1. Relational Algebra is a query language that process one or more relations to define another relation. It involves operators to perform them. There are two kinds of operators: Unary and Binary.
2. The set of Binary operators utilize the pull-based architecture; thus, they don't refer to the source instead it refers to the intermediate data i.e., can be Hash.
3. The set of Binary operators include:
    a. Set Intersection
    b. Equi Join
    c. Difference
    d. Cartesian Product.
4. Set Intersection: This operator implements an intermediate has on one of the relation and lookup is performed on the key. The significance of this operator is to provide the tuples from the relations which are in common.
5. Using the smallest relation, makes us use less memory and ultimately speeds the process of retrieval of data.
6. The process of set intersection is an asymmetric version of the algorithm.
7. Next, Equi-Join operator which takes up two relations R, S namely and builds up a hash on S. For every tuple t in R, a lookup operation is done on the hash of S.
8. The process is inefficient, as it must follow tuple by tuple. Thus, we need to find ways to optimize it based on the feature of scalability and reliability.
9. Using pipes helps us to achieve some state of parallelism and can achieve
    a. Code to stop by capturing the position and to start again where it has left.
    b. Also, might cause overhead for lots of context switching.
    c. Can achieve some state of parallelism.
10. We also must synchronize the tuple by tuple in the relations. Bundling the tuples together would eventually reduce to some extent. Also, due to various architectural reasons it also sometimes cannot achieve performance,
11. Another way to find the efficient way is to work is nested loop joins for cartesian product. Several versions are brought to light to tackle the situations.
    a. Version-1:
        i. Initial phase of development – reads them tuple by tuple. Thus, forming the inner loop for the relation to have B(s) operations for each tuple in R. Thus, leading to total N(R)*B(s)
    b. Version-2:
        i. With the challenges from the pipeline version, the ides of grouping tuples to form blocks thus reduces the disk IO operations.
        ii. Grouping tuples of R together and each tuple is checked over the tuple in the block of S.
        iii. Reducing the time complexity to B(R) + B(R) * B(S)
        iv. Thus, with the advent of memory there is a possibility to load the data and trade the cost with complexity, giving rise to Block Nested Loop Join.
    c. Version-3:
        i. This process involves loading of the relations R, S into memory based on the smaller size, giving rise to form fragments to load up the several blocks of relations.
        ii. This in turn reduces the time complexity proportional to the size of memory. B(R) +(B(R) * B(S))/M. If anyone of the relation is perfectly fitting into the memory. It would drop down the complexity to B(R) + B(S).