Performance Enhancements

Extraction modules that operate solely on the minimized database D^1– e.g. algebraic predicates – have negligible cost due to its miniscule size. However, modules that need to work with the original database D_I – e.g. NEP extraction – could incur significant overheads if D_I is large. Therefore, XPOSE employs the following techniques to reduce performance bottlenecks in these modules.

6.6 Correlated Sampling [17]

Correlated sampling is a technique that makes use of the schema join graph in the sampling process. This results in a higher probability of the sampled data satisfying the join predicates. It is used before the database minimization step to obtain a smaller D_I that produces a FIT-result, thereby reducing the iterations in the minimization. This is also useful in outer join queries, where random sampling is susceptible to producing tuples with mismatched keys.

6.7 View-based Database Minimization

We employ a minimization technique based on *virtual views*, which does not require copying the records of a table during the binary halving process. The views are created on the base table by utilizing system-generated tuple identifiers, which give the physical location of a row in the table – for instance, in PostgreSQL, this identifier is called `ctid` and consists of a block number and a record number within that block. The `ctid` of the first record of a table is (0, 1). The number of records present in a block, n_b, is table-width dependent but computable from the schema. Based on n_b, we can estimate the `ctid` of the middle row of the table. For example, the following queries create a view containing roughly the upper half of table T:

```
Alter Table T Rename to T_dummy;
Create View T as
    Select * From T_dummy
    Where ctid between '(0,1)' and '(|T_dummy|/2n_b,1)';
```

If a FIT-result is obtained, the view creation continues recursively with the upper half; if not, it shifts to a virtual view on the lower half. This reduction continues until a D^1 is achieved. The key improvement over the explicit halving approach is the avoidance of the time and space costs for materializing the intermediate tables.

6.8 Hash-Based Result Comparators

This component aims to efficiently verify the equality of the results of $Q_{\mathcal{H}}$ and $Q_{\mathcal{E}}$ over D_I . The direct but slow technique is to explicitly compute, using the EXCEPT command, the difference between the results in both directions – i.e. $\mathcal{R}_{\mathcal{H}}$ EXCEPT $\mathcal{R}_{\mathcal{E}}$ and $\mathcal{R}_{\mathcal{E}}$ EXCEPT $\mathcal{R}_{\mathcal{H}}$, and verify that both are zero. This especially becomes a performance bottleneck in NEP extraction. Therefore, XPOSE employs the following hash-based result comparators instead. While PostgreSQL has several options for hash functions, we use the `HashText` hash function since it works across datatypes.

6.8.1 Global Hash Method: This method is used if the query has `ORDER BY` ($\vec{O}$) or `GROUP BY` attributes. It sorts $\mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{E}}$ on remaining projection attributes wrt $\vec{O}$, then computes hashes on each result table. If they are equal, the result tables are the same.

6.8.2 Rolling Hash Method: This method is used if the query has no physical ordering. Here, we calculate a hash value for a relation by applying a hash function to each tuple in the relation and then aggregating the results. In the PostgreSQL database, to get the rolling hash of the tuples present in the result set $\mathcal{R}_{\mathcal{H}}$, we use:

```
Select SUM(rh_hashes.hashtext)
From (Select hashtext(R_H::TEXT) From R_H) as rh_hashes;
```

The same evaluation is done for $\mathcal{R}_{\mathcal{E}}$. Comparing the two hashes confirms or denies the unordered set equality of $\mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{E}}$.