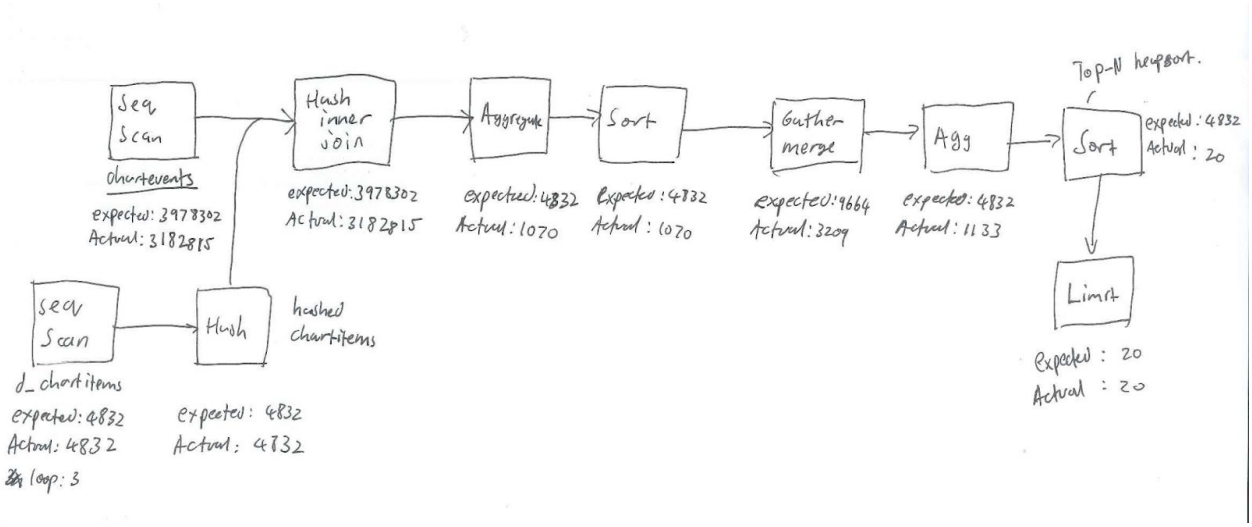
It selected a parallel gather-merge plan as well, because it realized that the aggregation could be broken up into sub-problems and later merged together. This takes advantage of parallel processing on my multi-core cpu, and can speed up the query.

A top-N heapsort was also used because the optimizer realized that the limit statement could be directly applied when sorting, to discard records that are clearly lower than the current top 20.

c) What does PostgreSQL estimate the size of the result set to be?
The estimated result set is a size of 20 rows with a width of 21 bytes.

d) When you actually run the query, how big is the result set?
Actual size is 20 rows (width 21 bytes).

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?



The estimated differences can be seen in the above diagram. These estimates are carry-foward i.e estimates are based on the estimated previous outputs. For example, the aggregation of joined inputs expected to find 4832 rows, but actually only aggregated 1070 rows.

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?

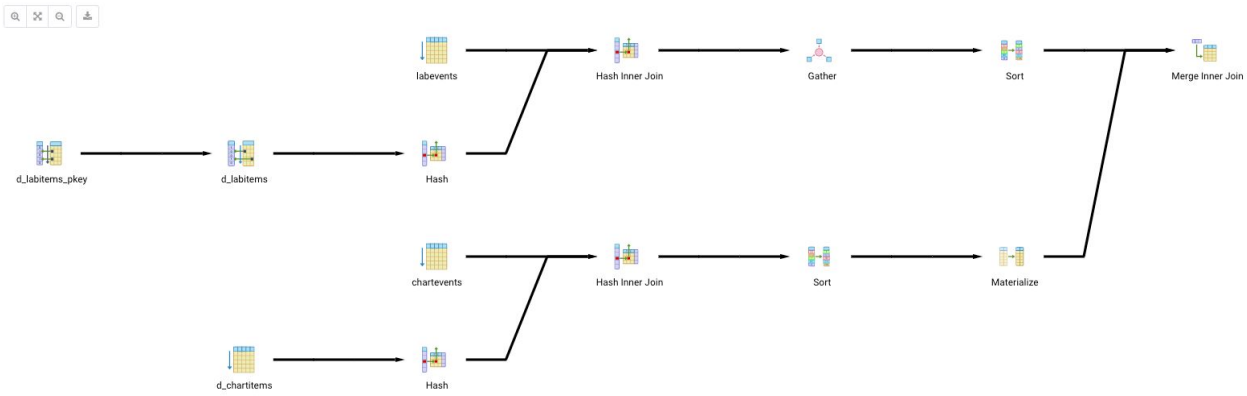
Although there is an index on chartevents(itemid) chartevents_o4, unfortunately there are not many predicates that can be pushed down in order to reduce the cardinality. Therefore, it would not be wise to have a nested loop join with chartevents on the inner loop. In this situation, the query plan chose a hash join utilizing d_chartitems on the inner hash which is correct because there are fewer values in d_chartitems so the hash table is small.

The plan also used parallel processing and a gather-merge which is a good choice because it allows multiple workers to aggregate and sort the label, which enables the gather-merge step to merge the solutions.

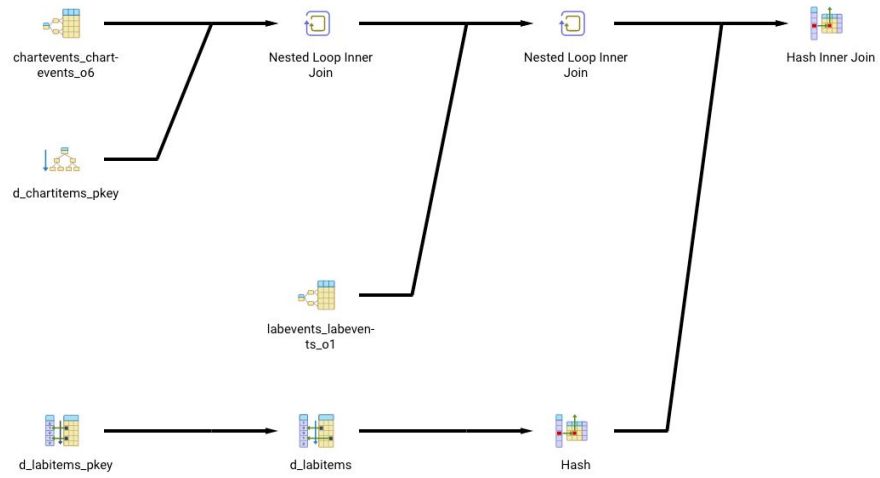
3) Estimating the cost of query plans

3.1) Explain the difference in query plans, and why the plans are different?

Plan 1:



Plan 2:



Why are the plans different?
The difference in the two query plans arise from different selectivity criteria in the two queries. In the first plan, ce.charttime > '3000-02-17 00:00:00' while in the second plan, ce.charttime > '3600-02-17 00:00:00'. When we do a quick search on table statistics for chartevents.charttime with the following query:

```
SELECT attname, histogram_bounds FROM  pg_stats
WHERE tablename='chartevents' AND attname='charttime';
```

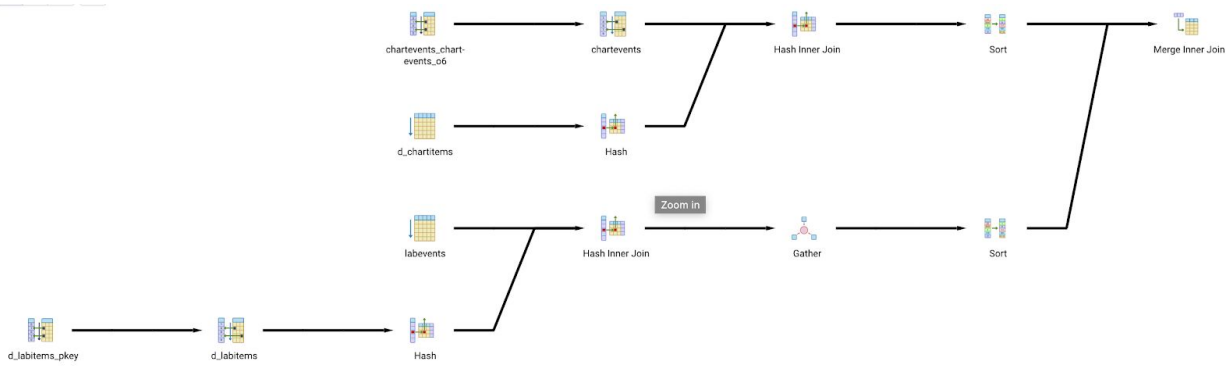
We get the histogram statistics that is recorded for charttime in this particular record. There were many buckets, but the final buckets in this range is
… "3108-05-07 02:45:00","3115-126-04-11 23:00:00","3502-11-09 18:00:00"}
Therefore, our selection predicate in the second query plan is outside of the range of the histogram statistics for the table chartevents. PostgreSQL recognizes that there will be very few matches when searching chartevents.

The resulting difference:
The first query utilizes a full seq-scan of the chartevents table, while the second query utilizes a index-scan on chartevents_o6 (index of charttime). The first query expects a large number of tuples to be scanned and populated so uses a full scan, while the second query effectively recognizes the fewer number of matching tuples in charttime.

The first query uses a hash join algorithm to match the scanned chartevents with d_chartitems on itemid. The reason for this is because it assumes that a large number of chartevents will need to be joined. In this situation, a hash join minimizes the cost of the inner loop by building a hash table on d_chartitems. The second query has a few chartevents, and therefore uses a nested inner loop join because it reduces the cost of building a hash table and can directly access the matching d_chartitems from the index.

The first query uses parallel hash joins directly on labevents and chartitems. It then merges the outputs of the labevent workers. It joins the outputs from labevents and chartevents at the end. On the other hand, the second query builds a left-deep tree only, and only joins when a match is found. The second query is able to do this because the initial set is small, and fits directly in memory.

3.2) b)
What is PostgreSQL doing for this query? How is it different from the previous two plans that are generated?
Plan 3:



Postgres is recognizing the much higher selectivity criteria in plan 3 compared to plan 2, but lower selectivity than plan 1. It then makes the appropriate choices to improve performance. In general, the query plan 3 is more similar to plan 1, because it is comparatively more selective than plan 2.

The change that has been made to plan 1 is that rather than doing seq-scan of chartevents, plan 3 uses an index "chartevents_chartevents_o6" btree (charttime). Using this index is smart because it expects a small ratio of matching rows after applying the filter. (but not small enough to warrant nested loop joins as in plan 2)

c)
Run some more EXPLAIN commands using this query, sweeping the constant after the '>' sign from 3000 to 3600. For what value of charttime 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 charttime for which PostgreSQL switches query plans. [2 points]

Switch 0 plan (Plan 1): 3000 - **3363**;
Switch 1 (index on chartevents): 3364 - **3443**
Switch 2 (Parallel chartevents) 3444 - **3489**
Switch 3 (No intermediary materialization) 3490 - **3496**
Switch 4 (No parallel chartevents) 3497 - **3499**
Switch 5 (Plan 3, chartevents becomes inner in join) 3500 - **3502**
Switch 6 (Plan 2, nested loop) 3503 - **3600+**

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

The plans switch because as the selectivity criteria of charttime increases, the values of certain variables e.g selected-tuples/total-tuples or intermediary table size changes. Since different types of operations have different overheads and costs, the most optimal set of operations (as predicted by postgres) will also change. Take the below instances for example:

The first switch from the plan 1 to the other plan that utilizes an index on chartevents is because postgres estimated that that 1277910 chartevents will be matched for charttime > 3363 and

1272098 for charttime > 3364. Postgres decides that random accessing of 1272098 records from disk is faster than a full scan of 9548444 chartevent tuples. On the other hand, it believes that a full scan is faster than random accessing of 1277910 records;

The fifth switch to chartevents being moved from the outer join to inner join. This happens because of the following estimates for selectivity of chartevents:
charttime> 3499 estimates that sorted and joined chartevents is 54020 rows.
charttime> 3500 estimates that sorted and joined chartevents is 39532 rows.
Since labevents selects approximately 48305 rows, when charttime>3499, postgres estimates that chartevents is the larger set, and because of a rule where the smaller relation is placed as the inner join, postgres chooses labevents. However, when charttime>3500, chartevents is subsequently smaller, and is hence placed as the inner relation. That being said, the cost should be similar in both situations.

The final switch from query 3 to query 2 is because of the following selectivity:
charttime > 3502 estimates 10556 rows
charttime > 3503 estimates 1 row.
This is likely because the histogram statistics for the table chartevents ends at 3502. Therefore, postgres estimates that with 1 row from chartevents, a nested inner loop would be much faster than building a hash table on chartitems for a single chartevent.

e)
Suppose the crossover point (from the previous question) between queries 2 and 3 is charttime23. 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 0 3363 (1693.617 ms) - 3364 (892.697 ms)
Switch 1 3443 (372.161 ms) - 3444(359.333 ms)
Switch 2 3489 (157.766 ms) - 3490 (151.318 ms)
Switch 3 3496 (140.652 ms) - 3497(127.390 ms)
Switch 4 3499 (128.757 ms) - 3500(101.828 ms)
Switch 5 3502 (84.734 ms) - 3503 (1.093 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 overestimates/underestimate 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

The first plan does not appear to be the most optimal place to switch. There is a large decrease in time between 3363 and 3364, when the plan changes to an index on chartevents rather than a seq-scan. This suggests that the switch could have happened earlier, and perhaps sped up the query.

A seq scan would require scanning the whole table. This would have a cost of 9555810. Assuming that it takes 3 IOs to search the unclustered index, doing the index scan would result in 1277910 * 3 * read time which may vary depending on hdd/ssd. To estimate this, If we look at the EXPLAIN ANALYZE of the previous query, we recognize that seq scan takes approximately 1226.803 ms. Bitmap scan of 1277910 matching values takes 296.952 ms. We could have had 4x the selectivity and bitmap scan would have been faster than a seq scan. Therefore, we could have switched plans at around 3000, which has a selectivity of 4756731 < 1277910*3.

The switch point between 3502 and 3503 also seems to be overestimated. This reason is because postgres estimates that 3502 would select 10556 rows based on the table statistics. But in reality it actually selected 679 rows. Since we are near the tail end of the table's histogram, it did not estimate the selectivity accurately. If we use a nested loop join, it would need to be faster than the time it takes to build the inner hash table. It's likely that the switch point could have happened at 3501, which has an actual selectivity of 3781 rows. But overall it the switch is quite optimal based on the known table statistics.

The other switch locations seem to have been adequately selected by the query plan as optimal switch points based on the run time differences.

4) Access methods

a) Things to note as assumptions:

1. In the events table, eid is a primary key (unique). Therefore, there can only be one record at each leaf node. So the number of leaf nodes is equal to the size of the events table
2. I calculate the depth of the B+ tree using the size of a page, the size of a pointer and the size of an eid. I used this formula: $d = log_{\frac{n}{2}}(L)$, where d is the depth of the tree, n is the number of pointers ($\frac{4KB}{(8B)(2)}$) and the L is the number of leaf nodes.
3. There are 3 I/Os to search for the Index (assuming the)

Total time = search time in the B+ reading the tuple from disk

= ((depth of tree -1)* (disk access + time t0 read a 4KB page)) + (disk access + reading time of tuple from disk)

= $((4)(10^{-3} + \frac{1}{10^{-8}}(4 \times 10^{3}))) + (10^{-3} + \frac{1}{10^{-8}}(8+8+20+8+8))$

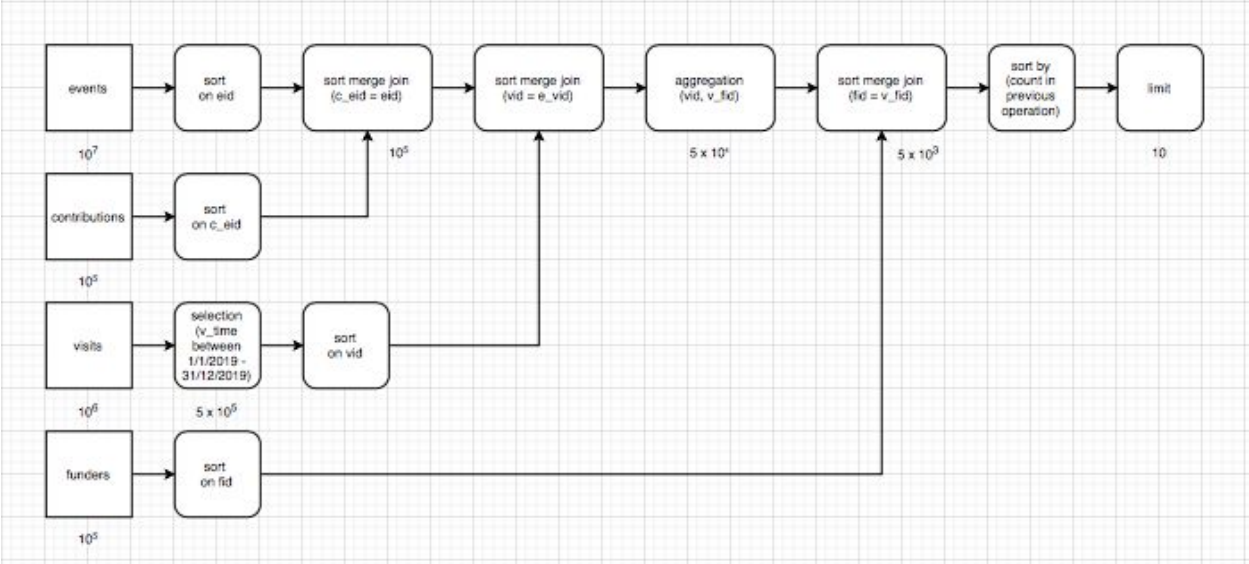
= $0.4016 + 0.00000052 = 401ms$

b) The difference between a clustered B+ tree and an unclustered B+ tree is that in the former, the data is stored contiguously in disk and in the latter it is not. Therefore, the leaf nodes have

pointers to random locations in the disk. Clustered B+ trees are most useful in range search queries. However, in this query only one record is being retrieved (because eid is a primary key), there is no difference between using a clustered and an unclustered B+ tree. Bth are going to make 4 I/Os (including the root) to search the B+ tree and the final I/0 to read the tuple at the location the pointer at the leaf node points to.
Runtime = 401ms

C & d)

Assuming that all sorting and aggregation operations happen in memory, we think the merge join would be a better joining algorithm. The reason is because we are assuming that sorting happens in memory; we have a large and unlimited memory and tables that are relatively small in size.



e)
Assumptions:

- For calculating the runtime of this, I am assuming that aggregation and sorting are done in memory and hence by the CPU. So the only runtime costs estimated here will be for joins.
- No information regarding how data is stored in the disk is provided. Disk access time is small, so it will be ignored.

Total runtime = (cost of joining the events and contributions table) + (cost of joining the resulting table and the visits table) + (cost of joining the resulting table and the funders table)

cost of joining the events and contributions table:

$$= \frac{10^7 \times (8+8+8+8+20)}{10^8} + \frac{10^5 \times (8+8+8+8)}{10^8} = 5.2s + 0.032s = 5.232s$$

cost of joining the resulting table and the visits table:

$$= \frac{10^6 \times (8+8+8+8+8)}{10^8} + \frac{10^5 \times (8+8+8+8+8+8+20)}{10^8} = 0.40s + 0.068s = 0.468s$$

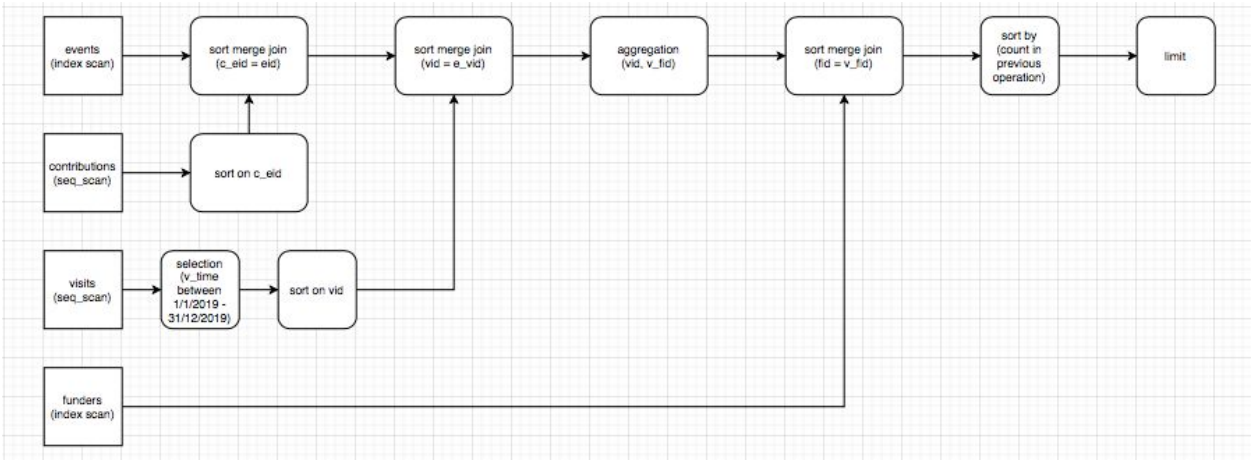
Note: when calculating the tuple size for the merged table, I don't count the headers of the each table, I remove join key and I add 8 bytes for a new header

cost of joining the resulting table and the funders table:

$= \frac{10^5 \times (100+8+8)}{10^8} + \frac{5 \times 10^3 \times (8+8+8+8+8+8+8+8+8+20)}{10^8} = 0.116s + 0.0046s = 0.1206s$

Total runtime = $5.232s + 0.468s + 0.1206s = 5.8206s$

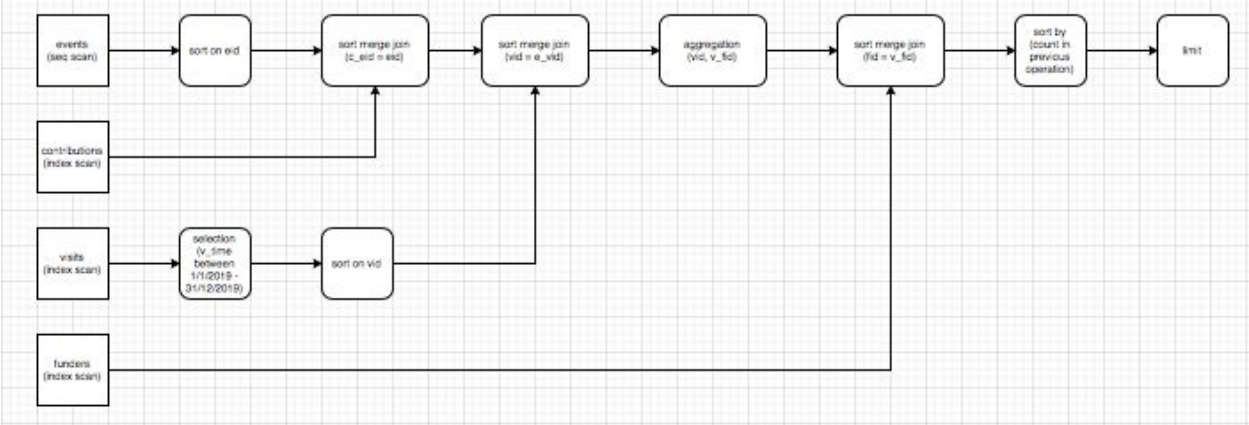
f)



Notes:

- The initial scan of the events table will use the clustered (already sorted) B+ tree index on eid to perform sort merge join with the contributions table (does not have an index on c_eid and needs to be sorted). Because the buffer pool is unlimited and the table sizes are small, it will be assumed that sorting happens in memory and there are no significant sorting costs that need to be included in the calculations. Runtime cost for this join:
  - $= \frac{10^7 \times 8}{10^8} + \frac{10^5 \times (8+8+8+8)}{10^8} = 0.8s + 0.032s = 0.832s$
- As for the visits table, there is no index on v_time. Because selection is pushed down in the tree, we will have to do a sequential scan, apply selection and sort it on vid. The result of the previous join is already sorted. Although there is a clustered index on e_vid, using this on a already sorted merged table can make incur more costs (will behave like an unclustered index because the e_vids in events are a subset of the the indices in the B+ tree)
  - $= \frac{10^6 \times (8+8+8+8+8)}{10^8} + \frac{5 \times 10^5 \times (8+8+8+8+8)}{10^8} + \frac{10^5 \times (8+8+8+8+8+8+20)}{10^8} = 0.40s + 0.200s + 0.068s$
  - $= 0.668s$
- The index on fid can be used to scan the funders table in a sorted fashion and then merge it with the resulting merged table from the previous operation:
  - $= \frac{10^5 \times 8}{10^8} + \frac{5 \times 10^3 \times (8+8+8+8+8+8+8+8+8+20)}{10^8} = 0.008s + 0.00460s = 0.0126s$
- Sorting and aggregation happens in memory and so they are ignored
- **Total runtime =** $0.832s + 0.668s + 0.0126s = 1.513s$

g)



- The events table does not have an index on eid. So, the table will be sequentially scanned and sorted on eid. There is a clustered index on the contributions table which can be used to perform a sort merge join. With sorting costs ignored, the runtime of the first join here is:
  - $= \frac{10^7 \times (8+8+8+8+20)}{10^8} + \frac{10^5 \times 8}{10^8} = 5.2s + 0.008s = 5.206s$
- Because there is a clustered index on v_time, an index scan can be used to perform the selection very quickly. The resulting table will be sorted on vid. This is then merged with the resulting table from the previous operation:
  - $= \frac{10^6 \times (8)}{10^8} + \frac{5 \times 10^5 \times (8+8+8+8+8)}{10^8} + \frac{10^5 \times (8+8+8+8+8+8+20)}{10^8} = 0.08s + 0.200s + 0.068$
  - $= 0.348s$
- The clustered index on fid allows for another merge sort join with the resulting table from the aggregation operation:
  - $\frac{5 \times 10^3 \times (8+8+8+8+8+8+8+8+8+20)}{10^8} + \frac{10^5 \times 8}{10^8} = 0.00460s + 0.008 = 0.0126s$

Total runtime = $5.208s + 0.348s + 0.0126s = 5.569s$

h) Yes, the query can be flattened.

SELECT name, count(*) from funders, visits, events, contributions
WHERE v_fid = fid,
AND vid = e_vid
AND c_eid = eid
AND v_time BETWEEN '1/1/2019' AND '31/12/2019'
GROUP BY vid, v_fid
ORDER BY COUNT(*) DESC
LIMIT 10