Journal Homepage: - www.journalijar.com

INTERNATIONAL JOURNAL OF ADVANCED RESEARCH (IJAR)

Article DOI: 10.21474/IJAR01/11726
DOI URL: http://dx.doi.org/10.21474/IJAR01/11726

ISSN NO. 2320-5407

RESEARCH ARTICLE

CORRELATIONS AND QUERY PROCESSING

Bhanu Shanker Prasad
P.G Deptt of Statistics and Computer Application,TMBU,Bhagalpur,Bihar,India Job Status:- M.D.N H/S School,Dumrama, Amarpur, Banka.

Manuscript Info

Manuscript History
Received: 15 July 2020
Final Accepted: 18 August 2020
Published: September 2020

Key words:-
Optimization, Selectivities, Relation, Partition, Statistical , Correlation

Abstract

It is known that optimization of join queries based on average selectivities is sub-optimal in highly correlated databases. Relations are naturally divided into partitions , each partition having substantially different statistical characteristics in such databases. It is very compelling to discover such data partitions during query optimization and create multiple plans for a given query , one plan being optimal for a particular combination of data partitions. This scenario calls for the sharing of state among plans, so that common intermediate results are not recomputed. We study this problem in a setting with a routing-based query execution engine based on eddies. Eddies naturally encapsulate horizontal partitioning and maximal state sharing across multiple plan. The purpose of this paper is to present faster execution time over traditional optimization for high correlations, while maintaining the same performance for low correlations.

Introduction:-

Traditional query optimizers pick one execution plan per query, based on first-order statistics about the underlying data. In particular, a join order is determined based on join selectivities that are computed over a relation as a whole. However, real-world databases often contain skewed data with complex correlations, and first-order statistics are not sufficiently powerful to capture the underlying statistical properties of the data. Indeed, one can get better join selectivity estimates by modeling data correlations [6,13]. However, the presence of data correlations does not only make selectivity estimation harder-it also offers opportunities for more ef-fective query optimization.

When data correlations are present, the input relations are naturally divided into partitions, each partition having completely different statistical characteristics. It is then very attractive to create *multiple plans* per query, each plan being optimized for a different combination of data partitions. Consider for example the join query $R \bowtie S \bowtie T \bowtie U$. Assume that S is naturally partitioned into two partitions, $S = S_1 \cup S_2$, where S_i (similarly, S_2) has a low selectivity when it joins with R (T), and a high selectivity when it joins with T (R). A possible optimization process may decide to partition S into S_i and S_2, and pick the plans $(R \bowtie S_1) \bowtie (T \bowtie U)$ and $((T \bowtie U) \bowtie S_2) \bowtie R$. The combined cost of the two resulting plans can be smaller than the cost of any possible monolithic plan.

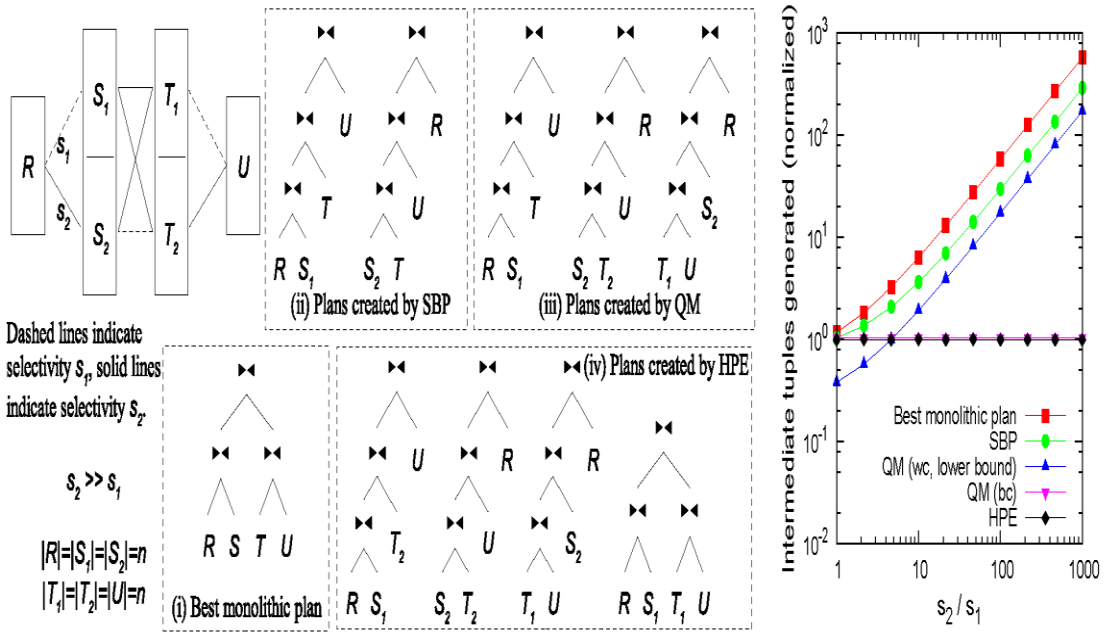
Corresponding Author:- Bhanu Shanker Prasad
Address:- Bulaki Sah Lane, Baltikarkhana Chowk, Mirjanhat, Bhagalpur , Bihar,812005.

With the introduction of partitioning, query optimization consists of two tasks: Determining the partitions of the input relations, and creating a plan for each combination of partitions. Unfortunately, the two problems are inter-dependent. A partitioning of the relations is optimal only with respect to already chosen join plans. A partitioning is query-plan specific, because it is evaluated against the selectivities of the joins in the join plans; it is not merely a set of clusters based on the statistical properties of the data. Conversely, a collection of join plans is optimal only with respect to a certain partitioning. This inter-dependence yields a much larger optimization space than the one considered by traditional query optimizers.

Further, an optimization process that results in multiple plans per query naturally raises the issue of sharing state among the constituent plans at execution time. Identical intermediate tuples should not be constructed multiple times from different plans during query execution. In the example above, the intermediate relation T ⋈ U is required in both plans. This relation should not be constructed twice; rather, it should be shared between the two plans.

This work presents the first study of horizontal partitioning during query processing with *maximal* sharing of intermediate results. In particular, the contributions of this paper are the following: First, we offer a more formal study of the general problem than hitherto. We introduce the notion of *conditional join plans* (CJPs), a representation of the search space resulting from horizontal partitioning that captures both the partitioning and join order aspects. We define recursive cost formulas for CJPs, and are thus able to define query optimization as a search problem in a suitable space. In addition, we show how to estimate correlated join selectivities using low-overhead summaries based on graphical models. Then, we focus on the case of query execution with eddies [1] and symmetric hash joins. This case is particularly interesting, because sharing is maximal; an intermediate tuple that is used by different join plans is computed only once. We show how query execution with eddies restricts the search space, and we provide a low-overhead greedy algorithm for this space. Our algorithm can achieve an order of magnitude better execution time than the best monolithic plans in databases with high correlations, while being on par with traditional query optimization for uniform data.

The rest of this paper is organized as follows. reviews related work and eddies.



(a) Running example query and the join plans chosen by a (i) a traditional query optimizer, (ii) selectivity-based partitioning [18], (iii) query mesh [17], and (iv) our approach.

(b) Intermediate tuples generated (normalized).

Figure 1:- A 3-join query R ⋈ S ⋈ T ⋈ U used as a running example throughout the paper.

Partitioning with eddies

Eddies with symmetric hash joins [1,8] provide a framework that naturally encapsulates horizontal partitioning and state sharing, making it an ideal framework for exploiting data correlations through horizontal partitioning. With eddies, fixed query plans are no longer constructed. Instead, the operators that are involved in the query are connected with a central router (the eddy), and query execution proceeds by routing the tuples through the operators. The eddy makes a routing decision for each individual tuple. This enables multiple plans to be executed simultaneously for the same query, each plan operating on a different subset of (base or intermediate) tuples. These multiple plans are not created explicitly; rather, they are implied by the eddy routing policy. Note that although eddies were introduced as a way to achieve adaptivity in a streaming environment, we do not use them as such. We assume a more traditional setting, where the data is static. This eliminates the adaptivity overhead of eddies.

Consider the join query R ⋈ S ⋈ T ⋈ U and the execution of the query using an eddy, as shown in Figure 2. Tuples from relations *R* and *U* each have only one possible destination: *R* ⋈ *S* and *T* ⋈ *U,* respectively. However, *S* tuples can be routed to either R ⋈ S or S ⋈ T, and T tuples can be routed to either T ⋈ U or S ⋈ T. The eddy can use a predicate on one of the relation attributes to distinguish the routing destinations. In Figure 2, the eddy uses the predicate ϕ_S (e.g., ϕs = (S.Y > 5)) to route S tuples. Tuples from S that satisfy ϕ_S are routed to R ⋈ S, yielding partition S_1. Tuples from S that do not satisfy ϕ_S are routed to S ⋈ T, yielding partition S_2.

In Figure 2 the intermediate results, as stored in the hash tables of the symmetric hash joins, are shown. While all R (U) tuples are stored in the join R ⋈ S (T ⋈ U), the relations S and T are partitioned. The $S_1(T_1)$ partition is stored in R ⋈ S (T ⋈ U), and the S_2 and T_2 are stored in S ⋈ T. Thus, the intermediate results created are RS_1, S_2 T_2, and T_1U. The RS_1 and T_1U tuples are stored in S ⋈ T (their only routing destination). The state of S ⋈ T is then as shown in Figure 2. The subsequent routing of intermediate results in the combined execution of the four plans shown in Figure 1(a). The state captured in the joins at the end of query execution is shown in Figure 2. Note that the relations RS_1 and T_1U that are common in multiple plans are computed only once. Eddies provide maximal sharing of intermediate results at execution time, with no extra optimization time overhead.

Eddy restrictions

The routing nature of query execution with eddies imposes constraints on the possible partitions as well as on the join plans that can be executed. This in turn imposes restrictions on the CJPs that can be considered during query optimization. Consider for example the valid CJP for our example query in Figure 6.

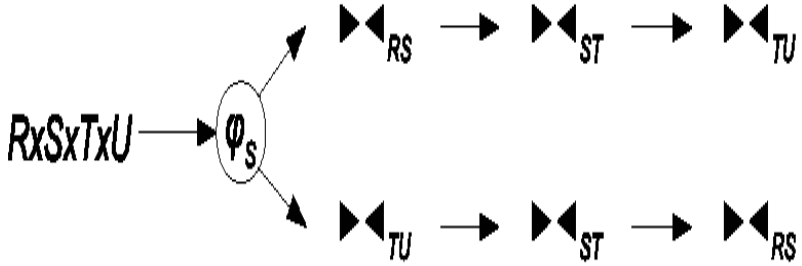


Figure 6:- A CJP that is not eddy-compliant.

This CJP is equivalent to the join plans ((R ⋈ S1) ⋈ T) ⋈ U, ((T ⋈ U) ⋈ S_2) ⋈ R. If we were to execute this query with an eddy, we need to make a routing decision for T tuples using a predicate, 0_S, on relation S. If ϕ_S = T, T needs to be joined with R N S_1, while if ϕ_S = F, T needs to be joined with U. There is no possible routing that can achieve this. The routing decisions for T tuples can only be made using a predicate on T, 0_T. The restrictions on the possible CJPs is the price paid for state sharing provided by eddies.

The constraints imposed by eddies affect the CJP search space as follows. Given a query Q, we can construct a *unique* CJP structure $P_e(Q, F_e)$, called the *eddy CJP structure.* Any CJP valid for Q that can be executed using an eddy (called an *eddy-compliant* CJP) can be derived from the eddy CJP structure by assigning values to the predicates in F_e. Hence, the eddy CJP structure determines the *eddy CJP space* for this query. Appendix A

details an algorithm that, given a query, constructs the unique eddy CJP structure. Figure 7 shows the eddy CJP structure for our example.

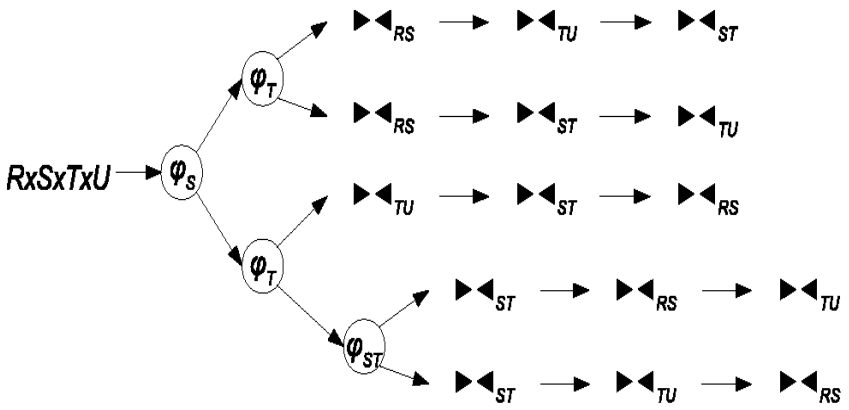


Figure 7:- The eddy CJP structure for our running example. Note that the predicate 0_T must have a unique value.

For example, assume that we are given a partitioning budget of c =1 and we decide to use the predicate 0_S = (S.Y > 5). Then, all the eddy-compliant CJPs can be derived from the eddy CJP structure of Figure 7 by assigning values to 0_T and 0_{ST} from the set $\{0_{true}, 0f_{alse}\}$, as defined in Section 3.1. These values must be honored across sub-plans. For example, assume that we choose $0_T = 0_{true}$. Then we must use the join order (R ⋈ S_1) ⋈ (T ⋈ U) for partition S_1 and the join order ((T ⋈ U) ⋈ S2) ⋈ R for partition S_2. Note that 0_{ST} is not defined in the $0_T = 0<^{TM}_e$ sub-plans because the intermediate result ST is never formed in these sub-plans.

Since eddies provide maximal sharing, the recursive cost function $COST_{NS}$ from Section 3.4 does not estimate the cost of an eddy- compliant CJP correctly. Fortunately, we can define a recursive cost function $COST_{Eddy}$ that estimates the cost of an eddy-compliant CJP with sharing accounted for. Only one change is needed to $COST_{NS}$: Instead of including the decision predicates of the set , where *X* is the set of relations relevant to the join node under consideration, in Equations 3- 5, we simply include all the decision predicates in $. The following holds.

Lemma 2. *If* P *is eddy-compliant,* $COST_{Eddy}$ (P) = ||P||s
PROOF. See Appendix B.

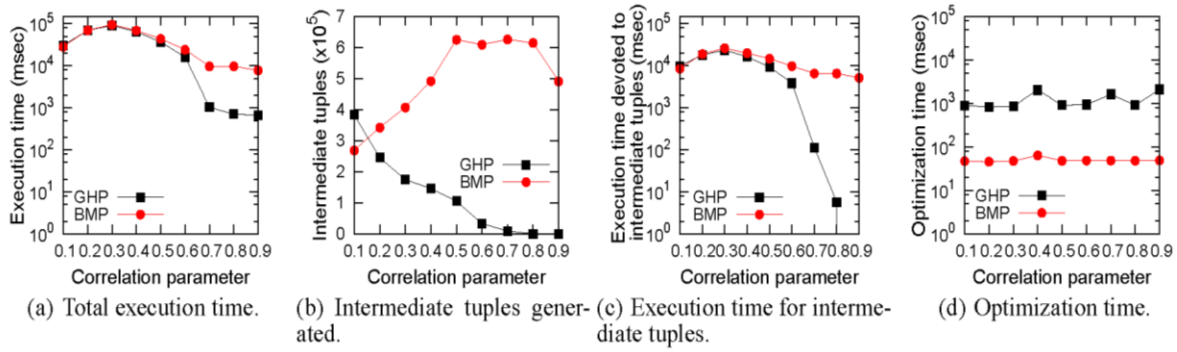
Denote the eddy CJP structure for a query Q by $P_e(Q, F_e)$. We can now formally state the problem we are solving.

Horizontal paritioning with eddies. Given a query Q and a partitioning budget c, find the plan

$$P^*(\mathcal{Q}) = \underset{|\mathcal{F}(P)|\leq c, \mathcal{F}(P)\subset\mathcal{F}_e}{\arg\min} [\text{COST}_{Eddy}(P(\mathcal{Q}))]$$

that is valid for Q and is eddy-compliant.

Put differently, query optimization has to partition the predicate variables F_e of the eddy CJP structure into two disjoint sets: The first set of size at most c contains predicates that are assigned normal predicate values (e.g., S.Y > 5), and the second set, of size at least $|F_e|$ — c contains predicates that are assigned values from the set $\{0_{true}, 0f_{alse}\}$. The choice of the two sets and the choice of values should yield the minimum cost. Once the predicates in F_e have been assigned values, it is trivial to construct an eddy routing policy that executes the resulting concrete CJP.



(a) Total execution time. (b) Intermediate tuples generated. (c) Execution time for intermediate tuples. (d) Optimization time.

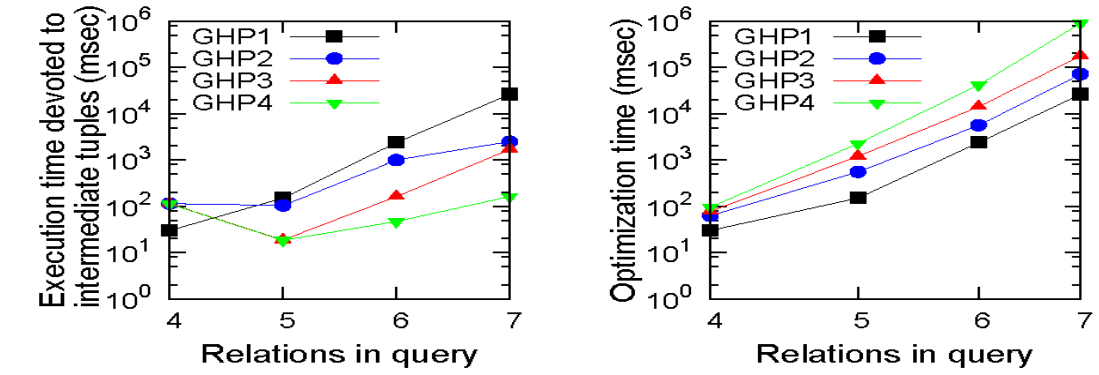
Figure 5:- The effect of varying the correlation parameter r in a 3-join query when all the joins have the same selectivity.

Greedy search

While possible, it is computationally infeasible to exhaustively search the eddy space. We propose an algorithm that starts from the best monolithic plan for a query and gradually builds an eddy- compliant CJP. At each step, the algorithm cycles over all the decision predicates on attributes that have not been used yet, and picks the one that yields the best cost when used to split the plan into two sub-plans. This is done greedily: when the algorithm introduces a split, it assumes that no future splits will occur, but rather that the best monolithic plans (under the eddy constraints) will be used for the sub-plans. The algorithm stops if it has introduced the maximum number c of decision predicates allowed, or if no further cost improvement can be achieved.The gradual construction of the CJP has three advantages. First, the complete eddy CJP structure does not need to be stored. Second, the sizes of the CJPs whose costs will be evaluated are controllable; a CJP with more than c decision predicates is never generated. Finally, the cost of the final CJP is guaranteed to be less or equal to the cost of the best monolithic plan. However, since there is no backtracking, the algorithm can obviously get stuck in local minima; an initial choice for a locally optimal decision predicate can lead the algorithm to assume that no cost improvement can be made by further splitting. Appendix C provides the details and pseudo code for the greedy search algorithm, as well a discussion of its cost as compared to the cost of exhaustive search.

Conclusions and Future Work

Data correlations provide opportunities for more effective query optimization by partitioning relations. We first present a principled way to approach the problem of horizontal partitioning as search in the space of conditional join plans. CJPs provide an intuitive way to think about the problem, and recursive cost formulas for CJPs can be defined. Further, we show how to efficiently estimate correlated selectivities using a statistical model with low storage overhead. Then, we show how the sharing of intermediate results that is inherent in eddies restricts the space of possible CJPs. A



(a) Execution time for intermediate tuples. (b) Optimization time.

Figure 9:- Varying the number of relations and GHP iterations.

greedy search with controlled iterations in this space is proposed that can achieve an one order of magnitude better execution time for highly correlated databases, while performing on par with the best monolithic plan at low correlations.

This work opens several lines of research that we plan to pursue. First, a problem that remains open is whether shared computation is always beneficial. Second, it would be interesting to explore multi- query optimization in this environment, where multiple queries are optimized together to produce many join plans that share computation. Finally, we would like to explore the parallel query processing case where the optimization metric is throughput.

References:-

1. Avnurand J. M. Hellerstein. Eddies: Continuously adaptive query processing. In SIGMOD, pp. 261-272, 2000.
2. Bizarro, S. Babu, D. J. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In VLDB, 2005.
3. Chandrasekaran, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In CIDR, 2003.
4. Cluet, and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products In ICDT, 1995.
5. Deshpande. An initial study of overheads of eddies. SIGMOD Record, 33(1):44-49, 2004.
6. Deshpande, M. N. Garofalakis, and R. Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. In SIGMOD, pp. 199-210, 2001.
7. Deshpande, C. Guestrin, W. Hong, and S. Madden. Exploiting correlated attributes in acquisitional query processing. In ICDE, pp. 143-154, 2005.
8. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In VLDB, pp. 948-959, 2004.
9. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. Foundations and Trends in Databases, 1(1): 1-140, 2007.
10. D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. CACM, 35(6):85-98, 1992.
11. P. L. Fackler. Generating correlated multidimensional variates.
12. L. Getoor. Learning Statistical Models from Relational Data. PhD thesis, Stanford University, 2001.
13. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In SIGMOD, pp. 461-472, 2001. D. Koller, and N. Friedman. Probabilistic graphical models. MIT Press, 2009.
14. R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In VLDB, pp. 128-137, 1986.
15. R. V. Nehme, E. A. Rundensteiner, and E. Bertino. Self-tuning query mesh for adaptive multi-route query processing. In EDBT, 2009.
16. R. V. Nehme, K. Works, E. A. Rundensteiner, and E. Bertino. Query mesh: Multi-route query processing technology. PVLDB, 2(2), 2009.
17. N. Polyzotis. Selectivity-based partitioning: a divide-and-union paradigm for effective query optimization. In CIKM, 2005.
18. V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In ICDE, pp. 353-, 2003.