Takeaway Sheet 7 - Jonathan Jacobs

Question 1

The best partitioning scheme for large range scans over a single column, like "Salary" in the provided query, would be **Range Partitioning** on the "Salary" column. This strategy ensures that the data is split based on salary ranges, making queries that filter by specific ranges more efficient since only relevant partitions are accessed.

Question 2

hash-partitioned?

Shuffling is **not necessary** when aggregating over local data that is already hash-partitioned on the grouping attribute(s). In this case, all relevant data for each group is already located on the appropriate node due to the partitioning strategy. This eliminates the need to redistribute or shuffle data across nodes, allowing local aggregation to be performed directly.

range-partitioned on A?

Shuffling is **not necessary** when aggregating over local data that is already range-partitioned on column "A." With range partitioning, each node contains all the relevant data for a particular range of "A" values, which ensures that the grouped data is already co-located on each node. Therefore, the aggregation can be done locally within each partition, and no further redistribution (shuffling) is required across nodes.

Question 3

An RDMS would choose a broadcast join over a partitioned hash join in these situations:

1. **Smaller Table Size**: One table is much smaller and can fit into each node's memory, minimizing data movement.
2. **Skew Prevention**: The larger table has skewed data that could lead to uneven distribution, which a broadcast join prevents.
3. **Performance Optimization**: Broadcasting the small table avoids excessive data shuffling, improving performance despite higher memory usage.

Question 4

How would you implement argmax in MapReduce?

Implement your map() function

```python
def map(key, value):
    """
    This map function emits the key and its associated value.
    :param key: A unique identifier or record name.
    :param value: The value to compare for finding the maximum.
    """
    emit(key, value)
```

Explanation:

- The `map()` function receives a key-value pair and emits it unchanged.
- The reducer will then find the key associated with the maximum value. The `reduce()` function will receive groups of intermediate key-value pairs, with each group having a unique key. To implement `argmax`, the reducer will identify the key that has the highest value among all the inputs.

Implement your reduce() function

```
def reduce(key, values):
    """
    This reduce function identifies the key with the highest value.
    :param key: A placeholder key grouping all intermediate values 
together.
    :param values: An iterable of values associated with the key.
    """
    # Find the maximum value from the given values
    max_value = max(values)
    emit(key, max_value)
```

Explanation:

- The `reduce()` function aggregates all intermediate values for a given key.
- It calculates the maximum value among those values and emits the key-value pair, where the value is the maximum.

Question 5

How would you implement an inner join in MapReduce?

Implement your map() function

```
def map(key, value):
    """
    This map function emits a tagged value for each record to indicate the 
source table.
    :param key: The join key, usually a shared attribute between tables.
    :param value: A string representing the data record.
    """
    # Extract table name from value to identify the source
    table_name = value.split(",")[0]  # Adjust based on your data 
structure
    # Emit a tagged value: (table_name, value)
    emit(key, (table_name, value))
```

Explanation:

- Input: The function takes a key (join attribute) and a value (record) as inputs.
- Output: It emits the join key and a tuple `(table_name, value)`, where `table_name` is a tag indicating the source table.
  The reducer will then group all records by their join key, allowing the join logic to be implemented based on table tags.

Implement your reduce() function

```python
def reduce(key, values):
    """
    This reduce function performs an inner join for records sharing the same join key.
    :param key: The shared join key across both tables.
    :param values: An iterable containing tagged records from both tables.
    """
    # Separate records by their table tags
    table1_records = [v[1] for v in values if v[0] == 'table1']
    table2_records = [v[1] for v in values if v[0] == 'table2']

    # Perform an inner join by combining each pair of records with the same key
    for record1 in table1_records:
        for record2 in table2_records:
            # Emit the join key with the combined record from both tables
            emit(key, (record1, record2))
```

Explanation:

- **Input**: The function receives a key (shared join key) and a collection of tagged values.
- **Processing**:
  - Separates values into different lists based on their table tags.
  - Combines each pair of records from both tables that share the same key.
- **Output**: Emits the join key with the combined record (a tuple) containing matching pairs.