Adaptively Reordering Joins during Query Execution

Quanzhong Li* Minglong Shao+ Volker Markl* Kevin Beyer* Latha Colby* Guy Lohman*

*IBM Almaden Research Center
{quanzhli, marklv, kbeyer, lathac}@us.ibm.com
lohman@almaden.ibm.com

+Computer Science Department
Carnegie Mellon University
shaoml@cs.cmu.edu

Abstract

Traditional query processing techniques based on static query optimization are ineffective in applications where statistics about the data are unavailable at the start of query execution or where the data characteristics are skewed and change dynamically. Several adaptive query processing techniques have been proposed in recent years to overcome the limitations of static query optimizers through either explicit re-optimization of plans during execution or by using a row-routing based approach. In this paper, we present a novel method for processing pipelined join plans that dynamically arranges the join order of both inner and outer-most tables at run-time. We extend the Eddies concept of "moments of symmetry" to reorder indexed nested-loop joins, the join method used by all commercial DBMSs for building pipelined query plans for applications for which low latencies are crucial. Unlike row-routing techniques, our approach achieves adaptability by changing the pipeline itself, which avoids the bookkeeping and routing decision associated with each row. Operator selectivities monitored during query execution are used to change the execution plan at strategic points, and the change of execution plans utilizes a novel and efficient technique for avoiding duplicates in the query results. Our prototype implementation in a commercial DBMS shows a query execution speedup of up to 8 times.

1. Introduction

Most database management systems use an optimizer to generate an optimal query execution plan during the query compilation phase. Once an execution plan is generated, the plan remains unchanged during its execution. Since optimizers use cost-based models to optimize a query, the quality of a query execution plan depends heavily on the cost model and related statistics. However, the statistics may be unreliable or insufficient, since the statistics are approximations of the original data, and the statistics may not be accurate or up-to-date. Furthermore, there are many situations where no statistics are available and some defaults or assumptions have to be used, which deteriorates the accuracy of the cost model. For example, an optimizer may assume uniformity for value distributions or assume that column values are independent, which can result in incorrect selectivity estimates when the values are, in fact, highly correlated. When the assumptions used by an optimizer are not justified by the underlying data, the resulting errors in estimating the characteristics of the base data can propagate exponentially while estimating the cost of a plan [10, 18]. Thus, although an execution plan generated by an optimizer may be the optimal plan based on the current statistics and assumptions, it may be far from optimal for the actual input data set.

We use the following example to illustrate some of the problems with static optimization.

Example 1:

```
SELECT o.name, a.driver
FROM Owner o, Car c, Demographics d, Accidents a
WHERE  c.ownerid = o.id AND o.id = d.ownerid
  AND  c.id = a.carid
  AND  (c.make='Chevrolet' OR c.make='Mercedes')
  AND  o.country1 = 'Germany'
  AND  d.salary < 50000;
```

This query tries to examine accidents in Germany for certain makes of cars. Since the query involves only two makes, a likely query execution plan would have the `Car` table as the outer-most table with an index scan on `make`. Because there are relatively few `Chevrolet` cars sold in `Germany`, the `Owner` table (with the predicate `country1 = 'Germany'`) has the better filtering power when processing `Chevrolet` cars. Therefore, it would be more efficient to place this table before the `Demographics` table in the query execution pipeline. However, when scanning `Mercedes` cars, it would be better to have the `Demographics` table before the `Owner` table, since the former would have a better filtering effect than the predicate on `Owner` in the case of luxury cars. In this query, any fixed order of the `Demographics` and `Owner` tables would be sub-optimal for the entire data set. The optimal order can only be achieved by monitoring the execution and dynamically switching the orders according to the data being processed. This example illustrates the need for allowing the run-time system of a DBMS to adaptively process queries according to actual data characteristics.

In this paper, we propose a novel method of adaptively reordering joins during query execution. We extend the Eddies concept of "moments of symmetry" [3] to indexed nested-loop joins, and we focus on optimizing the evaluation of pipelined plans. Our proposed method

1-4244-0803-2/07/$20.00 ©2007 IEEE.

exploits information such as join selectivities that can be monitored during query execution to rearrange join operators at run-time. Because of the complexities of contemporary database applications, comprehensive statistics on join selectivities and correlations of different column values are costly to maintain and not very reliable. On the other hand, simpler statistics such as table cardinalities and single column predicate selectivities tend to be more reliable and easier to maintain. Our method exploits simple and reliable statistics, as well as statistics collected by run-time monitors to estimate the benefit of changing join orders for the remainder of the current query. We also present novel and efficient techniques for avoiding duplicates that could otherwise result from operator switching.

The rest of the paper is organized as follows. In Section 2, we give a brief survey of related work and contrast it with the contributions of this paper. Section 3 provides an overview of pipelined query execution and existing techniques for costing them. Detailed descriptions of our adaptive reordering techniques are provided in Section 4. We present the results of an extensive evaluation in Section 5 and summarize our contributions and ideas for future work in Section 6.

2. Related Work and Our Contributions

Several adaptive query processing techniques have been proposed in the literature. These techniques can be classified broadly into two categories: a) optimizer plan-based techniques and b) row-routing techniques.

Many techniques have been proposed for generating multiple execution plans for a single query [2, 17, 19]. These techniques either choose the best complete plan at the beginning of execution, or choose from embedded alternative sub-plans in the middle of execution at materialization points. To deal with the uncertainties of run-time resources and parameter markers in queries, [11, 14] propose generating multiple plans at compile time; when the unknowns become known at run time, a decision tree mechanism decides which plan to execute. Adaptive data partitioning (ADP) [19] divides the source data into regions to be executed by different but complementary plans, with a final phase that "stitches-up" the results from each partition to derive the complete result. In the technique proposed in this paper, by contrast, only one execution plan is generated with a small number of switchable single-table access plans embedded in that plan. Adaptation is achieved by modifying the join order of the current plan in mid-execution. This avoids the overhead of compiling, storing, retrieving, and deciding among a large number of alternative join plans.

Another plan-based approach calls the optimizer to generate a new plan when inconsistencies are detected [20, 22]. Progressive query OPtimization (POP) [22] detects cardinality estimation errors in mid-execution by comparing the optimizer's estimated cardinalities against the actual run-time count of rows at natural materialization points (e.g., sorts) in the plan. If the actual count is outside a pre-determined validity range for that part of the plan, a re-optimization of the current plan is triggered. The DB2 LEarning Optimizer (LEO) [25], in contrast, waits until a plan has finished executing to compare actual row counts to the optimizer's estimates. It learns from misestimates by adjusting the statistics to improve the quality of optimizations in future queries. However, LEO does not address how to improve the performance of the current query in mid-execution. The technique proposed in this paper focuses on dynamically reordering a pipelined join plan in mid-execution, and does so without having to re-invoke the optimizer to generate new plans.

Row-routing techniques, typified by the Eddies framework [3, 12, 13, 24], do not require a complex compile-time optimizer. Instead, a run-time optimizer routes each row independently through a sequence of join operators. Since each row may be routed differently, information about which joins have been completed must be maintained with each row. This avoids the complexities and overhead associated with a heavy-duty compile-time optimizer, and provides extreme flexibility and fine granularity for adaptation. Unlike row-routing techniques, the technique proposed in this paper does not require major changes in the run-time of main stream database systems. We leverage the capabilities of an existing optimizer and exploit any available statistics to obtain an initial execution plan. The adaptation is achieved at run-time by changing the pipelined plan itself only at strategic points that provide us enough opportunities to adapt to changing data characteristics while minimizing the bookkeeping and, at the same time, avoiding row-level routing overhead. The Eddy state module (SteM) enhances Eddy by allowing the choice of access methods and join algorithms. Unlike SteM, our work is based on the indexed nested-loop join, which has very small memory footprints and does not hash or cache rows in memory.

Techniques to process stream joins are another form of row routing. Query scrambling [1, 27] adapts to data delays from remote data sources by attempting to let other parts of the query progress. XJoin [26] extends the symmetric hash join by allowing parts of the hash tables to be moved to secondary storage. When input rows are unavailable, XJoin uses the rows from secondary storage to continue probing hash tables in memory. Adaptive caching [7] adjusts cache placement and cache maintenance dynamically in a streaming environment. Adaptive ordering [6] of stream filters focuses on adaptively processing data streams. Dynamic plan migration [28] changes join plans dynamically for stream

join plans. Most of these stream processing techniques are based on the symmetric hash join algorithm. Bulk join methods like symmetric hash joins require a huge amount of memory, thus being prohibitive for large scale OLTP systems, where pipelined plans are mandatory and the only low-resource pipelined join method is indexed nested-loop join (or index merge). Supporting indexed nested-loop join is crucial in order to bring adaptation techniques to commercial database systems.

Both plan-based and row-routing schemes require special techniques to deal with the incorrect introduction of duplicates due to run-time changes in the execution strategy. Plan-based re-optimization methods either delay producing results or keep track of the results that have been output. One technique suggested in a variant of POP saves all row IDs (RIDs) returned to the user by a pipelined plan. The rows produced by any newly re-optimized plan must then be anti-joined with those saved RIDs to eliminate duplicate results [22]. To prevent duplicate results caused by overlapping processing stages in XJoin [26], Urhan et al. proposed a row-marking algorithm based on timestamps. SteMs [24] also used a similar timestamp technique to prevent duplicates. This paper avoids the generation of duplicates by reordering inner tables only at safe points in the processing. Duplicates that might be introduced when the outer-most table is reordered are avoided by simply adding a predicate on the ordering column to prevent generating duplicates.

In summary, the main contributions of this paper are:
1) We propose a technique that can dynamically rearrange the join order of pipelined join execution plans, which:
- includes the reordering of the important class of indexed nested-loop joins. Both inner and outer-most tables can be reordered in response to changing data characteristics measured during execution.
- does not require row-level bookkeeping of joins; it needs only one cursor per table to remember the position of the last processed row of that table.
- minimizes the number of plans that must be pre-compiled by using one initial execution plan with a small number of switchable single-table access plans.

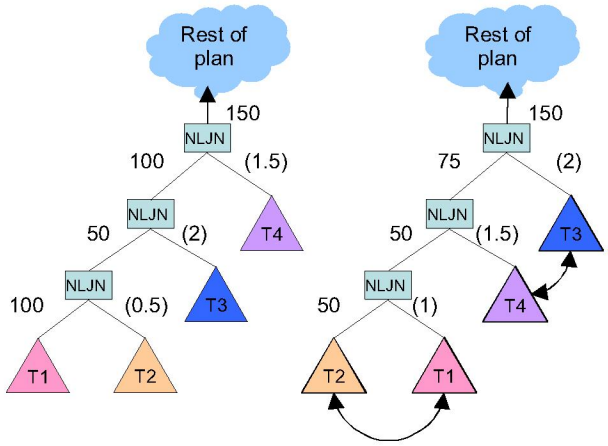
2) We provide a detailed experimental evaluation of this proposed technique using a prototype implementation in a commercial database system. The results show that the adaptive join reordering technique can provide significant improvement over static optimization.

3. Background

In this section, we review pipelined query execution, and describe the high-level ideas of adaptive join reordering, including the important assumptions, the terminology, and the goals to be achieved.

3.1. Pipelined Query Execution

Pipelined plans are frequently used by a lot of applications, including OLTP, CRM, and many Web applications (e.g., to obtain the first screen-full results quickly), for which low latencies are crucial. Nested-loop join, especially indexed nested-loop join is the preferred and predominant join method used by all commercial DBMSs for building pipelined query plans. Another important advantage of pipelined plans using indexed nested-loop joins is their small memory consumption. Since a pipelined query plan is a non-blocking plan, there are no materialization points. The techniques that we present are complementary to, and can readily work with, other techniques such as POP, LEO and XJoin.



(a) Original Plan (b) Reordered Plan

Figure 1: An example of a nested-loop join plan

In this paper, we assume that proper indexes are built on join columns. A pipeline can be either a whole query plan or a pipelined portion of a bigger and more complex plan, as shown in Figure 1(a). We shall call the outer-most table of a query plan the *driving table* or the *driving leg*. The rest of the tables are referred to as *inner tables* or *inner legs*. We shall use Figure 1(a) as an example to briefly describe pipelined processing in a database. A database execution engine (run-time) starts by retrieving an outer row from the driving table, T_1. Then the values of the join attributes of this row are fed to the access operator of T_2, which uses its join predicates to fetch rows that satisfy the predicates local to T_2 and the join predicate with the values from T_1 treated as constants. These qualified rows are then used as the inputs of the second join operator. The processing of the second join operator is similar to the first one, except that now the left join table is not a base table, but is the set of matching rows from a previous join.

Using pipelined processing, the inner table of each join operator will be probed for each incoming row from the corresponding outer table. Each inner table will request another outer row only after it has completely finished the

processing of the current outer row, i.e., when all matching rows have been already sent on to succeeding join operators. This is an important property that we exploit during join reordering to minimize the bookkeeping overhead. This property also distinguishes the reordering algorithm for inner tables from that for driving tables.

3.2. Cost of Pipelined Execution Plans

We define the cost of pipelined processing to be the sum of the processing costs of individual joins in the pipeline. The cost of a join operator is calculated by multiplying the cardinality of its outer table with the cost of the join operation incurred for each row from the outer table. In the following, we shall formalize this cost calculation.

For ease of exposition, we define table positions in a pipelined query plan by starting from the driving table, which has position 1, and thereafter numbering each table in increasing order following the pipelined plan. Since the positions of tables in a pipelined plan may be changed during reordering, we define a mapping function called "*o*". This mapping function maps table positions in a pipeline to table IDs: "*o(position)* $\rightarrow$ *ID*". A join order of tables in a pipeline can then be represented as a table sequence, $T_{o(1)}$, $T_{o(2)}$, … , $T_{o(k)}$, assuming that there are *k* tables in the pipeline with IDs from 1 to *k*. For example, the sequence in Figure 1(b) is T_2,T_1,T_4,T_3.[1]

For a table T, we define *C(T)* as the cardinality of T. Since there may be local predicates on table T, we use $C_{LEG}(T)$ to represent the cardinality of this leg after applying local predicates. We also use *JC(T)* to denote the join cardinality of T, which is the number of matching rows per incoming outer row after applying local predicates on the inner table. For ease of description, we define the join cardinality of the first table, $JC(T_{o(1)})$, to be the leg cardinality of that first table, $C_{LEG}(T_{o(1)})$, and we define $JC(T_{o(0)})$ to be 1. For each incoming row there is a cost to probe the table, and we use $PC(T_{o(i)})$ to represent the cost of probing $T_{o(i)}$. With the above notations, the cost of a pipelined plan for a query *Q* is defined in the following equation:

$$Cost(Plan(Q)) = \sum_{i=1}^{k}\left[PC(T_{o(i)})\prod_{j=0}^{i-1}JC(T_{o(j)})\right] \qquad (1)$$

where $JC(T_{o(0)}) = 1$ and $JC(T_{o(1)}) = C_{LEG}(T_{o(1)})$. A similar cost definition can also be found in previous work ([21]). The goal of adaptive join reordering is to dynamically estimate the parameters in Equation (1), and minimize the cost by reordering the joins. As an example, the numbers in parentheses in Figure 1(a) and Figure 1(b) are the join cardinalities of inner legs. Other numbers represent the number of rows flowing through each join operator, i.e., the outer leg cardinalities. Suppose that the probing costs are the same for all tables, and equal to *p*. For plan (a), the total cost is 251*p*, while for plan (b) the total cost is 176*p*.

3.3. Inner Table Rank and Order

For Equation (1), if the order of two adjacent tables $T_{o(i)}$ and $T_{o(i+1)}$ is switched, the new order is better if and only if the following is true:

$$\frac{JC(T_{o(i+1)})-1}{PC(T_{o(i+1)})} > \frac{JC(T_{o(i)})-1}{PC(T_{o(i)})} \qquad (2)$$

This property is called adjacent sequence interchange property (ASI), as observed in [15][21][23]. Based on this property, we can define the *rank* of each inner table as

$$rank(T_{o(i)}) = \frac{JC(T_{o(i)})-1}{PC(T_{o(i)})} \qquad (3)$$

Similar definitions of rank can also be found in static optimization schemes, e.g., the optimization of expensive methods [16] and user-defined predicates [9]. If we want to get the optimal join plan for a specified driving table, the inner tables should be ordered by their ranks in ascending order

$$rank(T_{o(2)}) < rank(T_{o(3)}) < \ldots < rank(T_{o(k)}) \qquad (4)$$

In other words, once we pick a driving table, the optimal join plan starting with this driving table can be determined by comparing the ranks of inner tables[2]. The intuition behind the rule is that we should process the most selective joins as early as possible so that unwanted data can be discarded as early as possible.

4. Adaptive Join Reordering

This section describes the algorithms of the adaptive join reordering in detail, including the monitoring of pipelined execution at run-time.

4.1. Inner Table Reordering

According to pipelined query processing, for an incoming outer row, all matching rows of an inner table will be found and sent to the next inner table. When all matching rows have been processed, the next outer row is requested. At this moment, we say this table is in a *depleted state*. We extend the concept of "*moment of symmetry*" [3] to indexed nested-loop joins, and use the notion of depleted state to describe the status of pipelines.

If table $T_{o(i)}$ is in a depleted state, then all tables from $T_{o(i+1)}$ to $T_{o(k)}$ will have already finished processing their

[1] This is exactly how System R [8] represented its plans (with some annotations) as an ordered sequence of table IDs.

[2] Note that for cyclic query graphs, the availability of join predicates may change when a table is positioned in different places of a pipeline. The join cardinality and rank may also change accordingly (this will be explained further in Section 4.3.4). So a total ordering via Equation (4) may not be possible. In this case, the rank can be calculated on composite tables. The reader may refer to [21] for further details about the composite ranks.

incoming rows, and these tables are also in depleted states. In other words, the whole segment of the pipeline starting from $T_{o(i)}$ is depleted. Since there are no outstanding rows or states remaining in this segment of the pipeline, changing the order of these tables within this segment will not affect the correctness of join results. Unlike row-routing techniques, we apply the change of join orders on the pipeline itself, only when the pipeline is in a depleted state. This technique minimizes the bookkeeping and avoids making routing decisions at the row level. At the same time, it provides sufficient opportunities for adapting the join order, as demonstrated in our experimental study.

REORDER_INNER_TABLE(i) :
1: **if** $T_{o(i)}$ finishes a batch of *c* incoming rows from $T_{o(i-1)}$ **then**
2: Calculate rank($T_{o(i)}$);
3: Check the orders of $T_{o(i)}$, $T_{o(i+1)}$, … $T_{o(k)}$;
4: **if** the ranks are not in increasing order **then**
5: Reorder ($T_{o(i)}$, $T_{o(i+1)}$, … $T_{o(k)}$) according to the current ranks;
6: **end if**
7: **end if**

Figure 2: Reordering Inner Tables

The algorithm to reorder inner tables is shown in Figure 2. This algorithm guarantees that an inner table reordering happens only when the table is in the depleted state. No bookkeeping on individual rows is needed to track the execution status of inner tables. The parameter of "*c*" at line 1 is used to control the reordering check frequency. It is a tunable parameter used to balance the optimality and the run-time overhead.

4.2. Changing the Driving Table

The driving table is the most important table in a pipeline. It determines the running time of the whole pipeline. Using different driving tables may result in significantly differences in performance. Thus, it is crucial to make sure that the correct driving table is being used. Changing the driving table in a pipeline is more complicated than reordering inner tables. Unlike inner tables, there are no depleted states that can be used to reorder the driving table. If a driving table is changed to an inner table, the already scanned rows in this old driving table may join with the rows from the new driving table again. Consequently, duplicate results may be generated. A query processing system should keep track of the execution status of the driving table in order to avoid generating duplicate rows caused by reordering. The challenge here is minimizing the tracking overhead. A straightforward way is to remember the generated results and add an anti-join at the end of the pipeline to check for duplicates. However, it is costly to keep all the results and perform anti-joins. One much better solution is to prevent generating duplicates in the first place.

We propose a low-overhead tracking method that exploits the order in which the driving table is accessed. Our technique adds a local predicate on the driving table to exclude the already processed rows if the driving table is changed. Suppose that the table access method on the driving table is an index scan. The access of the rows in the driving table will follow two orders. The first order is the sorted order of scanned index key values, which are already ordered when the index is built. If there is only one value to scan (e.g., for equality predicates), we can ignore this order. The second order is the record id (RID) order, which is also maintained in increasing order by the index for efficient data access. If we keep the scan positions of these orders, we can simply use a local predicate to exclude the accessed rows. For instance, suppose that the driving table is accessed through an index on "*age*" and the original query contains a local predicate on this index, "*age > 30*". Also, assume that we are in the middle of processing the rows at "*age = 35*" when we decide to change the driving table. In this case, we can either postpone the change until the scan on the rows of "*age = 35*" finishes, so that a simple predicate "*age > 35*" is sufficient to filter out the rows already processed, or we can make the change immediately and add a more complex predicate "*age > 35 OR (age = 35 AND RID > cur_RID)*", where *cur_RID* is the RID of the current row in the *age = 35* list.

If the access path of the driving table is a table scan, then the rows will be accessed in RID order for efficient sequential I/O. In this case, we can use one local predicate to restrict the RIDs if the driving table is to be changed to an inner table. For example, if we decide to change the driving table after the row with RID value 100 has been processed, then we can add a local predicate "*RID > 100*" to filter out the rows with RIDs less than or equal to 100.

Note that the positional predicate used to prevent duplicates is different from the cursor needed to maintain the status of a table scan. When a driving table is switched to an inner table, usually an index access path is preferred over a table scan. In such a case, a predicate must be used to prevent duplicates from being generated. At the same time, the original cursor is also needed, in the event that this table is switched back to a driving table again and the original scan needs to be resumed. By exploiting the access order and applying an additional predicate, the driving table will become switchable with other inner tables[3]. The algorithm for changing the driving table is presented in Figure 3. We use the cost functions defined in Section 3.2 to evaluate different possible plans. Step 1 in the algorithm is to control the frequency of checking the driving table. Users can decide the value of "*c*"

[3] Please note that reordering driving tables may change the implicit sort order provided by driving tables. If a sort order needs to be maintained, we need to add a sort operator at the end of this pipeline.

according to different characteristics of queries and data sets. Both step 2 and step 3 require the estimated leg cardinalities and join cardinalities in cost calculations. We will describe how we estimate these values at run-time in the next section.

```
REORDER_DRIVING_TABLE() :
1: if a batch of c rows has been produced by this driving leg
   then
2:    Estimate the remaining work of the current plan, based
        on what has been observed;
3:    Estimate the cost of other plans with different driving legs;
4:    Pick the best (cheapest) one;
5:    Adjust the join orders according to the new plan;
6:    Add extra local predicates to the old driving leg;
7:    Reset the scan of the new driving leg;
8: end if
```

Figure 3: Reordering Driving Table

4.3. Run-time Monitoring

The goal of dynamic run-time reordering is to adapt to the data characteristics found while processing the query and to correct any erroneous estimates by the optimizer. This requires gathering actual run-time information, such as the incoming and outgoing number of rows resulting from both join predicates and local predicates. By monitoring these numbers, we can get more accurate and more recent estimations than those from a static optimizer. The cost derived using these run-time statistics also better reflects the characteristics of the data being processed, which enables the query processing to choose a better join order. Thus, run-time monitoring is important for adaptive join reordering. We will introduce how these estimations are calculated in this section.

4.3.1. Local Predicate Selectivity

At run-time, we monitor the number of rows processed by local predicates. Figure 4 shows the flow of rows through a single table T with n local predicates, LP_1 to LP_n. For a local predicate LP_i, the number of incoming rows is I_i and the number of outgoing rows is O_i. For the whole table T, I is the number of incoming rows from previous tables, and O is the number of rows output to the next table.

When T is the driving table, the value of I is 1. Figure 4 shows a common case in which a local predicate is applied by an index scan. We describe the case when index applies a join predicate in Section 4.3.2. If the access path is a table scan, we can ignore the first predicate in the figure. From the counters in Figure 4, the selectivity of a local predicate used for an index scan, S_{LPI}, can be estimated by the following equation:

$$S_{LPI} = \frac{O_1}{I_1 * C(T)} \tag{5}$$

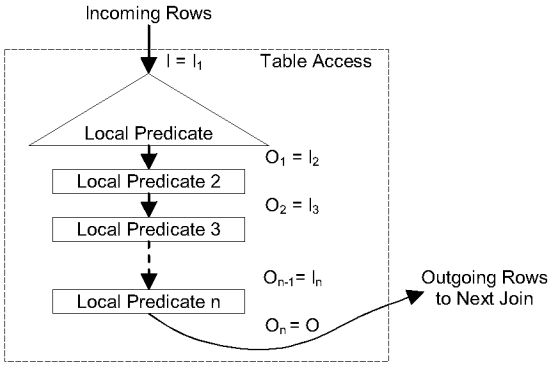


Figure 4: Monitoring Local Predicates

Please note that the first local predicate filters not the incoming rows, but the Cartesian product of the incoming rows and the current table. Hence the effective incoming row count is $I_1*C(T)$ for this predicate. If no local predicate is applied by an index scan, the value of S_{LPI} is just 1. The selectivity of the rest of the local predicates, S_{LPR}, can be estimated together as:

$$S_{LPR} = S_{LP2} * S_{LP3} * \cdots * S_{LPn} = \frac{O_n}{I_2} \tag{6}$$

During query processing, the local predicates S_{LP2}, …, S_{LPn}, are applied together to filter rows. For our purpose, they can be treated as one composite predicate, and we do not need the selectivities of individual local predicates. So we only need to measure the selectivity of the combined local predicates from the values of O_n and I_2. Note that since the measurement in Equation (6) is the combined selectivity, correlations among local predicates will not affect the accuracy of the estimate.

4.3.2. Join Predicate Selectivity

The selectivity estimation of a join predicate is very similar to that of local predicates. If there is only one join predicate, and it is used as the first predicate, then we can use the following equation to estimate the selectivity S_{JP} of this join predicate:

$$S_{JP} = \frac{O_1}{I_1 * C(T)} \tag{7}$$

If there is more than one join predicate, as shown in Figure 5, the selectivity of the rest of the join predicates will be calculated using the same formula as Equation (6):

$$S_{JP} = \frac{O_2}{I_2} \tag{8}$$

In Equation (8), only counters of incoming and outgoing rows are used, since the rest of the join predicates function the same way as local predicates.

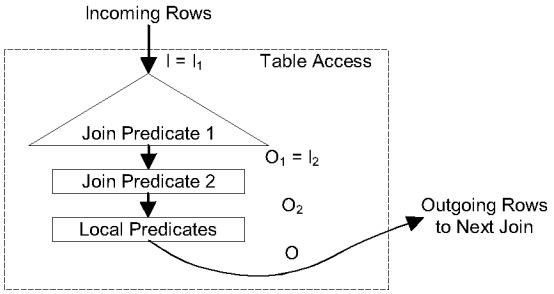


Figure 5: Monitoring Join Predicates

4.3.3. Leg Cardinality

In order to calculate the cost of different candidate join orders, it is necessary to obtain the leg cardinalities. The leg cardinality of table T can be calculated by:

$$C_{LEG}(T) = C(T) * S_{LP}(T) = C(T) * S_{LPI}(T) * S_{LPR}(T) \quad (9)$$

where $C(T)$ is the cardinality of the base table; $S_{LP}(T)$ is the selectivity of all local predicates; $S_{LPI}(T)$ is the selectivity of the index predicate(s) if a local predicate is used in the index scan, and $S_{LPR}(T)$ is the selectivity of the rest of the local predicates. If there are no local predicates used in an index scan, S_{LPI} is defined as one. One assumption here is that the base table cardinality can be obtained from a DBMS. This is typically done via statistics collected at insertion time by a database statistics-gathering utility. For the initial driving table, there is only one scan over its index, and $S_{LPI}(T)$ cannot be estimated accurately. In this case, we may obtain the selectivity estimation from the optimizer.

Our technique for estimating local selectivities and leg cardinalities can effectively avoid the cardinality estimation errors caused by correlations among columns.

Example 2:

```
SELECT o.Name, c.Year
FROM  OWNER o, CAR c,
WHERE c.OwnerID = o.ID AND
      c.Make = 'Mazda' AND  c.Model = '323' AND
      o.Country3 = 'EG' AND o.City = 'Cairo';
```

In Example 2, the predicates `c.Make = 'Mazda'` and `c.Model = '323'` are correlated. When this query is processed by the DBMS, the estimated selectivity of the `Make` predicate is 0.076 and the estimated selectivity of the `Model` predicate is 0.016. Based on the independence assumption, the selectivity of both predicates would be 0.076 * 0.016 = 0.00122. However, the correct selectivity of both predicates is actually the same as the selectivity of only the `Model` predicate, which is 0.016. The difference between the estimate and the actual is more than thirteen times. This error can be avoided using the monitored information. Suppose that we access the `Car` table using a table scan, so that the monitored selectivity includes both predicates, and so the correlation is included in that monitoring. In this case, the estimated combined selectivity will reflect the actual value. If the access method is an index scan on the `Model` column, then the optimizer will estimate the index selectivity, S_{LPI}, to be 0.016. When we measure the local selectivity, S_{LPR}, the measured value will be 1, since the rows satisfying the `Model` predicate also satisfy the `Make` predicate. The final combined selectivity, $S_{ILP} * S_{LPI}$, is still 0.016, which is the correct value.

4.3.4. Join Cardinality

For an inner table, we can easily calculate the value of the join cardinality by monitoring the incoming and outgoing rows. If the incoming row count is I and the outgoing row count is O, then the join cardinality of table T can be estimated by:

$$JC(T) = \frac{O(T)}{I(T)} \quad (11)$$

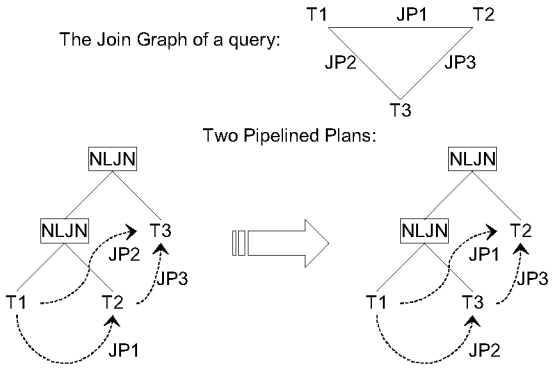


Figure 6: Join Cardinality Change Illustration

When switching the join order of two inner tables, we need to consider the fact that the available join predicates may change. Consequently, the join cardinality should be adjusted accordingly. We use Figure 6 to illustrate how join cardinalities should be adjusted. Figure 6 shows the join graph of three tables, with three join predicates, JP1, JP2 and JP3. When we change the join order from the left plan to the right plan in the figure, the join predicates on T_2 and T_3 will be changed. Accordingly, the new join cardinality of T_3 is adjusted to: $JC_{new}(T_3) = JC_{old}(T_3) / S_{JP3}$, while the new join cardinality of T2 is adjusted to: $JC_{new}(T_2) = JC_{old}(T_2) * S_{JP3}$.

For a driving leg, there is no join cardinality measurement. If the current driving leg is to be positioned as an inner leg, we need to estimate the join cardinality for the old driving leg. To do this, we use the join selectivity of the join predicates between this driving leg and other inner legs. Let us assume that we want to change the driving leg from T_1 to T_2 in the left plan of Figure 6. In the original plan, there is a join predicate, JP1, between T_1 and T_2. Since we have the estimation of the leg cardinality of T_1 and the selectivity of the join predicate JP1, we can estimate the join cardinality of T_1 as:

$$JC(T_1) = S_{JP1} * C_{LEG}(T_1) \quad (12)$$

4.3.5. Run-time Monitoring Summary

With all of the above monitoring and estimations, we can calculate the cost of any join order and choose the best one. While monitoring the incoming and outgoing rows, we calculate estimations using the counts over the latest "*w*" incoming rows. At the same time, we can choose the simple average or the weighted average to combine history and current measurements. We call the parameter "*w*" the *history window*. The principle used to decide the value of "*w*" is to make the reordering flexible enough to adapt to data changes, while at the same time preventing short-term fluctuations. We can monitor fluctuations and adjust window sizes adaptively during run-time. In this paper, we assume that the windows size is predetermined and unchanged during query execution.

5. Experimental Results

To evaluate the proposed technique, we implemented the prototype of adaptive join reordering in a leading commercial DBMS. The DBMS was able to estimate table cardinalities via statistics giving table sizes and average row sizes, and the data value distributions were assumed to be uniform during optimization. In our experiments, we used the DMV data set obtained from IBM [5, 22]. The data set contains information about cars, owners, demographics, and accidents. There are four tables in the data set with data skews and correlations among columns. In our experiments, we used the data with 100K owners. The tables and their cardinalities are shown in Table 1. We used five query templates whose query execution plans generated by the optimizer were mostly pipelined index nested-loop joins. The queries generated were all 4-table joins with different local predicate combinations using these templates. We also considered the TPC-H benchmark queries, but only a few queries had pipelined execution plans. Furthermore, unlike the DMV data, TPC-H does not have skewed data. Therefore, we did not use TPC-H in our evaluation.

Table 1: Tables in the DMV Data Set

Table	Cardinality
Owner	100,000
Car	111,676
Demographics	100,000
Accidents	279,125

By default, we used a check frequency ("*c*") of 10 and history window size ("*w*") of 1000 in our experiments. The experiments were performed on a lightly-loaded IBM Power PC with 1.4GHz CPU, 16GB memory. The DBMS was configured with a 400MB buffer pool. Each query was executed 5 times, and the average elapsed time was measured.

5.1. Reordering both Inner and Outer Legs

Figure 7 shows a scatter plot of the elapsed time of about 300 queries from the 5 query templates we used. In the figure, the X-axis is the query processing time without applying our adaptive reordering technique. The Y-axis is the query processing time with the join reordering of both driving and inner legs. Any point below the diagonal shows an improvement.

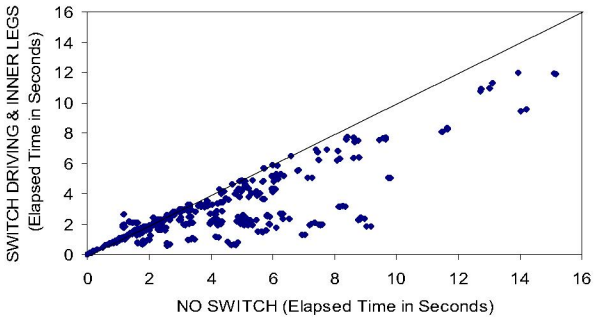


Figure 7: Scatter Plot of Elapsed Time

In Figure 7, with a few exceptions, almost all of the queries had significant performance improvements. The speedup was up to 7 to 8 times. The improvement of the total elapsed time of all the queries was more than 20%. If we count only the queries that had their join orders changed, the total improvement was about 30%. This figure demonstrates the robustness of using our adaptive reordering technique in coping with insufficient statistics and incorrect assumptions.

We next separate the performance of reordering only inner legs and only driving legs. Also, there were a few queries (less than 10) that showed small performance degradation. We will explain the reason for this.

5.2. Reordering Only Inner Legs

In this experiment, we demonstrate how adaptive join reordering can improve the performance by changing only the join order of inner legs.

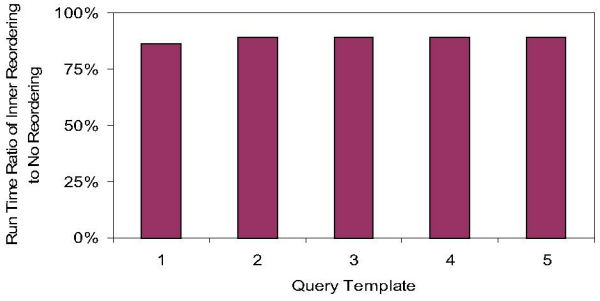


Figure 8: Reordering Inner Legs

Figure 8 shows the normalized elapsed time of the executions of queries from 5 templates. The elapsed time in a template is the average elapsed time of processing all queries in this template. The elapsed time when only inner legs can be switched is shown as a percent of the elapsed time without any order changes. This figure demonstrates that adaptive join reordering could identify incorrect join

orders and correct them according to the observed join cardinalities. Just reordering inner tables improved the performance by 10% to 20%, for those queries whose join order was changed.

5.3. Reordering Driving Legs

Adaptively reordering driving legs is more aggressive than just reordering inner legs, since normally the driving leg is the dominant factor for the execution time of a query. Thus, adaptively and correctly reordering driving legs could give us significant performance improvements. The experimental results of reordering driving legs are shown in Figure 9, again as a percent of the cost with no reordering.

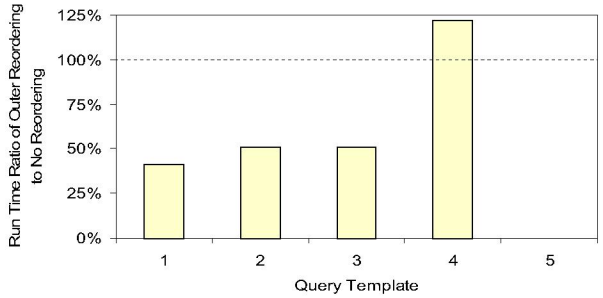


Figure 9: Reordering Driving Legs

Although there were many queries in Template 5 for which join orders of inner legs changed, their driving legs were not changed by our algorithm. So, there is no result presented for this template. All other templates, except Template 4, showed significant performance gains. In these queries, the average elapsed time with our adaptive reordering technique was less than 50% of the time used by the original DBMS. As for Template 4, a slight degradation was caused by an incorrect index access path chosen by the optimizer. Although the reordering algorithm figured out a better driving leg, the index scan chosen by the optimizer for the new driving leg was not optimal due to lack of detailed statistics.

We use the predicates on the `Owner` table in Example 3 to illustrate this. There are two predicates on the `Owner` table. Because the optimizer assumes a uniform distribution, it chose an index access on the `country3` column. However, due to the actual skew of the data, almost one third of the table would be scanned. This index scan was much more expensive than the index scan on the `city` column. Incorporating the cost of an index scan into the cost calculation, and being able to choose different index scan access paths at run-time, will be an important future extension to our adaptive join reordering technique.

Example 3:

```
SELECT o.name, a.driver
FROM Owner o, Car c, Demographics d, Accidents a
WHERE c.ownerid = o.id AND o.id = d.ownerid AND
      c.id = a.carid AND c.make = 'Chevrolet' AND
      c.model = 'Caprice' AND o.country3 = 'US'
      AND o.city = 'Augusta' AND d.age < 52;
```

The DBMS that we used provided a tool to collect more sophisticated statistics, such as data distributions and frequent values. The experiments with these statistics collected also demonstrated huge improvements with our reordering techniques, with up to two-fold speedups. Due to space limit this result is not shown in this paper.

5.4. Overhead and Thrashing

Using queries whose join orders were not changed by the adaptive reordering algorithm, we observed that the average overhead of monitoring and checking for reordering was 0.68% and 0.67% for inners and outers respectively. This is a very low overhead considering that we checked reordering every 10 incoming rows for each leg, i.e., the reordering check frequency c was 10.

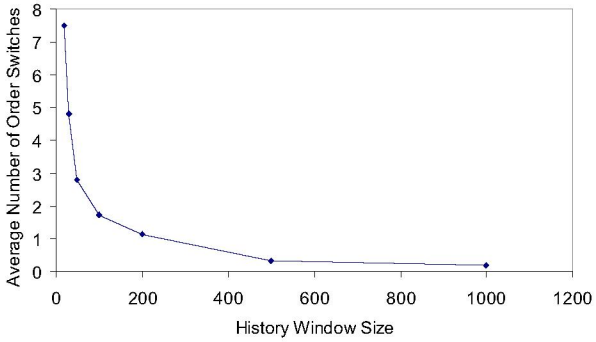


Figure 10: Reordering and History Window Size

We also measured the number of join order switches with respect to different history windows sizes "w". The result is shown in Figure 10. When the window size was small, we observed dramatic fluctuations without noticeable performance improvement. When the window size was over 500 rows, the performance and the reordering became very stable.

5.5. Reordering with More Tables

To test our technique with a larger number of joins, we extended the DMV data by adding two tables, `Location` and `Time`, both of which can be joined with the `Accidents` table. There were 100 queries with six table joins in this experiment, and the performance comparison is shown in Figure 11. The degradation of several queries was also caused by incorrect index selection, as explained in Section 5.3. Other than that, most of the queries demonstrated performance speedups up to 8 times.

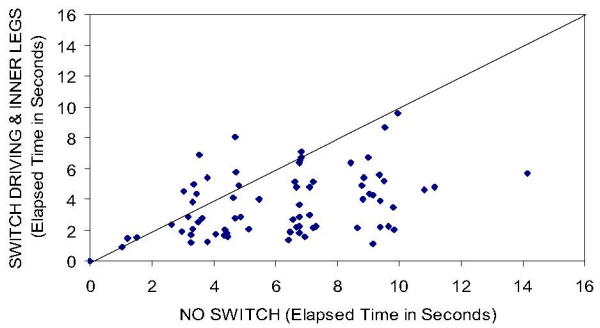


Figure 11: Scatter Plot of Six-Table Join Reordering

6. Conclusions and Future Work

Adaptive join reordering is a light-weight run-time re-optimization technique that improves both robustness and adaptability of a query processing system. This technique dynamically arranges the join order of pipelined join execution plans according to the observed data characteristics at run-time. Since the adaptation is achieved by changing the pipeline itself, we minimize the bookkeeping and avoid making routing decisions at the row level. Our duplicate prevention technique is simple and effective for supporting switching both inner and driving tables. Experimental results demonstrated the significant benefits and robustness of our proposed adaptive join reordering technique.

Although we focused our adaptive join reordering on nested-loop joins, it is not difficult to see that this technique can be extended to pipelined hash joins as well. Since pipelined joins can be pipelined parts of a bigger plan, we are investigating the integration of our technique with re-optimization techniques [20, 22]. Also, choosing index scan access paths dynamically at run-time will be an important extension of our adaptive reordering technique.

7. Acknowledgement

We would like to thank Hamid Pirahesh for initiating this research and for his inspiring discussions.

8. References

[1] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In DIS '96, pages 208–219, Washington, DC, USA, 1996.

[2] G. Antoshenkov and M. Ziauddin. Query processing and optimization in oracle rdb. The VLDB Journal, 5(4):229–237, 1996.

[3] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In SIGMOD '00, pages 261–272, New York, NY, USA, 2000.

[4] S. Babu, and P. Bizarro. Adaptive Query Processing in the Looking Glass. CIDR 2005: 238-249.

[5] S. Babu, P. Bizarro , and D. DeWitt. Proactive re-optimization. In SIGMOD'05, Baltimore, Maryland, 2005.

[6] S. Babu, R. Motwani, K. Munagala, I. Nishizawa and J. Widom. Adaptive ordering of pipelined stream filters. In SIGMOD'04, Paris, France, 2004.

[7] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In ICDE '05, pages 118–129, Washington, DC, USA, 2005.

[8] M. W. Blasgen, M. M. Astrahan, et al. System R: An Architectural Overview. IBM Systems Journal 38(2/3): 375-396 (1999).

[9] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. ACM Trans. on Database Systems,24(2):177–228, 1999.

[10] S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. ACM Trans. on Database Systems, 9(2): 163-186, 1984.

[11] R. L. Cole , and G. Graefe. Optimization of Dynamic Query Evaluation Plans. SIGMOD Conference 1994: 150-160.

[12] A. Deshpande. An initial study of overheads of eddies. SIGMOD Record, 33(1):44–49, 2004.

[13] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In VLDB, pages 948–959, 2004.

[14] G. Graefe, and K. Ward. Dynamic Query Evaluation Plans. SIGMOD Conference 1989: 358-366.

[15] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. ACM Trans. on Database Systems, 9(3):482–502, 1984.

[16] J. Hellerstein. Optimization techniques for queries with expensive methods. ACM Trans. on Database Systems, 23(2):113–157, 1998.

[17] IBM, IBM Red Brick Warehouse Administrator's Guide. http://publibfp.boulder.ibm.com/epubs/pdf/c1873950.pdf

[18] Y. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In SIGMOD, May 1991.

[19] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to source properties in processing data integration queries. In SIGMOD '04, pages 395–406, New York, NY, USA, 2004.

[20] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In SIGMOD'98, Seattle, Washington, June, 1998.

[21] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In VLDB 1986, pages 128–137.

[22] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In SIGMOD'04, pages 659–670, 2004.

[23] C. L. Monma and J. B. Sidney. Sequencing with series-parallel precedence constraints. Mathematics of Operations Research, 4:215–224, 1979.

[24] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In ICDE'03.

[25] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In VLDB'01, pages 19–28, 2001.

[26] T. Urhan and M. J. Franklin. Xjoin: A reactively scheduled pipelined join operator. IEEE Data Eng. Bull., 23(2):27–33, 2000.

[27] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In SIGMOD'98, pages 130–141, New York, NY, USA, 1998.

[28] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In SIGMOD'04, pages 431–442, 2004.