ADAPTIVE QUERY PROCESSING

Sandeep Kathula
Indiana University
Bloomington
3209E 10TH ST
Bloomington
IN 47408
Ph: 925-487-6153
skathula@iu.edu

Venkata Koppu
Indiana University
Bloomington
3209E 10TH ST
Bloomington
IN 47408
Ph: 812-349-8793
vkoppu@iu.edu

Rohith Nedunuri
Indiana University
Bloomington
427 W.Hoosier Court
Bloomington
IN 47404
Ph: 812-360-7334
rnedunur@iu.edu

Ramprasad Bommaganty
Indiana University
Bloomington
427 W.Hoosier Court
Bloomington
IN 47404
Ph: 812-272-1368
rbommaga@iu.edu

Abstract:

The paper wishes to discuss the different techniques that have been put forward by years of research on changing the paradigm of optimize and then execute. The primary cause of the recommendation for a change in paradigm has been due to the massive explosion of data over the years and the accompanying problems such as insufficient statistical information about the data available .Hence during compile time the Query optimizer may not have enough statistics needed, knowledge of system resource etc. to produce an optimal Query Plan Thus, the concept of Adaptive Query Processing came into existence as an alternative to the traditional query optimization techniques of the relational database systems.

1. Introduction:

The relational data model trumped over the other proposed database models due to its multitude of benefits such as physical data independence, multiple storage formats and execution plans. Also, the fact that the model transformed a declarative, logic based query into an algebraic query evaluation tree contributed to its simplicity and ease of use.

However, the relational model has its own limitations. The relational DBMS query processing is reliant on factors such as a stable environment, sufficient statistical information and fewer correlations between the attributes. As the scope of databases have begun to expand, the traditional optimize then execute process falls short on certain counts like querying over data streams, providing support to data mining and OLAP operations and also processing XML data.

In contrast, the aggregation of techniques called the adaptive query processing uses the feedback obtained from query execution to solve the above mentioned problems.

1.1 Query processing in relational database systems

The following section discusses the existing query processing procedure in relational DBMS. It is implemented in three major stages:

a. **Statistics generation:** This process includes profiling the relation instances, collecting information on cardinalities and number of unique attributes and also generating histograms over certain fields. It is usually done offline and the commands used to perform this process are RUNSTATS and UPDATE STATISTICS.

b. **Query Optimization:** The query optimizer is used to generate the query plan. A query plan is a tree consisting of unary or binary relational algebra operators over the data that has to be processed. The process of query optimization combines cost estimation, pruning heuristics and exhaustive enumeration. Cost estimation calculates the cost associated with running each of the subexpressions of the query while pruning heuristics and exhaustive enumeration reduce the overall search space and list all the available data respectively.

c. **Query execution:** The two important aspects associated with query execution are pipelining the computations and scheduling them in a query plan. The process of each operator processing the tuples of its sub operators and propagating it to its parent is known as pipelining. Pipelining leads to better response times and higher throughputs. For query plans that are complex and use too much resources, materializing the intermediate results and using them as the input to the next stages might work efficiently.

Scheduling computations in a query plan is important to the performance of a DBMS. There are two basic approaches to scheduling computations, namely: iterator and data driven. While the iterator approach uses methods such as open(), close() and getNextTuple() to perform scheduling by traversing through the query plan tree till there are no more tuples to compute, the data driven approach performs scheduling based on the rates of filling up and emptying of the input and output queues which are controlled by the operators. Data driven approach finds its application in parallel database systems.

1.2 Motivations for AQP

The query optimizers of modern systems are more powerful and consequently are able to perform a better search of the query plans by relying less on pruning heuristics. But the traditional query processing methodology does not perform quite well on the face of the following conditions:

a. Unreliable cardinality estimates: Measuring statistics in the real world scenarios can be taxing and more often than not the collected statistics are not error free.

b. Queries with parameter markers: Since form inputs are used to gather data from non-technical users, the pre computed plans for parameter markers might not be efficient at all.

c. Dynamically changing data, runtime and workload characteristics: The query plans might change during the runtime while processing data streams and the allocation of resources also varies as a result of the allocated workloads.

d. Complex queries involving many tables: Usage of heuristics leads to an increase in the estimation errors in cases where there are many tables and the data is too complex.

e. Interactive querying: The traditional query processing methods do not work in an interactive environment.

f. Need for aggressive sharing: The traditional query processing system does not support state sharing among the queries but with regards to data streams where a huge number of queries are executed, they become an important feature.

There have been two approaches to solve the above problems:

1. Designing an application level processor to override the local DBMS optimizer and give user specified 'hints' on the kinds of queries to execute.

2. The more useful approach has been to develop adaptive query that utilize feedback during runtime to deliver efficient outputs.

2. Conventional Optimization Techniques

2.1 Query optimization

Selection ordering and join enumeration are the two methodologies that are considered for cost based optimization techniques.

A predicate is an expression that evaluates to true or false. In this context, a predicate is a subexpression of a query that becomes true when a tuple satisfies the predicate condition.

2.1.1 Plan Space

Selection ordering is the process of determining the order of executing the predicates based on the costs associated with each of them. If a query is considered as a conjunction of "n" commutative predicates, S1, S2…,Sn to be enforced on a set of tuples belonging to a relation R, then the cost of each predicate can be c_i and the probability that a tuple satisfies the probability can be p_i . There are mainly two different types of query plan evaluations:

a. Serial plans: The predicates are executed in a serial order which represents the order in which the predicates should be applied to the tuples of the relation.

b. Conditional plans: The execution of a predicate depends on the prior execution of another predicate. Conditional plans are usually a generalization of the serial plans wherein the values of certain attributes lead to different predicate order executions.

2.1.2 Static planning

Predicates can be categorized into the following:

a. Independent predicates: The predicates are assumed to be independent of one another by the query optimizers and rank ordering of the predicates are calculated to determine the order of execution. The rank ordering is computed using the formula $c_i / (1 - p_i)$ and the optimal serial order has a time complexity of O(n(log n)).

b. Correlated predicates: It is presumed that if there are correlations between the predicates then it leads to an NP Hard problem. However, by assuming that the correlations between predicates can be computed using the correlation information, an algorithm called the Greedy algorithm can be used which calculates the probability of a predicate given the previously chosen predicates are true.

2.1.3 Multi-way join queries: Plan Space

As mentioned earlier a SPJ (Select-Project-Join) query is one form of restricted query for which a query plan is prepared. The preparation of plan space rests on picking three main components, namely, access methods, join order and join algorithms.

Access methods refer to the method of reading the necessary tuples. There are many choices available to read the data such as a direct table scan, setting up and index and scanning over it or by using materialized views.

According to the accepted definition,

"A join order is a tree with the access methods as leaves, each internal node represents a join operation over its inputs."[3]

Simply put, it determines the order in which the different relations can be joined using the operators that are available. There are binary and n-ary join operators available. An important point to note is that if the predicates of a query form a cycle, then a spanning tree of the predicates will be considered and the eliminated predicates will later on be added on as "residual" predicates.

There are many join algorithms can be used to execute a join. For example, Nested loop join, merge join, hash join, block nested join and sort merge join are some of them. Join algorithms are usually picked on the basis of access method that has been used. For data streams, a pipelined join operators are preferred and an index nested loop join or sort merge join is preferred for an access method which is using an index.

2.1.4 Pipelining versus Materialization

For a Select-Project-Join query, there are two types of query plans:

a. Non-pipelined plans: These plans contain at least one blocking operator that saves the intermediate results of an operation and passes it on to the next operator as an input. The process of saving the intermediate results at a point is known as materialization and that state is known as the materialization

point. A blocking operator finds its application when a sorting or hashing has to be performed.

b. Pipelined plans: As discussed before, pipelining is the process of sending the output of one operator to the subsequent operator as an input and finally deriving the output on the fly. A pipelining can be partly pipelined or fully pipelined. A hash join operation is partly pipelined because it performs a build operation for the hash table and the process is not actually completely pipelined. On the contrary, a fully pipelined operator might be a symmetric or doubly pipelined hash join operator that takes in an input and then builds it with respect to a relation while also probing it with the other input relations. This method leads to a faster response time and also is quite efficient when the data is in the form of streams from several inputs.

2.2 Choosing an effective plan

The ultimate goal of a query optimizer is to choose the minimal-estimated-cost plan from its plan enumeration space. The most essential aspect of the cost estimation is selectivity estimation where given a set of inputs and an operation, the cardinality is predicted. Modern DBMS systems rely on histograms that are created offline to record distributions of selections and join key attributes. But correlations between attributes may cause the histograms to be difficult to align or intersect

The following ideas contribute to an effective plan:

a. Robust query optimization: It is always better to choose plans which have a low cost estimated for different parameter values for an optimizer rather than a plan that has low estimates for the given expected parameter values. Query plans can be made robust by using operators such as MJoins or eddies.

b. Parametric Query optimization: Certain planning decisions are postponed to runtime whereby instead of a single robust plan, there are plans for different situations. The advantage of this approach is to allow intermediate results as parameters to make decisions.

C. Inter-Query Adaptivity: Self tuning and Autonomic Optimizers: Self-tuning histogram is one technique where the query execution is passively observed to collect the knowledge and later on use these statistics to better predict the selectivity estimates in the future. In projects like LEO (Learning optimizer), arbitrary subexpressions are monitored while executing and the difference in actual observed selectivities with optimizer's selectivities are computed. This difference is then used to adjust the estimates of optimizer for future queries.

3. Adaptive Query Processing

The aim of Adaptive Query Processing is to find logical and physical execution plans that are most suited to the runtime conditions. This can be done by interleaving the query execution in middle and by modification of the query execution plans. The major dissimilarity between various adaptive techniques are the differences in the way they interleave the query execution. Some of the adaptivity techniques like choose nodes and mid-query reoptimization interleaves the query execution several times whereas eddies operator interleaves them at points even where they are not clearly distinguishable. Interleaving is done because the information for choosing the optimal query execution plan may not be available in the beginning of the query execution in real time which is mostly observed in data streaming. So, by interleaving query execution based on then available data query execution plan is chosen at runtime. Traditional Database engines are mainly optimized for disk based I/O's such that they always tend to decrease the number of I/O's required and supplies data to DBMS at very high rates. This however will greatly affect the adaptation opportunities when there are wide range sources of data and data streams. To address these problems three new operators are introduced which offer greater flexibility and more opportunities for adaptability. But the only problem is that the require lot of memory. The three operators are:

a. Symmetric Hash Join**:** Hash join is not well suited for adaptive query processing where the data is in the form of data streams or when data comes from wide range of sources. In hash join until hash table is completely built on the inner relation (build phase) it cannot be matched with data in other relation (probing phase). This makes it unsuitable for handling data streams and wide area data sources. The build relation on which hash table is to be built is to be chosen in advance which may not be possible. In contrast to hash join the symmetric hash join does not wait until it obtains all tuples from one of its input relations before producing results. It maintains two hash tables one for each relation. A tuple is retrieved from one of the two relations and is inserted into appropriate hash table (hash table for that particular relation from which the tuple has come from). If tuple is obtained from inner relation it matches to the tuples from hash table of outer relation and if the tuple is from outer relation it is matched with tuples from hash table of inner relation. This operator can have maximum adaptivity since it is symmetric (it can process data from both the input relations depending on the availability of data from either relations). Upon this proposal lot of work is being done and new operators are being introduced. Some of them are XJoin and doubly pipelined hash join which adapted this symmetric hash join to multi-threaded architecture using producer consumer threads. Later another operator ripple join has been derived from symmetric hash join which helps in improving correctness of aggregations. This can be used for both equijoin and non-equijoin. But the main problem is that it requires large amount of memory for having hash tables on both the relations.

b. Eddy operator: With eddy operator we can have high level control on query execution plans. Here the main idea is to process routing tuples through operators and to adapt by changing the order in which tuples are routed. The eddy operator is responsible for making the routing decisions. It executes the query by routing the tuples through operators. The tuples which are processed by all the operators are sent to output. Eddy can adapt to changing data by simply changing the routing order of the tuples. Choice of operators should be chosen in advance. Symmetric hash join is one of the operator

mostly used because it can provide immediate feedback for determining operator selectivity. On the other hand sort-merge operators are not used as they cannot produce outputs till they get the complete input. To route the tuples to the correct operator tuple signature is used which contains set of base relations that the tuple contains and operators through which it is passed. Two bit sets ready and done can be maintained by each and every tuple where they have the information about through which operators the tuple has already processed by and through which operators it has to be processed by. Eddy has a routing policy (set of rules which states the routing destination among many available alternatives). This will be directly impacting the performance of this operator. Eddy can maintain statistics at very fine granularity because it processes each and every tuple. These statistics are used for building or changing the routing table. It uses the routing table to find valid destinations of the tuple and it can process to appropriate operators.

c. **MJoins:** MJoin also known as n-ary symmetric hash join. It is used to join more than two relations by allowing tuples from the relations to come in interleaved fashion. It has many advantages over widely used binary join especially when there are data streams. It builds a hash index on every join attribute of every relation of the query. The number of hash tables can range from 2(n-1) to n depending on if the join is on different attributes or on same attribute. It uses light weight tuple router to route tuples from one hash table to another. Eddy operator can also be used for this purpose. When a new tuple arrives from a relation it is sent to hash table based on hash value and is matched or probed in some order with other relations. This order is called probing sequence. For example three hash table are built on a, b, c attributes of p for which the probing sequence can be x->y->z. That means it is first probed with x and the intermediate result be probed with y and then the next intermediate result obtained is probed with z. When the queries have sliding windows it has high performance when compared to binary join especially in data streaming. Because of this reason large amount of work in data streaming is being done in the field of MJoins. It is very easy to adapt MJoins because change in query plan leads to change in probing sequence. Handling memory overflow in MJoins is harder when compared to that of symmetric hash join. To solve this problem a technique called coordinated flushing has been developed which maximizes output rate while allowing tuples to spill back to disk. But this technique can only be applied when all joins in a query are on same attribute.

3.1 Adaptivity Loop

In query adaptivity query execution is supplemented with adaptivity loop which monitors query processing and adapting it. It has four components which executes in a line or in parallel. They are:

a. **Measure:** It monitors parameters which are appropriate for goals

b. **Analyze:** When goals are given it analyzes how to meet the goals.

c. **Plan:** Based on the analysis it makes decisions about how the behavior is to be changed

d. **Actuate:** Decisions made are to be executed.

4. Adaptive selection ordering

Selection ordering is determining the order of applying set of selection predicates (age=10, sid<100, etc.) to all the tuples of a relation so as to find the tuples which satisfy all the predicates. It is one of the most serious problems in query optimization. It received more importance in the fields of web, high speed data streams and sensor networks. It is also studied in the fields of fault detection and machine learning. Moreover multi way join queries have same behavior as selection ordering. If the query is conjunction of n predicates which give same answer even if their order is changed to be applied to all the tuples from a relation. We have costs of a predicate S_i as C_i and probability that tuple satisfies a predicate is $P(S_i)$. In conventional query optimization techniques serial plans are used which specifies a single order in which predicates should be applies to the tuples.

4.1 A-Greedy

In adaptive selection ordering an algorithm called adaptivity greedy (A-Greedy) is used. It is based on greedy algorithm. It monitor the selectivity of predicates over recent past and also ensures that order used by optimizer is same as the one which has been chosen by greedy. It has mainly three components.

a. **Query Executor:** It executes the query over newly arriving tuples from data streams and wide range data sources by having same order as that of the current execution plan chosen by the query optimizer. It uses serial order of predicates similar to greedy

b. **Profiler:** It maintains a sliding window profile over the relation using random sample of tuples which did not satisfy at least one predicate and only on the tuples which are seen in recent past. It contains the results of evaluation of all query predicates over selected tuples. When a new tuple arrives from data streams it randomly decides whether to include or exclude this tuple into profile. If a tuple is to be included it is evaluated over set of predicates and results are included in the profile. It also removes older tuples which are not under sliding window. It also computes expected cost of evaluation for the predicates.

c. **Reoptimizer:** It makes sure that the query plan under execution is same as that chosen by Greedy algorithm. It continuously monitors Greedy invariant over profile tuples and if the invariant is violated it re-optimizes evaluation order.

A-Greedy assumes that the statistical properties of the data won't get drastic and sudden changes. It also assumes that data properties of near past are predictive of data properties in near future. It measures the properties of the operators by evaluating all the

predicates over random sample of tuples and summary of the results will be shown in matrix form. The total costs incurred are sum of costs for predicate evaluation and cost of matrix updation. One single update in a profile can lead to as many as $n^2/4$ entries in matrix. Reoptimizer will analyze continuously violations of greedy invariant using matrix view. It takes O(n) time. If any violation is detected then Greedy algorithm is used to construct new execution plan.

4.2 Eddies

Eddy operator can also be used in selection ordering. To execute a query one eddy operator and n selection operators (for selection of n commutative predicates) are instantiated. The eddy operator directs the tuples through selection operators. The selection operators outputs the tuple to eddy when predicate is satisfied or else it will discard the tuple. It contains a bitmap called done which has the information about the operators through which a tuple has been routed through. All operators which have done bit as false are valid destinations to the tuple. If all bits of done bitmap are set to true then the tuple is to be output from the eddy operator. Eddy also maintains the selectivity of each operator by maintaining list of tuples which are passed through the operator and the tuples returned by the operator.

5. Adaptive Join Processing

Adaptation techniques for join queries are notably more complicated to design and analyze in contrast to those for selective ordering for the following reasons:

a. The execution plan space is much larger and more complex compared to selective ordering which arises due to the fact that there are large number of possible join orders and the different kinds of join operators themselves (binary vs. n-ary, blocking vs. pipelined etc.).

b. The "stateful" nature of join operators which means that they have an internal state that depends on the tuples processed by the operator. Example, hash join operators build hash tables of their input relations. As a result of this parameter (internal state), the executing environment itself plays a significant role in estimating the nature of query execution. For instance, whether the data is being read from a local disk, is arriving in the form of a data stream or is being read asynchronously over the wide area network, can substantially change the nature and trade-offs of query execution.

To tackle this complexity, the research community has developed a diverse set of techniques designed for specific execution environments or specific join operators, or in some cases, both. These adaptation techniques apply in different underlying plan spaces, though the particular. To troubleshoot this complexity, various set of techniques were developed which are targeted at either specific execution environments or specific join operators, or in some cases, both. These adaptation techniques can be represented in three parts, roughly based on the space of the execution plans they explore:

a. History-Independent Pipelined Execution

b. History-Dependent Pipelined Execution

c. Non-pipelined Execution

In most of the database management systems query execution plan consist of both blocking operators and pipelines which can be adapted independently using different adaptive processing techniques.

6. Adaptive Join Processing: History-independent pipelined execution

6.1 Pipelined plans with a single driver relation

A driver relation is the primary relation that is joined to the other subsequent relations of the query. For example, in a three way join algorithm consisting of the relations R, S and T. Here, R can be considered a driver relation while S and T are the driven relations. The access methods of S and T determine the types of joins performed. If the driven tables are scanned then a nested loops join is used or if a hash index is performed on them, a hash join is used. The query plan execution is totally history independent meaning that the position of a tuple in the driving relation does not affect its join selectivity.

6.1.1 Choosing Driver relation and access method

The driver relation can be chosen among the n different relations for a query**.** Access methods for the driven relations influence the join costs and the join order and hence the plan space increases manifold when there are many permutations of access methods for the driven relations.

6.1.2 Reduction of join ordering to selection ordering

The process of join ordering of the relations in adaptive query processing can be reduced to a selection ordering, provided the access methods and the driver relation are given.

Let c_i and f_i denote the probe costs and the fanouts of the join operators on the given relations of the query then there are two different plans:

Plan: R -> T -> U where the execution cost = Build Cost + |S| x $(c_1+f_1c_2+ f_1c_2c_3)$

Plan: T -> R -> U where the execution cost = Build Cost + |S| x $(c_2+f_2c_1+f_1f_2c_3)$

Here Build cost is a constant that is the cost of building indexes on R, T and U.

Though there are quite a similarities between selective ordering and join ordering like cost functions there are certain differences that need to be considered:

1. Though the rank ordering algorithm of the selection ordering can be used for join ordering, it is important to note that the fan-outs for the driver relation are greater than 1.

2. There are precedence constraints that must be satisfied before the join operation for all the relations is performed. For example, a relation R cannot be joined with T if they do not have any shared join attributes.

3. The join costs for different tuples may be different based on the fact that if they belong to a hash join then it depends on whether the required partition is available in the main memory for execution.

4. The selection ordering method may be suboptimal when cache effects are used for certain operators and execution models.

6.1.3 Adapting the Query Execution

6.1.3.1 Adapting the choice of driver table or access methods

Switching the driver tables or access methods during the execution is computationally expensive and usually involves high costs due to complex duplicate handling procedures. For adaptations of access method, one method is to run the access methods in parallel for a while the selectivities and costs are measured simultaneously and later on one access method can be chosen for each driven table. The problems associated with this method are that it does not explore the various combinations of access methods possible and there is a significant overhead involved for duplicate elimination.

6.1.3.2 Adapting the Join order

A-greedy algorithm and eddies can be used for join order adaptation unless the driver relation is not changed mid-execution. But the important consideration is that the algorithms used must obey the precedence constraints mentioned in the earlier section.

6.2 Pipelined Plans with multiple drivers

Using single driver relation is computationally cost effective and simple but certain joins like symmetric joins and sort-merge joins need multiple drivers. The problem with selecting a single driver before the query execution is that the entire relation has to be available before the entire process can begin and it does not work with streams of data. This might cause a lot of overhead which proportionally depends on the size of the relation as well. Also, picking a right driver relation involves comparing not just the table sizes but also their join fan-outs with the driven relations. There are certain operators that overcome the given difficulties but still maintain the history independency with respect to query execution. Some of them are as follows:

6.2.1 n-ary Symmetric Hash Joins/ MJoins

An n-ary symmetric hash join or Mjoin performs a query execution that is history independent i.e., the input tuples determine the operations inside the hash tables and not the option chosen by the router. The probing sequences encapsulate the decision logic that specifies the order to probe the hash tables for tuples of each driver table.

6.2.1.1 Choosing the initial probing sequences:

The rank ordering algorithm and Greedy algorithm can be used to compute the probing sequences for the driver relations. However, this process is efficient enough only when all the hash tables fit in memory.

6.2.1.2 Adapting the probing sequences:

The probing sequences can be adapted during execution based on techniques such as A – Greedy algorithm or by using eddies. Statistics can also be shared among each of the probing sequences to improve the efficiency.

6.2.1.3 Driver choice adaption:

Controlling and adapting a driver relation to be used can contribute in the following ways:

a. Drivers can be switched over when there are breaks in the asynchronous data sources.

b. If there is a hash join involving two base tables having different sizes, then the query optimizer has to first determine the table which has a smaller size, lower memory consumption and lesser I/O's. Instead, an adaptive technique is to initially oscillate between driver tables until one of them reaches the End of File and then pick it as the smaller relation. Then an asymmetric hash join can be run on the relations. This process is not completely cost effective though.

c. Interactivity with the users combined with successful query execution is usually preferred over closed query executions. But the existing join operators cannot provide both simultaneously without tradeoffs. Hence, drivers can be alternated and the query processor can switch between symmetric hash join and asymmetric hash join to get pipelined results as well as better interactivity.

d. Based on improved selectivity estimation the drivers can be switched by looking up a hash table that has been built on them. SHARP is a technique to adapt choice of driver tables and access methods simultaneously.

6.2.1.4 Limitations to Mjoins

Let us assume a three relation query consisting of R, S and T relations. Then for a three way join of all the relations, if the order of arrival of tuples is r_1,r_2,s_1,s_2, s_3,t_1,t_2,r_3,t_3 then they can be divided into three partitions as follows : $R_1 = \{ r_1,r_2\}$, $S=\{ s_1,s_2, s_3\}$, $T_1=\{ t_1,t_2\}$, $R_2=\{r_3\}$ and $T2=\{t_3\}$. If the order of join execution is $R \bowtie S \bowtie T$ then the resulting subqueries are $R_1 \bowtie S \bowtie T_1$, $R_1 \bowtie S \bowtie T_2$, $R_2 \bowtie S \bowtie T_1$, $R_2 \bowtie S \bowtie T_2$. These subqueries are executed using left deep pipelined plans and every relation that arrived last will serve as the driver relation while execution. However, the problem associated with this kind of execution is that the intermediate results are not stored for reuse.

6.2.1.5 Limitations

a. Reuse of the intermediate tuples cannot be done as they are not stored and hence re-computation is performed every time the query is executed.

b. The number of query plans that an optimizer can use is restricted due to the fact that any new arrival must first join

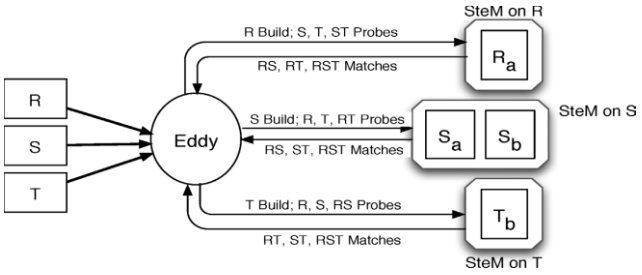

Figure -MJoins using STeMs

with the already arrived tuples and then with others, even though this order might be sub optimal. As intermediate tuples are not stored, any other alternative plans cannot be used.

6.2.2 State Modules(SteMs)

A SteM or a State Module is a unary operator that can be used with an eddy to allow for better adaptation flexibility than some other operators such as MJoins. There are a few features of STeMs that enhance its efficiency:

a. The atomicity and ordering of the build probe relations is not strictly enforced.

b. Probing of the other relations by the driver tuple is permitted not into the queries being built for the executing query but also into many other pre-built indexes, which are usually known as Access Modules.

c. In addition to finding driver relations from table scans or streams, the lookups into the pre-built indexes are also used.

There are basically two operators that can be used to implement the above mentioned features:

a. Access modules**:** An access module matches the tuples from the selected table and the probe tuples from the driver table and outputs them. However, it does not concatenate the probe tuples and the matching tuples. It just outputs the tuples of the indexed data source that have a match to the probe tuples.

b. State Module**:** A State Module is a data structure that accepts both build and probe tuples and performs the following operations: insert, search and delete. Contrary to access methods, STeMs concatenate the probe and matching tuples and output them.

6.2.2.1 Query execution and Hybridization

As previously mentioned, Hybridization is the process of changing the join operators during the execution like from a symmetric hash join to an asymmetric one back and forth. When a query is being executed, an eddy, a STeM on each base table and an access method on each base table are instantiated. Based on how the eddy chooses to route the tuples, the query optimizer picks the access methods, join orders and join algorithms

6.2.2.2 Mjoins using STeMs

SteMs can be used to perform Mjoins by just exposing the hash tables as three different SteMs. An eddy operator can be used to route tuples from one hash to the other. The tuple exchange between the eddy operator and the SteMs leads to the formation of joined tuples. The following is the process that occurs: Let us assume that there are three base relations R, S and T and have indexes built on all the join attributes. Assume that R has the index on attribute a, T has index on attribute b and S has the index on attributes a, b which are also the join attributes for each relation.

When a tuple from the relation enters the system, it is routed through the eddy operator and it is probed on the SteM setup on the relation. The output is again used to route through the eddy and probe with the other join attributes of other relations.

6.2.2.3 Asynchronous Index joins using SteMs

An index join using SteMs can be performed in a similar way to MJoins but by using an index access method on a base relation say R, instead of the usual scan access method. By using this method, only those tuples of R will enter the system that have been probed by S and T already, thereby easing out the computation required. These tuples will again be routed through the eddy operator and the entire process follows similar to a Mjoin using SteMs.

Limitations to SteMs**:** The two routing issues associated with using SteMs are:

a. Running multiple access methods concurrently leads to incorrect numbers of duplicate tuples that are produced.

b. Routing asynchronously can produce missing results.

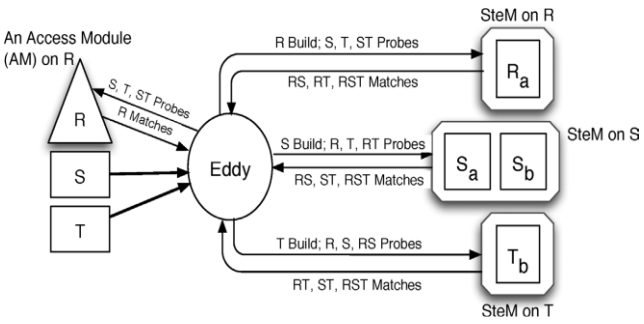

Figure - Asynchronous Index joins using STeMs

Post Mortem Analysis:

Horizontal partitioning is the division of the base relations into different partitions and then executing each one of them with the chosen operators. Using a MJoin or a SteM is similar to performing horizontal partitioning. Whenever a set of tuples enter the system contiguously and are routed identically through the different operators, then they are part of a partition of that relation.

Let us assume a three relation query consisting of R, S and T relations. Then for a three way join of all the relations, if the order of arrival of tuples is r_1,r_2,s_1,s_2, s_3,t_1,t_2,r_3,t_3 then they can be divided

into three partitions as follows : R_1 = { r_1,r_2}, S={ s_1,s_2, s_3}, T_1={ t_1,t_2}, R_2={r_3} and T2={t_3}. If the order of join execution is R ⋈ S ⋈ T then the resulting subqueries are R_1⋈S⋈T_1, R_1⋈S⋈T_2, R_2⋈S⋈T_1, R_2⋈S⋈T_2. These subqueries are executed using left deep pipelined plans and every relation that arrived last will serve as the driver relation while execution. However, the problem associated with this kind of execution is that the intermediate results are not stored for reuse.

Limitations:

a. Reuse of the intermediate tuples cannot be done as they are not stored and hence re-computation is performed every time the query is executed.

b. The number of query plans that an optimizer can use is restricted due to the fact that any new arrival must first join with the already arrived tuples and then with others, even though this order might be sub optimal. As intermediate tuples are not stored, any other alternative plans cannot be used.

6.3 Adaptive caching

In order to overcome the limitations of the existing system of MJoins which does not store the intermediate results for further execution, Adaptive caching is used. It is important to note that this approach is not entirely history independent since the decision to store certain intermediate results affects the execution state. Though this approach uses the Mjoin operator as the basic operator in the system. However, it differs from the Mjoins approach in the following ways:

a. It uses the A- Greedy algorithm to independently create probe sequences for each streaming relation.

b. The intermediate results can be added in the middle of the pipeline after the probe sequences for the streaming relations are chosen.

The working of the adaptive caching is possible through the use of an intermediate result cache. An intermediate result cache can be defined using: the relation X for which it is setup, the set of operators Y in the probing sequence. The notation is as follows: C^X_Y. The intermediate cache can be used in the following manner: The cached entries in the cache are a pair of the form (u, v) where u is the value of the join attribute and v is the result tuples that are generated after the probe sequence. If a similar kind of join attribute value is encountered during the execution of next tuple, then the existing matching tuple is already available. For example, if there is a pair (x,$(RST)_1$) then for other values where x=x_1, there exist pairs (x_1,$(RST)_1$). When a tuple arrives it is first looked up in the cache and if it is not available only then the probes are conducted.

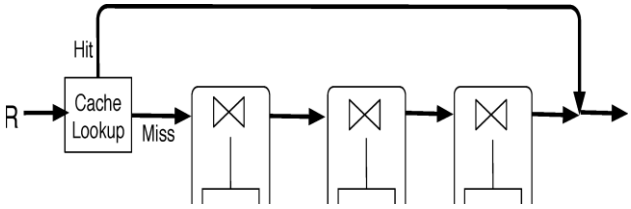

Figure - Adaptive Caching

It is interesting to note that Mjoins are not capable of using the previously computed matches and have to re-execute for every tuple.

6.3.1 Updating the Cache

Cache lookup has to be updated every time a tuple that has not been part of the earlier join is encountered. There are two ways to do this:

a. The tuples which are probed the first time can be inserted into the cache.

b. For data streams using sliding windows, the tuples can be constantly updated by maintaining fixed number of caches.

6.3.2 Choosing Caches Adaptively

A multiway join query can be executed using the following three phase approach:

a. A-greedy algorithm is used to make the decisions for adapting probing sequences based on operator selectivities.

b. The caches are selected adaptively given the probing sequence.

c. Memory allocation to the caches is also done adaptively.

7. Adaptive Join processing: History Dependent pipelined execution

Unlike the history independent pipelined execution, the states built up inside the operators is determined by the optimization or adaptation choices made by the query processor in a history dependent pipelined execution. The order of arrival of the tuples does not determine the operation to be performed on them.

7.1 Corrective Query processing

The CQP approach or Corrective Query processing approach uses a cost based optimization method instead of directly executing a query. In a CQP the decisions related to scheduling are separated from the decisions related to cost of computation. The also support a broader range of queries including aggregation and nested subqueries.

The working of the CQP approach is as follows:

The CQP engine chooses a plan initially and then begins executing it. The execution proceeds in different phases, which is similar to performing horizontal partitioning. Let us assume the CQP execution to consist of a join and aggregation operator with a stitch-up plan (which is indeed a combining process of all the intermediate results). First a join is performed over the data received through finite sequential streams and then the CQP engine prepares the estimates of cost and cardinality of the first plan. If a

better plan is found the processing of the first partition is completed and the optimized second plan is adopted for execution. If there are relations R, S and T which participate in CQP, then the first partition of data which executes with an initial plan can be represented as (R^0, S^0, T^0). When the new plan is re-estimated after the initial execution, the new phase or partition can be represented as (R^1, S^1, T^1). This process goes on till the stream data ends. A stitch up is performed on all the intermediate tuples to output the final results.

7.1.1 Analyzing the CQP approach using Adaptivity loop

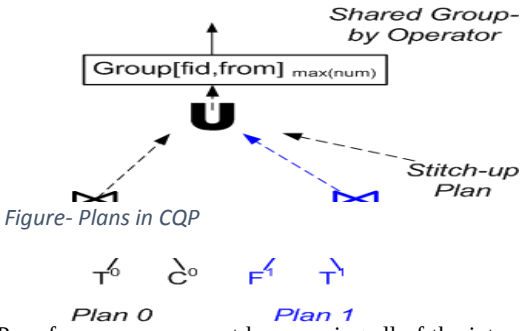

Figure- Plans in CQP

CQP performs measurement by exposing all of the internal states of the operators and it uses tuple order and unique value detectors to support specialized information gathering operators.

The analysis stage of the CQP differs from a conventional query processing in two ways:

a. The cost estimate is always calculated over the remainder of the source data and overhead for changing the plan since optimization is preferred for the further data available after the initial partition has already been executed.

b. The CQP approach models selectivities at a logical subexpression level rather than at the operator level thereby calculating the selectivity estimates which are independent of the evaluation plan.

The planning in CQP produces bushy plans that maximize pipelining and the query is executed as a union of all the per phase plans. The CQP query engine performs actuation using pipelined implementation of operators such as hash join, windowed sort and group by.

7.1.2 Separating scheduling from cost

Every query processing approach has two main cost computation issues, one related to scheduling tuple processing and the other related to operator ordering. The CQP approach separates these two issues unlike an eddy scheduling scheme because scheduling a less-desirable plan is not usually recommended. It is important to note that CQP uses pipelined hash joins to perform scheduling.

7.1.3 Analysis of CQP

CQP uses the feature of distributing operations such as selection, projection, joins and aggregations over unions to insert new partitions into the query plan. While selection and projection are quite trivial in their distribution over unions, joins and aggregations are the more complex ones.

Joins: A join expression over m relations divided into n partitions can be represented as:

$$R_1 \bowtie \cdots \bowtie R_m = \bigcup_{1 \le c_1 \le n, \ldots, 1 \le c_m \le n} (R_1^{c_1} \bowtie \cdots \bowtie R_m^{c_m}),$$

Where $R^c{}_j$ is some subset of the relation R_j.

Aggregations: CQP uses an adjustable window pre-aggregation operator to distribute operators such as min, max, min and count over the union. The window size of the pre-aggregation operator is adjustable as per the required conditions.

Some of the other operators that CQP can support are unions over unions, outerjoins and existential quantification.

Limitations: Though CQP supports many operators it does not support operators such as universal quantification, relational difference and NOT EXISTS predicate because they do not compute incrementally.

7.1.4 A generalization: Complementary Joins

An alternative plan proposed other than the horizontal data partitioning are the complementary joins which separately implements hash join and merge join dynamically. A router determines the path of a subsequent tuple from a data source. If a tuple that arrives is in chronological order with its predecessor then it is joined using merge sort and stored in the local hash table for R. A pipelined hash join is used if it is not in chronological order. Finally a mini stitch up is performed where the hash table from the pipelined hash join is combined with the hash table associated with the merge sort, where both of them belong to two different relations.

7.1.5 Open Problems related to CQP

Three of the important research avenues to open up from CQP are:

a. Though there might be a significant overhead, can the stitch up computations can be fit into the subsequent plan phase rather than a separate one?

b. Can the eddy exploration technique be beneficial to the CQP technique?

c. The existing technique does not consider the possibility of a fully distributed query plan.

7.2 Eddies with Binary operators

An eddy is an operator that routes through tuples through the operators using data and operator characteristics. For a three join relation query $R \bowtie_a S \bowtie_b T$, two symmetric hash join operators along with an eddy operator are instantiated. One valid route option

is that R tuples can only be routed to R⋈$_a$ S and T tuples can only be routed to S⋈$_b$ T.

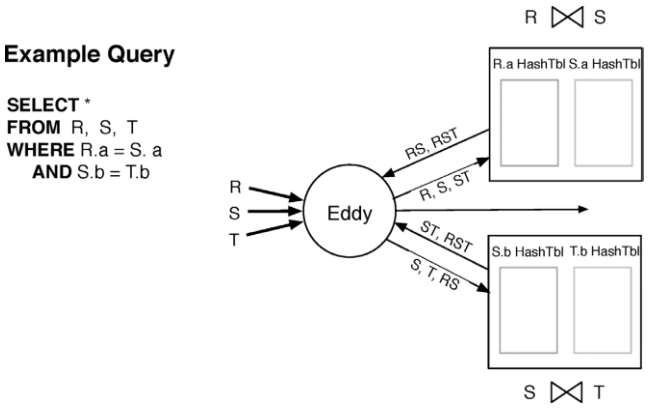

Figure - Eddies with Binary operators

In order to determine the validity of routing options, a lineage of the tuples can be maintained by using the bitmap table consisting of the *ready* and *done* bits. The operators are responsible for setting these bits to the tuples before they are returned to the eddy. The done bits indicates the operators that have been visited by the tuple and ready indicates a valid routing destination.

7.2.1 Routing Policies

The join states affect the output rates of the operators directly, the routing decisions are made by taking this state into account. However, the state accumulation leads to a challenge of designing routing policies for eddies.

There are two routing policies for this case:

a. Lottery scheduling: This policy is totally agnostic to the nature of the operators.

b. Deterministic routing with batching: It uses routing table and batching to reduce overheads. The eddy maintains the selectivities of the predicates, the domain sizes of the join attributes and the sizes of the relations. The routing decisions are made for batches of tuple at a time to reduce the overhead. A batching factor called K is used to invoke the reoptimizer.

An important note here is that the routing destinations for tuples alone determine the plans executed by the eddy and the order of arrival is not directly relevant.

Burden of Routing History:

Since the state gets accumulated inside the join operators, it severely limits the decisions that an eddy operator can make. This is known as the burden of routing history and it has the following impacts:

a. In the case of cyclic queries, an eddy might be unable to adapt spanning trees.

b. It is much harder to delete tuples from the execution state to make way for intermediate tuples because the state is stored in the operators.

c. In interactive environments, obtaining partial results is preferred but due to the burden of routing history it is not possible.

7.3 Eddies with STAIRS

STAIR is an operator that encapsulates the state stored inside a join operator. A STAIR on a relation R and attribute a can be denoted by ***R***.a which contains tuples from R and intermediate tuples that contain tuples from R. There are two operations that can be performed using a STAIR operator:

a. insert(***R***.a, t) : to store a tuple t inside a STAIR.

b. probe(***R***.a,val): for a given value val form the domain of attribute a of R, it outputs all the tuples of R where r.a = val.

Two STAIRs are called duals of each other when they are used instead of the join operators and interact with an eddy directly. The property of Dual Routing is defined as below:

"Whenever a tuple is routed to a STAIR for probing into it, it must be simultaneously inserted into the dual STAIR."[3]

An eddy inserts a tuple into one STAIR and probes its dual.

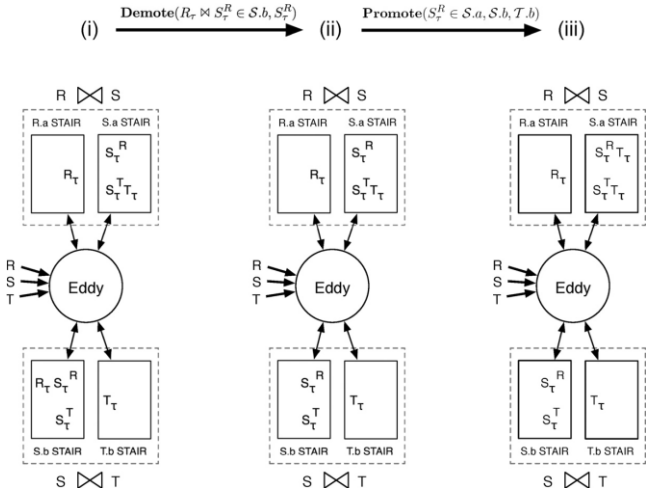

Figure - Eddies with STAIRs

7.3.1 State Management Primitives

There are two management primitives that allow the eddy to manipulate the join state inside the STAIRs:

a. Demote (t Є ***R***.a, t' = π_Y(t)), R Є Y where Y is a subset of schema(t): The process of removing some of the source tuples to reduce an intermediate tuple in the STAIR to a subtuple is known as demotion.

b. Promote (t Є ***R***.a, ***S***.b, ***T***.c), where S belongs to schema(t) and T does not belong to schema(t): The process of replacing a tuple t with t' in a STAIR ***R***.a where t' is a super tuple

consisting of an another join in the query which are found by probing into that relation.

7.3.2 Using State migration reduce burden of routing

State migration is the process of moving state from one STAIR to another. It can be used to reverse the routing decisions even though the query has executed for a certain period of time. Using the demote and promote primitives, if it is known that a sub-optimal query execution will happen by routing a tuple towards a certain subquery, the routing decision can be totally reversed.

In order to determine when to perform the state migration, a greedy policy exists which follows the routing policy by keeping the state inside the operator consistent with the plan that is currently being executed by the eddy. The migration of the state is done over a period of time in such a way that the internal join state and the eddy operator are in sync with respect to query plan.

7.4 Dynamic Plan Migration in CAPE

The CAPE query processor executes queries using a tree of binary operators and is not routing based. When a query is executing using the CAPE processor, different types of tuples are not at all permitted i.e., state migration has to be strictly performed when query plan is changed mid-execution.

State migration in CAPE can be done using the conventional approach or the parallel tracks strategy. The conventional approach uses the process of pausing the execution, migrating the state from old to new operators and then resuming execution using new query plan. The parallel tracks strategy is specifically used for data streams and it runs old and new plans simultaneously for a while before throwing away the old plans when its operators are no longer required in query execution. It is interesting to note that the state migration primitives supported by STAIRs can be used to perform migration in CAPE and vice versa.

8. Adaptive Pipeline processing- Non pipelined execution

Non pipelined operators are widely being used by most of the DBMS today. Here the query execution plan is divided into many pieces such that each piece is optimized and executed separately. Adaptation methods widely used here are plan staging, mid-query re-optimization and query scrambling.

8.1 Plan staging

In plan staging intermediate result relations are created entirely before processing the next leftover operators in the query execution plan. These points where intermediate result relations are created are called materialization points. They are used in sorting in sort-merge join, group-by, for building hash table in hash join and in many other places. They will divide the relational algebra tree into subgraphs where output of one subgraph are accessible as inputs for the next. Here first optimization is done and runs a single stage to completion and then using the results obtained optimization and execution of next stage happens and so on. At every step we can use the statistics obtained at previous steps.

8.2 Mid-Query Reoptimization

Mid-Query reoptimization has three steps.

a. First checkpoints are constructed in the query plans.

b. Then if the cardinality (uniqueness of tuples) through checkpoint is different from estimated value then checkpoint terminates current execution of the query and calls for another round of optimization and execution.

c. Repeat the step until end of relation.

Checkpoint will measure uniqueness of tuples flowing through it. Checkpoints are categorized according to the place where are in the relational algebra tree. Checkpoints placed just above materialization points are called lazy checkpoints. The drawback of lazy checkpoint is that it is infrequent because of which there may be huge performance losses. The alternative to lazy checkpoint is eager checkpoint where checking of cardinality is done in pipelined fashion. As a result they can be place anywhere in the relational algebra tree. But the problem with eager checkpoint is that it results in waste of work during reoptimization as the tuples which are that are processed until reoptimization have to be thrown away. However they are widely useful in the situations where query optimizer estimates lot of uncertainty. To avoid the wasted work the optimizer should try to reuse the intermediate results which are generated during query execution till the checkpoint. It is better to have intermediate results generated as the materialized views so that the query optimizer can decide how to use them in the best way at the time of reoptimization. The issue with the reoptimization is that it always incurs some costs. At least there will be cost of re-running the optimizer and restarting the query execution. Most of the times intermediate results are not reused perfectly after reoptimization. So, it is better to have reoptimization only if we are reasonably sure that a much faster plan will result in. It basically depends on the difference of estimated and actual cardinalities of checkpoint. If the difference is huge then it is likely that the current plan is not at all optimal and reoptimization has to be done.

8.3 Query Scrambling

The alternative to plan staging when there is wide range of data sources or when there is streaming data. Plan staging works perfectly when whole data is available before starting of query execution but when there are delays in arrival of data then plan staging will not be efficient and then query scrambling comes into picture. It is a method which reacts to huge delays in arrival of data by rescheduling the order of operators in query execution plan and occasionally changing the query plan itself. When there is delay in arrival of data easiest solution is that we don't change the query execution plan but change the order of operators. First the operators for which data is readily available are executed and so on. Sometimes no operators of the current plan can be scheduled right away due to delays. In such conditions it introduces and executes

new operators which are not initially included in the plan and are used to join tables which are not joined in the original plan. Later when the data arrives it calls the optimizer to construct a single query plan that best uses all the joins.

9. Challenges

a. Many recent adaptive techniques like Symmetric Hash Join, Eddies, M-Joins etc. have completely changed the way of query processing and optimization such that different optimization plans are used for different tuples and runtime decisions are being made. Moreover very less amount of work is done on Adaptive query Processing and research is in the initial stages. As a result we cannot know exactly how these complex operators affect adaptive query space.

b. Most of the techniques used like Eddies, M-Joins etc. are concentrated only in actuate in which runtime decisions are executed by doing a lot of work in changing system state. But there is very less concentration on monitor, plan and analyze steps. Adequate amount of research work should be done in these steps.

c. Correlation is one of the major reasons for estimation errors in cardinality which cannot be easily solved by the adaptation techniques. Firstly we consider a join order and later when we realize that number of tuples returned is very high than expected. The query execution plan may change where there will be join of other two relations which are still not explored by observing the correlation between them. This, sometimes is a very good technique as we can have a chance of focusing on exploring unknown relations, however sometimes it may result in continuous succession of bad plans.

d. Most of the processing is done in the main memory. Some adaptation mechanisms (some variants of symmetric hash join and corrective query processing) have higher work load than available memory. In that case there will be huge impact of memory overflow on performance. Lot of exploration has to be done in this aspect.

e. Even if a query execution plan speeds up most of the queries and slows down minute percent of queries, we may not have the best performance. The goal of adaptive query processing should be in a way such that it should dynamically support the user needs and query refinement. A lot of work needs to be done in this perspective.

10. References

[1]www.cis.upenn.edu/~zives/research/aqp-tutorial-description.pdf

[2]www.cs.umd.edu/~amol/talks/VLDB07-AQP-Tutorial.pdf

[3]www.cis.upenn.edu/~zives/research/aqp-survey.pdf

TEAM ROLES AND CONTRIBUTION: Every person contributed equally to the project.