PROBLEM 1

a) With the given assumptions, Scenario 1 has no space overhead.

The space for the B-tree index is dominated by the leaves, which use (essentially) $8 + 4 = 12$ bytes per key, i.e. $12N$ bytes in total.

b) The bound for insertion in a hash table is $O(1)$ I/Os, and in a B-tree $O(\log_B N)$ I/Os. (dominates $O(1)$ I/Os)

- In both cases, a point query can be done using the hash table in $O(1)$ I/Os.

- A range query can be implemented as $r$ point queries in $O(r)$ I/Os, or as a range query in the B-tree in $O(\log_B N + rB)$ time, plus $O(r)$ time for following pointers.

c) Scenario 1 is best when:

$$x + y + zr < x \log_B N + y + z(\log_B N + l)$$

$$z(r - \log_B N - l) < x(\log_B N - 1)$$
PROBLEM 2

a) \( R_1, R_2, R_4, A, \overline{B}, \overline{E}, R_3, R_6, R_7 \)

unless search alg. knows match there is at most

b) This is essentially the "Planar Point Location" problem. The only difference is that we discussed that problem for a polygon split in parts using line segments. We can reduce this case to that one by arbitrarily connecting the triangles with lines. This gives a solution with queries in \( O(\log N) \) I/Os.

c) Consider the following example:

\[ \text{N TRIANGLES} \]

\[ \text{POINT QUERY} \]

Since the bounding box for any subset of the triangles contains the point, a search must visit all nodes in the R-tree to see that the point is in no triangle.
**PROBLEM 3**

| Signature 1 | Est. $|R_4 \times R_3|$ | Est. $|R_2 \times R_3|$ |
|-------------|-----------------|-----------------|
|             | 900,000         | 700,000         |
| Signature 2 | 1,100,000       | 1,050,000       |
| Signature 3 | 630,000         | 1,980,000       |

$|R_4 \times R_3| = |R_2 \times R_3| = |R_4| |R_3|$, so no estimate is needed.

b) 
Est. $|R_2 \times R_2|$: $100 \cdot 100^2 = 1,000,000 \Leftarrow 12$ Bytes/tuple; 12,000,000 bytes

Est. $|R_2 \times R_3|$: $200 \cdot 100^2 = 2,000,000 \Leftarrow 12$ Bytes/tuple; 24,000,000 bytes

Est. $|R_4 \times R_3|$: $2,000 \cdot 100^2 = 20,000,000 \Leftarrow 16$ Bytes/tuple; 320,000,000 bytes

c) Since the join algorithm is linear in the size of its input and output, the total complexity is linear in the size of the relations and final result (fixed), plus the size of the intermediate results. Hence it is optimal to minimize intermediate results.

- If we take 4 relations, with no common attributes and compute the natural join (Cartesian product) using a left-deep join tree there is an intermediate result of size $N^3$. On the other hand, this expression only has intermediate results of size $\approx 2N^2$: $(R_4 \times R_2) \times (R_3 \times R_4)$. 
PROBLEM 4

a) LOCK ON	OBTAINED	RELEASED
       BEFORE	        AFTER
FREEBUSINESSSEATS (x) 1	          7
UPGRADES (x)              2	          6
SEATS (x)                  3	          7

b) TO MINIMIZE LOCKING TIME, AT LEVEL "READ COMMITTED"
THE TRANSACTION WOULD CARRY OUT 4 AND 5 WITH
A SHARED LOCK ON Seats. Thus, TRANSACTION 1 MAY
OBTAIN AN EXCLUSIVE LOCK ON Seats UP TO 6 WHERE
TRANSACTION 2 REQUESTS AN EXCLUSIVE LOCK.

c) • IF TRANSACTION 1 HAS THE SMALLEST
   TIMESTAMP, IT IS ROLLED BACK WHEN TRYING TO UPDATE
   Seats 1 because it has a larger read time stamp.

   • IF TRANSACTION 1 HAS THE LARGEST TIMESTAMP,
   IT WILL WAIT FOR T2 TO COMMIT BEFORE IT CAN
   ABORT UPDATE. LATER, WHEN T2 WANTS TO UPDATE WHAT
   T1 HAS READ, IT WILL BE ROLLED BACK AND
   T1 WILL COMPLETE.
PROBLEM 5

a) ITEMS FOUND IN FIRST PASS: milk, diapers, eggs, beer, chips

COUNT MATRIX FOR SECOND PASS:

<table>
<thead>
<tr>
<th></th>
<th>milk</th>
<th>diapers</th>
<th>eggs</th>
<th>beer</th>
<th>chips</th>
</tr>
</thead>
<tbody>
<tr>
<td>diapers</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eggs</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beer</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chips</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

FOUND PAIRS ARE {milk, diapers}, {eggs, diapers}, AND {beer, chips}.

b) milk \rightarrow diapers, \frac{3/4 - 5/8}{1/8} = 3/4
   diapers \rightarrow milk, \frac{3/5 - 4/10}{1/10} = 3/5
   eggs \rightarrow diapers, \frac{3/3 - 5/8}{3/8} = 3/3
   diapers \rightarrow eggs, \frac{3/5 - 3/8}{9/40} = 3/5
   beer \rightarrow chips, \frac{3/4 - 3/8}{3/8} = 3/4
   chips \rightarrow beer, \frac{3/3 - 4/8}{1/2} = 3/3

ANNOTATED WITH BUCKET NUMBERS

C) FOR THE FIRST PASS, SORT ALL ITEMS SO THAT THEY CAN BE COUNTED EASILY: O(N log N) I/Os.

NOW SORT THE HIGH SUPPORT ITEMS BY THEIR BUCKET NUMBERS - THIS GIVES THE ORIGINAL BUCKETS WITH LOW SUPPORT ITEMS REMOVED. FOR EACH BUCKET, GENERATE ALL PAIRS OF ITEMS, AND SORT THE PAIRS OF ALL BUCKETS SUCH THAT IDENTICAL PAIRS BECOME ADJACENT. FINDING HIGH-SUPPORT ITEMS CAN NOW BE DONE EASILY IN A SINGLE SCAN.