Database Systems Project:

Given the Relation scheme R (A1 , A2, …. An, B1* , B2*, …. Bm*). It has n number of A and m number of B* attributes where n, m>=0 and attribute Ai has any datatype and attribute Bj* has any data type, but Bj* is a set of values.

Let us consider a case where n=2 and m=2 and perform the JOIN query on R1(A1,A2,B1,B2) and R2(A1,C2,D1,D2). Here B1,B2,D1,D2 attributes are array type in postgresql. Here columns with array data type can have 1M, 1B or any possible number of values. These values could be integer, character or any data type.

Before performing the operation, I observed the storage structure of postgreSQL. PostgreSQL uses a fixed page size (commonly 8 kB) and does not allow tuples to span multiple pages. Therefore, it is not possible to store very large field values directly. When a row is attempted to be stored that exceeds this size, TOAST basically breaks up the data of large columns into smaller "pieces" and stores them into a TOAST table. Each table I create has its own associated (unique) TOAST table, which may or may not ever end up being used, depending on the size of rows inserted. The mechanism is accomplished by splitting up the large column entry into 2KB bytes and storing them as chunks in the TOAST tables. It then stores the length and a pointer to the TOAST entry back where the column is normally stored. Because of how the pointer system is implemented, most TOAST' able column types are limited to a max size of 1GB, but as our array can have 1B integer values too. It can exceed 1GB limit. So, TOAST may not be a very helpful option in this case.

In conclusion, because of Postgresql storage structure, saving Bi's column as an array would not be a good idea. Moreover, performing different query operations is also not convenient as compare to flat relational tables. For instance, it is hard to think of JOIN operation of two relational table on column B's. Hence, by considering the criteria such as easy to use, sql friendliness and storage, I can conclude that considering Bi as an array is not a good option and table looks more good when data filled in it looks uniform.

The next question comes to find a better alternative. One alternative is to think it as Many to Many relation ship between column A's and values of Bi. In this way I created another relational table with a single column and use it as a table inheritance. Here, parent table is containing the column A type and child table Ci for column Bi is containing all possible values present in column Bi by using this concept of table inheritance. It also provides useful flexibility: for example, child tables are permitted to have extra columns not present in the parent table and data can be divided in a manner of the user's choosing. I can also think of as many to many / one to many relationship with cross reference relational table between column A's and Bi for each i=1,2… perform join on these table to get the outputs.

Query Optimization: So far, I have created multiple flat tables instead of one table with array columns. This process helped in terms of storage but tables containing Bi columns may has millions or billions of rows. Doing sequential scan all over the table is very time consuming so the better approach would be to create a b-tree index on it. Creating index will increase the performance but it will take some storage space. The b-tree index could also completely fit into

main memory or It may require access some blocks from secondary storage. Moreover, a range query would take less time if I can reach to initial value of a range by index scan and further by sequential scan. Therefore, in some cases partial index could be a better option but can I explore any other better way?

Let us think of JOIN operation on R1 and R2: It is possible that tables containing Bi columns (1 row table) could not fit into one block. In such scenario to performing query operations I need to access many blocks, it would end up in increasing time. To resolve this issue, I can further create the partitions of tables containing Bi columns such that the rows which generally access together can come in one partition and 1 partition can completely fit into the one block.

Such partition will also help in unnecessary scanning during JOIN operation. Postgres perform partition operation by using the concept of table inheritance. There are many advantages and disadvantage of using partition mentioned below:

Benefits of Partition Inheritance:

1. Improved inserts and queries from being able to fit all indexes in memory.
2. Improved queries from being able to use constraint exclusion.
3. Having a smaller dataset (per partition) may enable the database to use sequential scans efficiently.
4. Maintenance operations like VACUUM and REINDEX run faster because of smaller indexes.
5. Deleting entire partitions is much faster, as it only requires a DROP TABLE and avoids expensive vacuuming operations.

Limitation of inheritance:

1. Inheritance does not automatically propagate data from INSERT or COPY commands to other tables in

the inheritance hierarchy.

Commands that are work automatically on Inheretance herarichy are : Select, Update and Delete

Commands that are not work automatically on Inheretance herarichy are: Insert, Copy

2. A serious limitation of the inheritance feature is that indexes (including unique constraints) and foreign key constraints only apply to single tables, not to their inheritance children.

Further, I can also perform VACUUM on each table to reduce the storage wastage.

To overcome with the limitation related to insert command, I can create Trigger, which will be active when any insertion operations are going on.

Now, creating index on each sub partition separately, can give us the advantage of saving entire index per partition in the main memory, thus this will further reduce the secondary storage access time.

Finally, the best way of performing Join operation between two relational table having b-tree index on each partition created by inheritance is to use Nested loop blocked based join with b-tree index on tables having Bi columns. Since tuple based nested loop join would take T(R1) times T(R2) I/O access of secondary storage while Block based would take B(R1) + B(R2).B(R1)/M-1. Here, B(R1)<B(R2) and M is the total memory size in terms of blocks. Therefore, I would consider the outer table that would be having less values in comparison to inner table.

Example:

```
Create table R1(
A1 int,
A2 int,
);


Create table R2(
A1 int,
A2 int,
);

Create table R1b1(
B1 int);
```

Similarly, I will create single column table for each Bi of both relation.
I can also create the cross-reference tables/junction tables in between the table containing A columns and table having B column, for each B.

Suppose B1, B2, B3, B4 all has 1million values (Note: all can have different number of values as well). I will create 10 partitions of each containing 10% of the total value in each. Also note that the value in each partition usually accessed together.

```
Create table R1b1(
Check (B1 > a and B1< b)
)inherits(R1b1);
```

Suppose (a,b) is containing 10 percentage value of whole B1. Also assume that it can fit in one block and it usually calls together.

Note: This is a one way of creating partitions. One can create it as per the requirement.

In the similar way I can create all 10 partitions for B1 and also for B2,B3 and B4. Infect, for any number of tables having B columns.

```
Create table R1b1(
Check( A1< 1000)
)inherits(R1);
```

Moreover, we could also create the partitions for columns name A's as per the requirement of table by the targeting aim of reducing secondary storage access.

Note: The process of insertion requires trigger to be fired at the time of insertion. And whole process also requires reindexing and Vaccum to perform the query effectively and reduce the wastage of storage space.

After creating the partitions, I can create the B-tree index as per the requirement. I would create the index only on those columns which are called in the query for instance, I would create an index of column Bj if the aim is to perform a JOIN operation on Bj column. Unnecessary index can also increase the total run time.

Conclusion: If a relational table R contains the values like array than we should split this relational table to multiple table and we can get the original table by performing JOIN in between parent table, child table and cross-reference table. We should also use table partition with inheritance as well as create index on tables having one row of B's columns. During the partition, we should also keep in mind that one partition can fit into one block. Moreover, Join operation should be done by Block nested loop with index method and outer table should be having less values in comparison to inner table on JOIN column.