Part 1:

1. Former-employee records:
   - Add a secondary index on Start Date Reasoning: This index will support both queries efficiently. The first query directly uses the Start Date, and the second query can use the index for the initial filtering on Start Date before checking the End Date condition.
2. Student grades table (first set of queries):
   - Add a secondary index on Grade Reasoning: Both queries filter based on the Grade column. An index on Grade will allow for quick lookups without scanning the entire table.
3. Student grades table (second set of queries):
   - Add a secondary index on className
   - Add a composite secondary index on (className, Grade) Reasoning: The first query needs to order by className, so an index on className will help. The second query filters on both className and Grade, so a composite index will be most efficient.
4. Chess database queries:
   - Add a secondary index on Players.Elo
   - Add a secondary index on Games.WhitePlayer Reasoning: The first query filters on Elo, so an index on Elo will improve performance. The second query joins Players and Games on WhitePlayer, so an index on WhitePlayer in the Games table will speed up the join operation.
5. Public Library database (first set):
   - None Reasoning: A natural join typically uses the primary keys of the tables involved. Since we already have primary indexes on both tables, no additional indexes are needed for this query.
6. Public Library database (second set):
   - Add a secondary index on CheckedOut.CardNum Reasoning: The first query filters on CardNum in the CheckedOut table, so an index on this column will improve performance. The second query, like in the previous case, uses a natural join on primary keys, so no additional index is needed for it.
7. Public Library database (third set):
   - Add a secondary index on Inventory.TitleID Reasoning: The LINQ query is joining Titles with Inventory. Assuming TitleID is the foreign key in the Inventory table referencing the Titles table, an index on Inventory.TitleID will improve the join performance. This is especially important since we're selecting all Serials for each Title, which suggests a one-to-many relationship.

Part 2:

Students table:

1. Number of rows in the first leaf node of the primary index before splitting:
   - Each row occupies: 4 (int) + 10 (varchar(10)) + 4 (int) + 1 (char(1)) = 19 bytes
   - 4096 / 19 = 215.57 Therefore, 215 rows can be placed in the first leaf node before it splits.
2. Maximum number of keys in an internal node of the primary index:
   - Each key occupies: 4 (int) + 10 (varchar(10)) = 14 bytes
   - 4096 / 14 = 292.57 An internal node can store a maximum of 292 keys.
3. Maximum number of rows if the primary index has a height of 1:
   - An internal node can have 293 child nodes (292 keys + 1)
   - Each leaf node can store 215 rows
   - Maximum rows = 293 * 215 = 62,995 rows
4. Minimum number of rows if the primary index has a height of 1:
   - The root node (only internal node) must have at least 2 child nodes
   - Each leaf node must be at least 50% full, i.e., 108 rows (half of 215, rounded up)

- Minimum rows = 2 * 108 = 216 rows

5. Maximum number of entries in a leaf node of the secondary index on Grade:
    - Each entry occupies: 1 (char(1)) + 4 (int) + 10 (varchar(10)) = 15 bytes
    - 4096 / 15 = 273.06 A leaf node in the secondary index can store a maximum of 273 entries.

Another table:

6. Maximum number of leaf nodes in the primary index if the table contains 48 rows:
    - Rows per leaf node: 4096 / 128 = 32 rows
    - Minimum leaf nodes needed for 48 rows: 48 / 32 = 1.5 Therefore, the maximum number of leaf nodes is 2.
7. Minimum number of leaf nodes in the primary index if the table contains 48 rows:
    - Each leaf node must be at least 50% full, i.e., 16 rows
    - 48 / 16 = 3 Therefore, the minimum number of leaf nodes is 3.