Progressive Optimization in a Shared-Nothing Parallel Database

Wook-Shin Han[*]
Kyungpook National University

Jack Ng
IBM Toronto Lab

Volker Markl
IBM Almaden Research Center

Holger Kache
IBM Silicon Valley Lab

Mokhtar Kandil
IBM Toronto Lab

ABSTRACT

Commercial enterprise data warehouses are typically implemented on parallel databases due to the inherent scalability and performance limitation of a serial architecture. Queries used in such large data warehouses can contain complex predicates as well as multiple joins, and the resulting query execution plans generated by the optimizer may be sub-optimal due to mis-estimates of row cardinalities. Progressive optimization (POP) is an approach to detect cardinality estimation errors by monitoring actual cardinalities at run-time and to recover by triggering re-optimization with the actual cardinalities measured. However, the original *serial* POP solution is based on a serial processing architecture, and the core ideas cannot be readily applied to a parallel shared-nothing environment. Extending the serial POP to a parallel environment is a challenging problem since we need to determine when and how we can trigger re-optimization based on cardinalities collected from multiple independent nodes. In this paper, we present a comprehensive and practical solution to this problem, including several novel voting schemes whether to trigger re-optimization, a mechanism to reuse local intermediate results across nodes as a partitioned materialized view, several flavors of parallel checkpoint operators, and parallel checkpoint processing methods using efficient communication protocols. This solution has been prototyped in a leading commercial parallel DBMS. We have performed extensive experiments using the TPC-H benchmark and a real-world database. Experimental results show that our solution has negligible runtime overhead and accelerates the performance of complex OLAP queries by up to a factor of 22.

Categories and Subject Descriptors: H.2 [DATABASE MANAGEMENT]: Parallel databases

General Terms: Algorithms, Performance

[*]Work carried out while the author was visiting the IBM Almaden Research Center. Wook-Shin Han was supported by the Korea Research Foundation Grant (KRF-2005-C00173).

Keywords: Query optimization, OLAP, Parallel databases, Autonomous computing

1. INTRODUCTION

Today's enterprises often rely on timely and insightful information for strategic business decisions. Ad hoc on-line analytical processing (OLAP) queries are formulated to analyze and identify patterns in data sets, thereby transforming data into useful and consumable information. Due to the large quantities of data needed to be analyzed (or "mined"), OLAP systems are generally backed by a large data warehouse implemented on a sophisticated *parallel* database management system (DBMS) [3, 21, 22]. Furthermore, OLAP typically requires complex relational operations on large data sets and, as such, query performance is a critical factor.

While commercial query optimizers can estimate quite accurately the intermediate cardinalities for most queries, factors such as outdated statistics and invalid assumptions on column independence may lead to significant errors in the estimation, resulting in a sub-optimal query execution plan (QEP or plan). This is particularly troublesome for complex, long-running OLAP queries where a suboptimal plan can cause performance to degrade dramatically, possibly running for hours instead of seconds. However, despite such an importance, commercial parallel DBMSs currently do not have a practical solution to tackle poor OLAP performance caused by suboptimal access plans. Enhancements in this area can therefore bring tremendous benefits to data warehousing and business intelligence applications.

Recently, progressive optimization (POP) [15] was proposed as a robust and innovative technique to address this problem. During query execution, POP attempts to detect sub-optimality due to optimization errors in an access plan and generates a better plan to continue the execution. Multiple iterations of re-optimization can be triggered to progressively refine the plan. Intermediate results from the partially executed query can also be reused, thus avoiding the need to start the execution from the very beginning every time re-optimization is triggered. This property allows POP to have a negligible runtime overhead cost and makes it a feasible solution to remedy poorly performing queries.

While the original POP solution (also called serial POP) describes methods and a practical system of mid-query re-optimization, its design is based on a serial processing architecture, and the core ideas cannot be readily applied to a parallel environment. Commercial enterprise data warehouses, however, are typically implemented on parallel databases due to the inherent scalability and performance limitations of a serial architecture. Thus, although serial POP is a robust technology, it is not a complete solution which can be readily applied to real-world data warehouses.

Given that POP is conceived to benefit complex OLAP queries of large data warehouses, adding support for parallel DBMSs is an important requirement for the technology to become a fully practical solution in the real world. This paper describes a comprehensive solution that integrates POP seamlessly in a parallel database architecture. To the best of our knowledge, this is the first and foremost solution to address the said problem by combining mid-query re-optimization techniques with parallelism exploitation. It should be noted that although our solution uses the concepts of serial POP as a basis, integrating the technology in a parallel environment introduces a whole new set of technical problems and challenges, and the innovative techniques used to solve them represent significant novelty in itself.

Our solution allows POP to operate seamlessly within a shared-nothing parallel database, where queries are compiled at a central node (called the coordinator node) which distributes the processing requests to all participating subordinator nodes and collects their results. At the coordinator site, the optimizer calculates validity ranges of the parallel plan operators of different candidate access plans and chooses the one with the lowest estimated cost to drive the execution. After a plan has been selected, the optimizer strategically inserts parallel checkpoint operators (PCHECK) in the plan. PCHECK marks the place where POP can validate the optimality of the chosen plan and possibly intervene in the execution. The coordinator distributes the augmented plan to subordinator sites for parallel execution. During query compilation, the solution also creates a virtual partitioned materialized view (VPMV) at the coordinator site for each checkpoint operator. At each node, upon completion of the plan operator below PCHECK, the subordinator sends status (cardinality, identifiers of the results completed up to PCHECK) to the coordinator in a non-blocking mode. The coordinator receives status messages about the completed results and records them in a VPMV with the validity range of that result and decides whether to POP based on a novel voting scheme. In reaching the decision to re-optimize, the coordinator feeds back VPMVs to the optimizer so that re-optimization can be triggered with the actual cardinalities measured and distributed intermediate results reused. Upon successful completion of the query, the coordinator invokes distributed garbage collection of intermediate results at all subordinator sites.

The main contributions of this paper are as follows: 1) we introduce the PCHECK operator and propose novel placement strategies of the PCHECK operators for parallel query execution plans; 2) we propose several variations of the PCHECK operator and their processing methods using efficient communication protocols; 3) we propose novel voting schemes to make a global decision of whether or not re-optimization should be triggered; 4) we propose a mechanism that re-uses a distributed temporary table in a parallel architecture; 5) we have performed extensive experiments with the TPC-H benchmark and a real-world database, and shown that our solution has negligible runtime overhead and accelerates the performance of complex OLAP queries by up to a factor of 22.

The rest of the paper is organized as follows. We give some background information on the shared-nothing database and serial POP in Section 2. Section 3 gives an overview of our approach. We present several PCHECK placement strategies in Section 4. Section 5 describes PCHECK processing mechanisms. We present several novel voting schemes in Section 6. We explain how to efficiently re-use distributed temporary results in a parallel architecture in Section 7. We present experimental results in Section 8. We review related work in Section 9 and give our conclusion in Section 10.

2. BACKGROUND

In this section, we first briefly describe the shared-nothing parallel database architecture. We then give an overview of the serial POP.

2.1 Shared-Nothing Parallel Databases

Commercial parallel relational DBMSs such as NCR Teradata [21], IBM DB2 MPP [3], Tandem NonStop SQL [22] are based on the shared-nothing architecture. Relational tables are partitioned across a collection of nodes. Here, some hash-based partitioning function is used to distribute rows.

Shared-nothing parallel DBMSs typically employ a query processing model whereby a central site serves as the query coordinator. When a user issues an SQL statement, the coordinator invokes the optimizer to generate a suitable QEP and distributes it to subordinator sites for parallel execution. Each site (also commonly called a node or partition), receives and runs the QEP against a subset of the table data and the results are sent back to the coordinator, which collects the rows from all subordinator sites and returns the final results to the user.

When joining two tables in a parallel database environment, the table data must be physically located in the same database partition where the join operation takes place. If not, the data must be transferred from one partition to the designated joining partition. This movement of data is accomplished by means of a table queue, which is a logical data pipe for transferring table rows among database partitions. Because relocating data is expensive and can have a large impact on performance (especially for large databases), the query optimizer employs several parallel join strategies to minimize such data movement when resolving relational table joins.

With the *collocated join* strategy, each node executes the join between two table partitions locally. Collocated join is chosen only if both tables participating in the join are partitioned on the join keys. With the *directed join* strategy, rows in the outer (or inner) table of the join are directed via table queues to one or more target database partitions based on the inner (or outer) table's partitioning attributes of the joining columns. Once the rows are relocated to the relevant database partitions, the join operation between the two tables occurs in parallel on these partitions. With the *repartitioned join* strategy, rows of both the inner and outer tables of the join are flown to a set of partitions based on the partitioning attributes of the joining columns. Note that unlike the pure directed table join strategy, the partitioning columns of both tables are different from the join columns. Hence, this strategy essentially performs a repartitioning operation on the joining tables. With the *broadcast join* strategy, each row in the outer (or inner) table is sent to all database partitions where the inner (or outer) table exists, regardless of inner (or outer) table's partitioning attributes of the joining columns. Broadcast join is usually chosen by the optimizer if other parallel join strategies cannot be used (e.g., no equality join predicates between the joining tables), but it can also be used in other situations where it is determined to be the most cost-effective join method (e.g., joining a very small table with a large one).

2.2 Serial POP

Serial POP attempts to detect suboptimal performance and switch to a better plan to continue the execution, without having to re-run the entire query from the beginning. Multiple rounds of re-optimization can be invoked until the optimal plan is used. Moreover, through a sophisticated matching scheme, intermediate results from previous rounds of execution are made available for reuse, further reducing the cost of re-optimization.

Figure 1 (from [20]) shows the architecture of the serial POP solution using a checkpoint runtime operator (CHECK) to validate the optimality of the executing plan. CHECK compares the actual cardinality at that point against a computed validity range that consists of a lower bound and an upper bound for the cardinality. If the actual cardinality is not within the bounds, query execution is aborted and re-optimization is triggered. Furthermore, additional knowledge and statistics from the current execution are fed back to the optimizer for use in generating a better plan. To minimize the overhead of re-optimization, intermediate/partial results (in the form of materialized views) from previous rounds of executions are reused whenever possible through a sophisticated matching process. CHECKs are inserted in a QEP using a placement strategy during the query compilation phase.

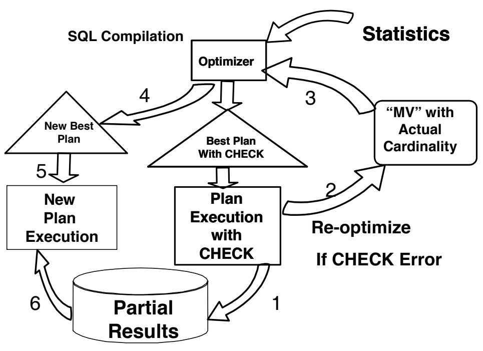


Figure 1: Architecture of the serial POP solution.

3. OVERVIEW OF OUR SOLUTION

Our solution allows POP to operate seamlessly within a shared-nothing parallel database. At the coordinator site (where the query is issued), the optimizer exploits the validity range calculation method used in serial POP for different candidate parallel access plans and chooses the one with the lowest estimated cost to drive the execution. After a plan has been selected, the optimizer strategically inserts dam operators (TEMP) and PCHECKs in the plan. The TEMP operator represents a temporary table materialization point and the checkpoint operator marks the place where POP can validate the optimality of the chosen plan and possibly intervene in the execution.

For the parallel POP solution, the PCHECK placement strategy takes into account the different parallel relational join strategies used by the optimizer. Details of this novel placement strategy are discussed in a later section. The coordinator then translates the augmented access plan to a low-level QEP and distributes it to subordinator sites for parallel execution. During query compilation, the solution also creates a VPMV structure at the coordinator site for each PCHECK. The structure contains the matching query graph information as well as other runtime information such as cardinalities relevant for intermediate results matching. Because the intermediate results, implemented as temporary tables, are distributed across different processing sites, our solution uses VPMV structures to "virtualize" these distributed results for the optimizer to generate view matching structures needed for reuse. The method for reusing distributed temporary results is elaborated in Section 7.

A subordinator partition receives and processes the QEP. When a checkpoint operator is encountered, the subordinator generates a POP *vote* using a POP-VOTE message and sends it back to the coordinator. The POP vote consists of statistics such as table cardinality, as well as information that is relevant for synchronizing the flow of re-optimization and reusing intermediate results in a parallel environment. In addition, the subordinator performs a local validity range check to determine whether it should wait for the coordinator's global POP decision or continue its query execution.

When a POP vote arrives, the coordinator will be interrupted from its normal processing to handle the vote. It extracts the information and uses a defined voting scheme to make a global decision to whether or not to trigger re-optimization and to abort the current execution. Further, for each POP vote, the coordinator finds the corresponding virtual VPMV structure to record its relevant runtime information.

Assuming re-optimization is triggered, the coordinator distributes POP-COMMIT messages to halt the query execution at subordinator sites and waits for their acknowledgements. Once all sites have aborted and sent back their replies, the coordinator sends a POP-CLOSE message to each subordinator site to perform the necessary cleanup and be ready for the next iteration of query execution. After synchronizing with the subordinator sites to perform cleanup, the coordinator examines what intermediate results can be reused. At this point, the coordinator re-invokes the optimizer logic and uses the feedback statistics to generate a better access plan to run the query.

The steps of collecting votes, making a global decision, and triggering re-optimization can be repeated multiple times until the optimal QEP is found. At the end of query execution, the coordinator distributes a POP-COLLECT-GARBAGE message to invoke distributed garbage collection of cached intermediate results at processing sites.

4. CHECKPOINT OPERATOR PLACEMENT STRATEGIES FOR PARALLEL QUERY EXECUTION PLANS

The main metrics concerning the placement of PCHECK operators are the risk and opportunity of re-optimization. In contrast to a serial query plan, a parallel plan can have table queue operators (TQs) for transferring table data among database partitions in relational joins as discussed in Section 2.1. Since joins in a parallel database highly depend on the existence and type of TQ, we introduce three kinds of PCHECKs to ensure that the re-optimization risks and opportunities are balanced in a parallel environment where data relocation may occur due to joins.[1] Our goal is to place these checkpoints as early as possible to avoid expensive data movement when re-optimization is necessary, while at the same time we need to take into consideration that the cardinality can be changed after executing a TQ operator, which in turn can affect the accuracy of the re-optimization decision.

The parallel checkpoint placement logic described below is executed at the coordinator site when the optimizer is invoked during the initial round of query execution as well as at re-optimization time. The following three types of parallel checkpoints are proposed.

Broadcast Join PCHECK

The broadcast join PCHECK operator targets the broadcast join strategy. When there is a broadcast join operator (BJ) in a query plan, only one child of the BJ (i.e., the inner or outer leg of the join) would have a broadcast table queue (BTQ) operator. Because the cardinality of the temporary inner or outer table for the join will not change after executing a BTQ, our solution strategically places a PCHECK *before* the BTQ operator. Furthermore, if no dam operator (i.e., a table materialization point such as sorting and temping) already exists before the BTQ operator, we add a TEMP operator below the PCHECK. Placing a materialization point before the BTQ operator ensures that the system can reliably determine if the cardinality of BTQ would violate the defined validity range; thus, when the cardinality is out of range, the expensive data broadcast operation can be avoided and re-optimization can be triggered earlier.

Usually a broadcast join is selected if the cardinality of the BTQ is small and the cost of broadcasting the rows is relatively inexpensive. Thus, although we introduce a dam operator before the BTQ operator if none already exists in the plan, the cost of materializing the dam is mostly negligible due to the following rationale: if the validity range of a PCHECK is not violated, it often means the table size is small (at least within the optimizer's estimation) and so the cost of materialization is also small; if the validity range is violated, the cost of executing the dam will be compensated by re-optimization avoiding disasters (the intermediate results can also be reused).

Figure 2(a) below shows a template of the broadcast PCHECK plan. The broadcast join operator has one child (on the left side) that broadcasts all rows across all the partitions where the join is executed. In the diagram, the PCHECK operator is placed before the BTQ and after the dam operator. The dam operator can be naturally introduced (e.g., in an ordered nested loop join) or artificially inserted by our solution for having a materialization point for the PCHECK operator as discussed.

Directed Join PCHECK

This directed PCHECK operator targets the directed join strategy. When there is a directed join operator (DJ) in a query plan, only one child of the DJ would have a directed table queue (DTQ) operator. Since the cardinality of the temporary inner or outer table for the join could change on some partitions after executing a DTQ (i.e., the ones receiving the table queue data), we place a PCHECK *after* the DTQ operator. Even though the PCHECK can also be placed before the DTQ as in the case of the broadcast join case, we intend to use a lazy checking strategy to avoid adding too much risk in POP due to inaccurate cardinality values after the table queue operation. However, if no dam operator exists before the DTQ, we add a TEMP operator below the PCHECK to introduce an artificial materialization point, just as in the case of broadcast PCHECK. We apply thresholding logic on the cardinality of the DTQ operator such that a dam is not placed if the estimated cardinality is beyond a certain threshold. Figure 2(b) shows a template of the Directed PCHECK plan. Based on the partitioning property of the joining columns, the DJ operator has one child (on the left side) which directs the rows of the outer table to a set of partitions where the inner table is located. The dam operator is placed on top of the DTQ operator and the PCHECK operator is in turn placed on top of the artificial dam.

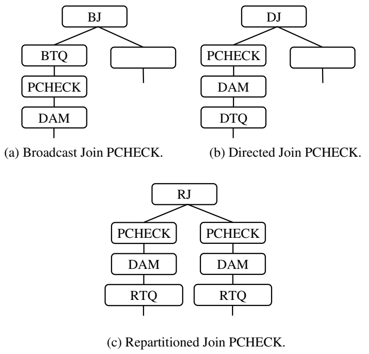

BJ
BTQ
PCHECK
DAM
(a) Broadcast Join PCHECK.
DJ
PCHECK
DAM
DTQ
(b) Directed Join PCHECK.
RJ
PCHECK
DAM
RTQ
PCHECK
DAM
RTQ
(c) Repartitioned Join PCHECK.

Figure 2: PCHECK placement strategies.

Repartitioned Join PCHECK

The repartitioned PCHECK operator targets the repartitioned join strategy. When there is a repartitioned join operator (RJ) in a query plan, both children of the RJ operator have a repartitioned table queue (RTQ) operators with which rows of both the inner and outer tables of the join are transferred to a set of partitions based on the joining columns' partitioning properties. The placement strategy used for this type of TQ also inserts the PCHECK operator *after* the RTQ operator. As with directed PCHECK, we intend to avoid introducing excess risk to re-optimization. Also, if there is no natural materialization point after the RTQ operator, a TEMP operator is added below the PCHECK as an artificial dam.

As Figure 2(c) illustrates, both children of the repartitioned join in the query plan have a TQ operator, with which table data is sent to different database partitions based on the partitioning information of the joining columns. The TEMP operator is placed on top of the RTQ operator and the PCHECK operator is placed on top of the TEMP.

5. PARALLEL CHECKPOINT OPERATOR PROCESSING

One fundamental property of the parallel database architecture is the ability to carry out tasks at different process-

[1]Since collocated joins have no TQs, we use the serial CHECK placement strategy to place PCHECK operators.

ing sites concurrently and independently to achieve superior runtime throughput. The natural synchronization points between the coordinator and its subordinators in a parallel query occur in the path of receiving and aggregating row data from subordinate sites. When a subordinate site receives a QEP from the coordinator, it performs the query execution orthogonal to other subordinators and therefore a slower site does not affect the progress of a faster one. Once a subordinator finishes its work, it can immediately perform tasks for other queries, making very efficient use of the available computing resources. Thus, the runtime design of the parallel POP solution needs to adhere to this characteristic and avoid introducing new synchronization points in its communication flow. Obviously, adding new synchronizations points can also negatively impact the response time of a query execution. This is crucial for the practicality of POP since the technology is designed to exist as a general feature in a database engine, which means it must have minimal performance impact on queries that do not need re-optimization (such as queries issued against a database with perfect statistics and short-running queries in an online transaction processing (OLTP) workload).

Hence, on one hand the throughput capacity and response time of a parallel DBMS must be preserved, but on the other hand it is obvious that tasks performed for a query would become wasted computations whenever it is aborted due to re-optimiation but the intermediate results cannot be reused. This means the parallel checkpoint operator processing design needs to strike a balance between these two competing design factors.

We propose three types of parallel checkpoint methods. 1) In the *synchronous checkpoint method* each subordinator waits for the coordinator's decision after sending its vote. The advantages of this method are that it is easy to implement and that it does not waste unnecessary computing resources if re-optimization is later needed by always waiting for the coordinator's global decision before proceeding. The main drawback of this method is that slow subordinators hold up faster ones, resulting in performance degradation for ordinary queries. 2) The second processing type is the *asynchronous checkpoint method*, with which each subordinator continues execution after sending its vote to the coordinator and will be interrupted asynchronously if the coordinator later makes a decision to re-optimize. This method does not slow down query processing for ordinary queries, but has the disadvantage of potentially wasting computing resources on faster subordinators if re-optimization is triggered after tasks have been performed beyond the checkpoint. 3) The *hybrid checkpoint method* combines the *synchronous* and the *asynchronous* checkpoint methods to balance between query performance and efficient utilization of computing resources. The heuristic used is that if the local cardinality is within the validity range, then the method behaves like the asynchronous checkpoint method. Otherwise, it acts just like the synchronous checkpoint method. That is, it blocks and waits for the coordinator's decision since there is a good chance that POP is globally needed.

With the hybrid checkpoint algorithm, typically no new synchronization points would be added to the communication and voting flow of parallel POP for queries that do not need to re-optimize. Moreover, the slowest subordinate partition would not become the bottleneck of the execution preventing faster partitions from making progress (i.e., forcing them to wait for the coordinator's global decision at a PCHECK operator).

As POP votes arrive, the coordinator gets interrupted to collect the votes at the closest interrupt detection point (a spot in the database engine designated to asynchronously detect the arrival of messages from subordinate partitions). It extracts information from the POP votes (based on the PCHECK operator number) and updates the matching VPMV structure (see Section 7 for details on these structures). In addition, based on the voting scheme, the coordinator makes a global decision whether or not to trigger re-optimization.

If re-optimization is needed, the coordinator distributes a POP-COMMIT message to all relevant subordinate partitions and interrupts their query execution. The coordinator will synchronize with all the subordinators before actually going back into the optimizer to invoke re-optimization. This is to ensure cleanup is done properly and the new query execution does not inadvertently collide with the current execution at the target partitions. Also, in the POP-COMMIT message the coordinator indicates which local temporary tables should be kept for possible reuse in future rounds of execution. The coordinator can drop a temporary table if it is subsumed by another existing table or if the system is running low on storage/memory resources.

Depending on the stage of its query execution (e.g., blocking on a checkpoint or actively processing the plan operators), either the subordinator can receive the POP-COMMIT message at the checkpoint or it must poll for the arrival of the message at the closest interrupt detection point. In any case, the subordinator will get interrupted and pick up the POP-COMMIT message, after which it will abort its execution, perform all necessary cleanup, and send back a reply to acknowledge the coordinator's decision. As part of the cleanup, the subordinator will cache the completely materialized temporary tables for potential reuse and drop all partially materialized results since the optimizer will not consider them for reuse. Further, for the local temporary tables that are saved, the subordinator stores the runtime information about those table objects for future reuse and the eventual garbage collection.

A subordinator also deletes local temporary tables that the coordinator did not request to keep. After the coordinator has received all the POP-COMMIT acknowledgment replies, it will send out a POP-CLOSE message to perform the final cleanup to close the QEP at subordinate partitions before a new round of query execution can take place. After this is done, the coordinator invokes the re-optimization logic in the optimizer to generate a new plan.

If re-optimization is not needed, the coordinator still examines the received POP votes to determine if any subordinate partition is waiting for its decision message. If no partition is waiting, the coordinator simply continues its query execution. But if one or more waiting partitions are found, the coordinator will send them a POP-CONTINUE message to unblock them so that they can continue their execution.

Note that because checkpoint processing can happen concurrently and asynchronously, it could be the case that a fast subordinate partition has already flown rows back to the coordinator by the time votes from other partitions return to the coordinating partition and the coordinator decide to trigger re-optimization. Since this has the potential danger of leading to duplicate rows being returned back to the user, in our current prototype the coordinator does not trigger re-optimization in this scenario. Alternatively, the proposed system can buffer up the received rows and not return them to the user prematurely until all votes for a particular checkpoint operator have been received, or the system can save relevant state information to remember what rows have been returned to the user and not return duplicates.

6. THE VOTING MECHANISM

One key component of the parallel POP solution is the voting mechanism. In a parallel database system, a query plan is distributed to multiple sites for execution and each site will process a subset of the table data. Since data is not always partitioned uniformly in a parallel environment, the cardinalities for one CHECK operator are not necessarily similar across different processing sites. As such, taking the outcome of a local validity range check at one site alone is not sufficient for generating the re-optimization decision. Rather, the decision needs to be reached collectively at the global level based on the input from each processing partition. A robust voting mechanism is therefore necessitated.

We propose a voting mechanism that consists of a reliable, asynchronous communication facility and a set of voting schemes for handling re-optimization decisions. The communication facility is used for exchanging votes and control messages needed to synchronize the voting process. The voting schemes are used for analyzing the collected votes and deriving a global re-optimization decision.

Each execution of the PCHECK operator generates a POP vote. Since a particular PCHECK operator can be executed concurrently and asynchronously at more than one location, the mechanism uses an internal table to keep track of the number of votes that have been gathered for each PCHECK operator. Depending on the voting scheme being used, the mechanism either must wait for all votes for the PCHECK operator to be collected first, or it can generate a decision after a number of votes for an operator have been received.

When a POP vote is gathered at the coordinator partition, the voting mechanism first identifies the sender of the vote and extracts the information that indicates whether the sender is waiting for a decision message. Then, the mechanism extracts the voting criteria, which is the information used to translate the vote into a *pro* or *con* position for re-optimization. Our solution currently uses table cardinality as the voting criteria. The extracted cardinality value is compared against the validity range defined for the corresponding PCHECK operator. If it is not within the range, it is counted as a vote for re-optimization; otherwise, it is counted as a vote against re-optimization. If necessary, the mechanism can also use other statistics or information as the voting criteria.

The following five voting schemes are defined:

- Unanimous voting scheme: all participants need to vote for re-optimization before the coordinator can trigger the process. This is the most basic and simple form of voting and serves mainly as a base line comparison for other voting schemes.

- Majority voting scheme: quorums are created as POP votes arrive at the coordinator partition. As soon as a quorum containing at least one-half of the total participants voting to re-optimize is formed, the coordinator will trigger re-optimization. We have found empirically that this is relatively the most versatile voting scheme and works well with both uniform and skewed data partitionings.

- Averaging voting scheme: all votes for a particular checkpoint operator must arrive before a global POP decision can be made. Once all votes have been collected, the coordinator will compute the average cardinality and trigger re-optimization if the value violates the validity range defined for that PCHECK.

- Maximum voting scheme: this is also called the veto scheme, since the coordinator will trigger a global re-optimization decision as soon as the actual cardinality of one arriving vote violates the validity range defined for the particular checkpoint operator, even if the majority is within the bounds. This voting scheme is designed to work with databases that have highly skewed data partitionings and the re-optimization decision is biased towards the partition with the largest number of rows for the partitioned table.

- Weighted voting scheme: this is a meta-scheme and can be applied to the majority, averaging, and maximum voting schemes. With the weighted voting scheme, each vote collected by the coordinator is weight-adjusted based on cardinality estimation errors or cost factors such as hardware capabilities. This voting scheme targets parallel databases that have skewed data partitionings and/or are implemented on a heterogeneous set of machines with dissimilar hardware capabilities.

Related to voting schemes is the concept of voting strategies. The voting mechanism can use either an *eager* or a *lazy* strategy. With eager voting, the decision message is distributed to subordinate partitions as soon as a global decision has been reached without waiting for the arrival of the remaining votes. For example, assuming the majority scheme is used, if three partitions are participating in the query execution and two subordinate partitions have sent back their votes to vote against re-optimization, then the coordinator will immediately send out a POP-CONTINUE message to resume the execution. In contrast, with lazy voting, the coordinator waits for all votes to arrive before distributing the decision message, even if a global decision has already been reached. We use a *combination* of the two strategies to ensure the voting mechanism and the overall re-optimization process operates optimally. In general, if the global decision is to continue with the current execution, the coordinator uses the eager strategy to communicate the decision to all partitions such that any subordinator blocking on the checkpoint operator can resume the execution as early as possible. On the other hand, if the global decision is to abort the current execution and trigger re-optimization, the coordinator switches to the lazy strategy and waits for all votes for that PCHECK to come in before sending out the POP-COMMIT message. This ensures that all the local temporary tables below that checkpoint have been fully materialized at all sites and can be reused during re-optimization.

Note that the voting mechanism is flexible and can easily incorporate new schemes and strategies based on future research. Furthermore, it has the capability to use a different scheme dynamically and automatically to adapt to various types of queries and data partitionings. Combined with the different modes of checkpoint processing, the voting mechanism provides a powerful and adaptive framework for future work.

7. RE-USING DISTRIBUTED TEMPORARY TABLE RESULTS

Typically for a parallel DBMS with the shared-nothing architecture, a physical base table (e.g., one that is created by the user) uses the same table identifier (TID) on all partitions where it is distributedly located. So it is a straightforward task for the optimizer to encode and identify a base table in a query execution plan since the same TID can simply be used by all partitions to reference the target table locally.

However, a temporary table (e.g., one that is created and used only for the duration of the query execution) has in general a different TID at each partition where the table is materialized. The optimizer has no knowledge about these temporary tables and the local TIDs they use, and as such it cannot assign a globally unique identifier to them. Thus, it is non-trivial to reuse POP temporary tables in a parallel environment. In particular, the method described in serial POP of using TIDs directly to identify and reuse intermediate results is not applicable in the parallel architecture. The problem of reusing intermediate results in a parallel environment is complicated by the fact that a temporary table may not have been materialized at all relevant partitions by the time re-optimization is triggered. Therefore, to reduce the overhead of parallel POP, we propose a method to exploit a partially materialized temporary table and its statistical information such that useful knowledge is fed back to the optimizer for re-optimization even when the table has not been materialized at all needed locations.

In order to reuse local temporary tables for re-optimization, the solution treats them as partitioned materialized views so that they can be fed into the optimizer when generating a new plan. For this purpose, VPMV structures are created and maintained at the coordinator site. At query compilation time, three pieces of information are extracted from the selected query plan and stored in the VPMVs for the purpose of results reuse: 1) matching query graph information; 2) ordering information; 3) partitioning information. The information is easily extracted from the properties of each dam operator in the plan. At query execution time, local temporary tables are cached at each partition when the dam operator is executed to completion. Then as the PCHECK operators are executed at each partition, TIDs and the cardinalities of the local temporary tables are flown back via a POP vote to the coordinator partition for storing in the matching VPMV. Note that the local TID is (and must be) unique at each partition, although uniqueness across partitions is neither guaranteed nor required due to the virtue of the VPMV structures.

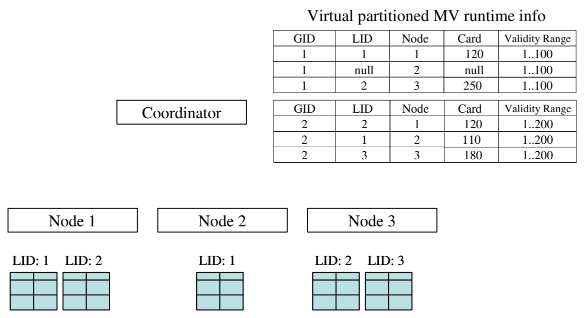
A two-level identification scheme is devised to achieve the reuse of distributed temporary tables at re-optimization time. Specifically, to allow the optimizer to identify and reuse a particular "virtual" temporary table, each PCHECK will have a corresponding VPMV structure created and a unique global identifier (GID) assigned to it at the coordinator partition; and for each unique VPMV, a local identifier (LID) is stored for each partition currently executing the query plan. This information, along with other statistics such as the cardinality, is sent back to the coordinator via a POP vote when a partition executes the PCHECK operator. Before the SQL compiler is called for re-optimization at the coordinator partition, the vote processing logic converts the intermediate results information captured in the POP votes into a partitioned materialized view by using the VPMVs created thus far. This ensures that the exact cardinalities for the distributed local temporary tables are available in the materialized view and can be used by the optimizer. Note that since several PCHECK operators can exist in a query plan, in-memory data structures are maintained to allow efficient access to the target VPMV for a given PCHECK operator number.

Because query execution at different partitions occurs concurrently in a parallel database, the virtual temporary table is not guaranteed to have been materialized completely in all relevant partitions when the coordinator triggers re-optimization (i.e., if the coordinator uses an eager voting strategy as discussed earlier). To avoid adding excessive risk to POP, we do not attempt to reuse the materialized rows of a VPMV unless the local temporary tables have been created at all partitions. However, the statistical information for the completed ones can still be exploited despite the fact that the materialized rows cannot be fully reused. The runtime logic creates a statistical view for this VPMV and uses the maximum cardinality (among all cardinalities sent back to the coordinator for that checkpoint) as the per-partition cardinality value for the new plan. This knowledge is then fed back to the optimizer for re-optimization. Note that we theoretically allow the reuse of rows even for partially materialized virtual temporary tables, but this is not implemented in the prototype due to the added complexity and the fact that the solution favors a more conservative re-optimization and a voting strategy/policy.

If the optimizer decided to reuse a temporary partitioned materialized view when generating a new plan, the coordinator keys off the GID of the chosen view and piggybacks each partition's LID in the QEP when distributing the plan to target partitions. This reduces the communication overhead of parallel POP since a separate runtime flow is not needed for sending out the LIDs. When a subordinate partition receives the QEP, it extracts the LID embedded in the plan and fetches the cached local temporary table for the execution.

Figure 3 shows an example of how VPMVs are maintained and reused in the coordinator partition. In this figure, we have two VPMV structures created when re-optimization is triggered. $VPMV_2$ (with GID 2) has been fully populated, which means all participating partitions have completed the materialization and as such it can be exploited as a partitioned materialized view in re-optimization. $VPMV_1$ (with GID 1), however, has only two of the three participating partitions which have materialized their local temporary tables; therefore, $VPMV_1$ cannot be considered for the full reuse. However, to exploit the feedback knowledge captured in $VPMV_1$, the coordinator converts this information to a statistical view and uses 250 (the maximum cardinality value in $VPMV_1$) as the per-partition cardinality for the new plan. The optimizer will then exploit all this information to generate a better plan.

Upon successful completion of the query, all locally cached temporary tables at database partitions can be cleaned up to free up storage. To initiate the distributed garbage col-



GID	LID	Node	Card	Validity Range
1	1	1	120	1..100
1	null	2	null	1..100
1	2	3	250	1..100

GID	LID	Node	Card	Validity Range
2	2	1	120	1..200
2	1	2	110	1..200
2	3	3	180	1..200

Figure 3: Example of intermediate results reuse based on VPMVs.

lection, the coordinator destroys temporary tables' corresponding VPMV structures maintained at its partition and then sends a POP-COLLECT-GARBAGE message to all partitions involved in the query execution. Each subordinator receives the message and deletes all local temporary tables requested to be dropped by the coordinator. After performing the garbage collection the subordinator sends a reply back to coordinator.

8. PERFORMANCE ANALYSIS

We prototyped our solution in a leading commercial parallel DBMS. We studied the performance behavior of our solution (called Parallel POP) in a number of different situations and in different environments. The two main degrees of freedom for the experiments are data correlation and data partitioning. Parallel POP provides great opportunity for workloads with a high degree of data correlation and unevenly partitioned data or skewed data.

Our first set of tests was done on a workload with high data correlation. The tables are partitioned evenly across the nodes. To understand the risk of Parallel POP we will also show results from experiments done on a second workload using completely uncorrelated and uniform data that is also evenly partitioned across the nodes. While we have a high risk of regression in such a workload, we also demonstrate how Parallel POP acts as an insurance policy against optimization errors made by the query optimizer due to incorrect statistics. In the third group of tests we modify the initial workload to introduce data skew across the partitions. With unevenly partitioned data on all nodes we study the impact of the four voting schemes and the difference in using synchronous, asynchronous, or hybrid voting.

8.1 Evenly partitioned data

To test the effect of Parallel POP in a correlated database we use a real-world workload obtained from the Dutch Department of Motor Vehicles (DDMV). The DDMV database schema consists of tables to store information about registered cars including their makes and models, registration information, owner information, dealer information and demographics including street addresses and zip codes. The main view used across all the queries joins 7 tables with cardinalities of 6.3M for car owners and 17.2M for all listed registrations. The cars are registered in 2437 cities across the Netherlands. There is a natural correlation in the data, e.g. between the make of a car and the number of registrations for it. The dataset also includes functional dependencies where for example a make of a car can be derived from its model. There is skew in the data as well with non-uniformly distributed values for the age of car owners or the registration dates of newly released car models. Most queries filter on a car make together with the car model, e.g. 'make = 'VOLKSWAGEN' and model = 'GOLF' and query performance suffers from the lack of correlation information. The queries commonly also apply range predicates on the registration date of a car. With an ever increasing number of car registrations we expect more results for the past few years compared to the same time frame during the 1970s. Given this data skew query performance would benefit from distribution statistics. While the queries differ in the use of filter predicates, they all result in the same 7-way join plan that is required to compute the view. That is a commonly occurring situation in warehouse environments were reports are generated for different market segments or different time slices but using the same global view definition. The queries we use for the tests are the original queries obtained from the DDMV and were not artifically modified for the purpose of the test. We partition the dataset into four nodes using partitioning keys that are commonly used as the join keys for the tested queries. The partitioning keys are either simple primary keys like the person id used to identify owners, or compound keys consisting of make and model to identify cars. Tests that use the DDMV workload run on a six-way S80 PowerPC RS64-III 415MHz system with 16GB physical memory. The tablespaces, indexspaces, temporary tablespaces, and logspaces are physically distributed across RAID arrays set up on SSA storage.

Figures 4 shows the results of the DDMV queries running with and without Parallel POP. The chart shows a speedup $(=\frac{\text{Runtime(query without Parallel POP)}}{\text{Runtime(query with Parallel POP)}})$ for each query.

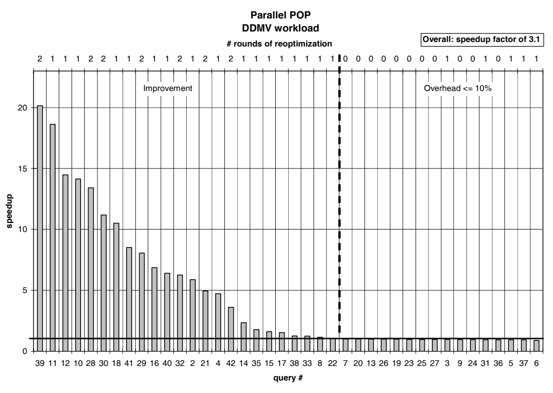


Figure 4: Relative speedup of DDMV queries with Parallel POP.

With Parallel POP we achieve a total speedup of factor 3.1 with the greatest runtime improvements of factor 20. The chart also shows in the top axis the number of re-optimizations used to achieve this level of runtime improvement.

The initial query plans for the DDMV queries are mostly deep-left nested loop join (NLJN) plans and PCHECK points are placed in the outer legs of each NLJN. These kinds of plans provide great opportunities for POP and re-optimization can be triggered early in every node. In this test we use the majority voting scheme with asynchronous voting. That means if an observed cardinality violates the computed validity range for the PCHECK operator in at least two nodes we trigger re-optimization. In the final plan the results of all nodes are delivered back to the coordinator. Depending on the nature of the plan we can also place more PCHECK points in the local portion of the plan that gets executed at the coordinator node and potentially trigger more rounds of re-optimization. We observed in the tests that especially queries with extremely long runtime are improved through Parallel POP. With a single exception no query runs longer than 500 seconds using Parallel POP compared to up to 4000 seconds for the baseline. POP has the ability to protect us from severe optimization errors due to invalid cardinality estimation. In the used DDMV dataset those errors can easily happen for the estimated cardinality of a join result due to data correlation and non-uniformly distributed data.

8.1.1 *Evenly partitioned and uncorrelated data with statistics*

Testing the other end of the spectrum requires a workload with no data correlation, uniform distribution, and evenly partitioned data across nodes. Such a workload would impose a risk of regression on the Parallel POP code because optimization errors should be limited and due to missing or incorrect statistics only. For accurate statistics, however, query optimizer will generate an optimal plan and Parallel POP is not expected to interfere with that plan decision. Because of Parallel POP we expect some overhead in the query runtime due to placement and execution of CHECK points in the plan.

To measure the overhead of Parallel POP and verify the risk of regression we choose the industry standard decision support benchmark TPC-H. The TPC-H database has eight relations to store information on PARTS, SUPPLIERS, CUSTOMERS, ORDERS, and LINEITEMS. We partitioned the database on to four nodes using the primary keys as partitioning keys. The two smallest relations are NATION and REGION and we left these tables un-partitioned on the coordinator node. The hardware used for the TPC-H tests consists of a two-way P615 Power4 1.45GHz system with 4GB physical memory. For physical storage of the database we again used SSA RAID disks. We note that we used a different physical environment to show the stability of progressive optimization in different parallel environments.

The TPC-H queries result in a range of different query plans including all kinds of join methods and join orders, common sub-expressions, group by expressions, and table and index accesses. Figure 5 shows the runtime ratio of the TPC-H queries tested with Parallel POP and compared to an execution without POP. The runtime ratio (in %) is defined as $\frac{1}{\text{Speedup}} \times 100$.

The runtime overhead measured for the full workload amounts to 2%. Given a 5% error margin for the tests, there is no single query with $> 10\%$ runtime overhead due to PCHECK point placement, execution, and voting. The voting scheme used in the tests was again asynchronous majority voting.

The risk of regression due to re-optimization is minimal for the tested workload. Only query 9 triggers a single round of re-optimization. The query is a complex six-way table join and the re-optimization did not cause regression nor did it result in any runtime improvements.

8.1.2 *Evenly partitioned, uncorrelated data without statistics*

To test the stability of Parallel POP we ran another experiment using the TPC-H dataset as above. This time we artificially modify the statistics information about the size of the three biggest relations, LINEITEM, PARTS, and PARTSUPP without actually removing any rows from the dataset. Simply by reducing the table cardinalities, index cardinalities, and column cardinalities we can force the query optimizer into mistakes due to incorrect statistics. The test should prove that Parallel POP can recover from these mistakes with a minimal risk of regression.

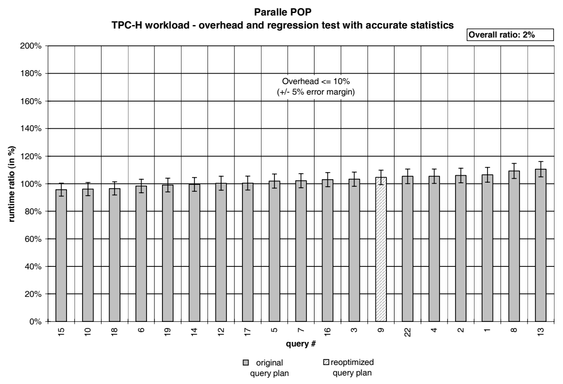


Figure 5: Runtime ratio for TPC-H workload.

Figure 6 shows the speedup factor of this tests. We observed that 70% of the queries recovered from the initial bad plan with re-optimization and have runtime improvements of up to a factor of 22. Re-optimization has the biggest opportunity when it happens early during query execution. Given that we reduced the cardinality of the three big tables, optimizer favors query plans which join these three tables early. The supposedly 'small' resultsets of these early joins turn out to be bigger than expected by the query optimizer. This marks an opportunity for Parallel POP to materialize the results and join them with the remaining plan using a different join strategy. For some queries, however, this also imposes a risk. Suppose we perform a two-way join between PARTS and PARTSUPP early in the plan and adjust the cardinalities of the join result with a PCHECK point. At this point Parallel POP can trigger re-optimization based on the corrected join resultset size but still with the incorrect statistics it has for the LINEITEM table. That results again in a second sub-optimal plan that will get executed until the 3-way join between PARTS, PARTSUPP, and LINEITEM is completed. Only at this fairly late point during query execution Parallel POP can trigger a final round of re-optimization that is based on accurate cardinalities for the result of the three-way join. That problem is known as the problem of 'partial knowledge' and can result in regressions as seen for queries 20, 19, and 8. With 70% if improved queries and only 10% queries with regression we still achieve a great speedup of factor 4.7 for the entire workload.

8.2 Skewed data partitioning

All the tests so far have been done using evenly partitioned data across the four data partitions in the shared-nothing environment. The tests demonstrate that Parallel POP works efficiently using the asynchronous majority voting scheme and performs comparably to a non-partitioned environment. The fact that data was balanced across nodes has proven to be transparent to the progressive optimizer.

To test the effect of different voting schemes and different synchronization modes we have to imbalance the data par-

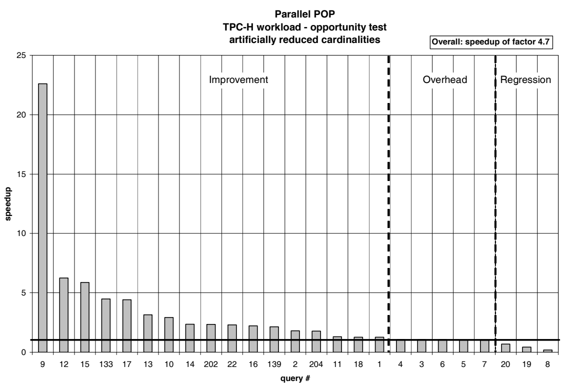


Figure 6: Speedup of TPC-H queries under artificially reduced cardinalities.

titioning. We introduce data skew by randomly removing rows from the DDMV dataset with a different amount of skew in all partitions. For the car owners tables, e.g. we remove 0% of rows from the coordinator partition, 50% of the rows from partition 1, 20% from partition 2, and 10% from partition 3.

As a result of this data skew query optimization will be based on an 'average' size of the tables in the different nodes. While this average may represent the number of tuples in one node well enough, it can be insufficient for all the other nodes. Parallel POP can trigger re-optimization if either 1, 2, 3, or 4 nodes do not agree on the compiled plan based on the used voting scheme.

Figure 7 shows the response times of a subset of queries executed for the DDMV workload using the asynchronous majority voting scheme with skewed partitioning of the data.

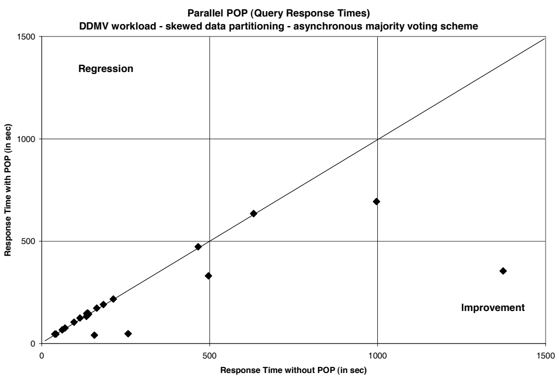


Figure 7: Response times of DDMV queries with skewed data partitioning.

8.2.1 *Effect of different voting schemes*

Different voting schemes can result in different re-optimization strategies with a different number of re-optimization rounds as seen in Figure 8. Based on four partitions used in the tests we trigger re-optimization using the maximum voting scheme as soon as we have at least one node with cardinalities out of bound at a PCHECK point. In the average voting scheme we require the average of four cardinalities to be out of bound and the majority scheme triggers reoptimization with at least two nodes out of bound. There is only one query 38 where the cardinality average violates the validity range based on the observed cardinality at a single node. Query 38 triggers re-optimization with the average scheme but not with the majority scheme. The unanimous voting causes the least number of re-optimizations because we require a unanimous decision which is unlikely to get with skewed data. Only five out of 20 queries qualify for re-optimization using unanimous voting. There is no instance where we have exactly three agreeing votes and therefore we have very similar results for the unanimous and the majority voting schemes throughout the test. That is due to the limited number of partitions where the difference between a majority and a unanimous decision is only 2.

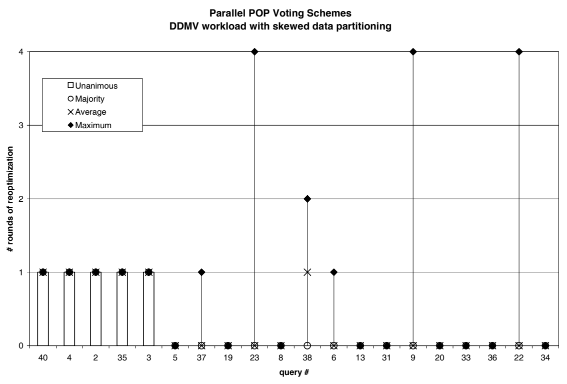


Figure 8: Number of re-optimization rounds for DDMV queries using different voting schemes.

The runtime ratio for the different voting schemes is shown in Figure 9. For the first five queries we see the expected improvements because we have all four nodes requesting re-optimization. All four voting schemes result in the same decision and re-optimization improves the plan across all the nodes. For all other queries we have no re-optimization with the default majority voting scheme. In cases where only one node requests re-optimization, we trigger with the maximum voting scheme and easily cause regressions. If three nodes can agree on a plan and only one requests re-optimization there is a high likelihood of regression because we overrule the majority of three nodes. The results could potentially be different if the requesting node owns 100% of the data and the three non-requesting nodes are almost empty.

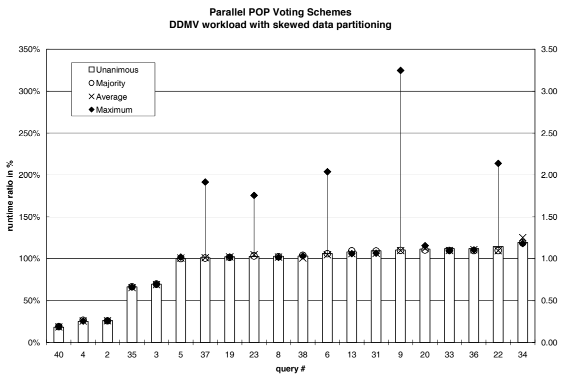


Figure 9: Runtime ratio of DDMV queries using different voting schemes.

8.2.2 *Effect of synchronous, asynchronous, and hybrid voting*

For the given data skew we have established that the majority voting scheme has the best opportunities and carries the least risk for Parallel POP. We continue to use that voting scheme to measure the difference in the use of synchronous, asynchronous, and hybrid voting modes.

Figure 10 shows the runtime ratio for the same set of queries using the synchronous, asynchronous, and hybrid voting mode. Asynchronous voting clearly outperforms synchronous voting because query processing can continue while some nodes did not submit their votes. It seems obvious that for a homogenous environment where all nodes have the same physics processing of 50 rows on one node has to be twice as fast as processing 100 rows on another node. In an asynchronous mode that allows us to return 50 rows to the coordinator node even before the other node has finished execution. On the flip side we miss out on some opportunity for re-optimization because we may have returned 50 rows already before we got to re-optimize based on final vote from the node processing 100 rows. That is what happened in query 6 where Parallel POP triggers two rounds of optimization with the synchronous voting mode but no re-optimization with the asynchronous and hybrid mode. Unfortunately there is no clear advantage of triggering re-optimization in query 6 at this late point in the process. As there is no clear performance difference between the asynchronous and hybrid voting scheme, we recommend using the hybrid voting scheme because it consumes less resources.

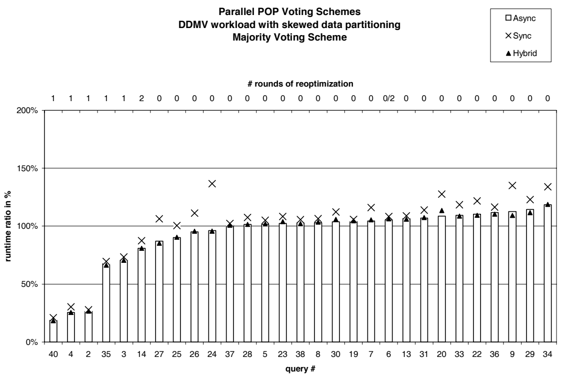


Figure 10: Runtime ratio for DDMV queries using synchronous, asynchronous, or hybrid voting.

8.3 Summary

In a series of tests we have shown how Parallel POP works with different degrees of data correlation and evenly vs. unevenly partitioned data. The stability of progressive optimization in parallel environments has been demonstrated with two different workloads in different physical environments. The true potential of Parallel POP has been demonstrated for highly correlated datasets, for uncorrelated datasets with incorrect statistics that result in optimization errors, and for skewed data partitioning. For skewed data partitioning there are differences in the use of the four voting schemes and three voting modes Parallel POP can operate in. We compared the results of all four voting schemes and the three voting modes and established that the hybrid majority voting generally has the greatest opportunity for runtime improvement through re-optimization with the least risk of regression.

9. RELATED WORK

The challenges and open issues of query optimization in parallel database systems were highlighted by Hasan et al [11]. While considerable progress has been made over the past decade, parallel query optimization remains an active field of research fueled by the advent of data-intensive business intelligence and data warehousing applications. The problem of parallel query optimization has two dimensions: 1. execute the parallel plan right; and 2. execute the right parallel plan. The former involves optimal utilization of the available parallel machine resources (e.g., CPU, memory, disk, network, etc.) to execute the query and is related to the more general problem of resource scheduling and load balancing in parallel and distributed systems [6]. In contrast, the latter deals with selecting the optimal plan to drive the parallel query, which is closely related to relational optimization in a serial architecture (e.g., optimal access and join methods based on accurate cardinality estimates). Our research on parallel POP belongs to this category. To achieve the best possible query performance in a parallel database environment both dimensions need to be optimal and therefore techniques in one area complement the other. A query that is perfectly utilizing all the available resources can still suffer poor performance due to a suboptimal plan and vice versa.

Two general strategies to parallel query optimization exist in the literature. One approach is to split the operation into two phases [11, 12]: serial optimization followed by parallelization. The first phase hides the parallelism metrics from the optimizer and involves finding the optimal serial QEP as in a non-parallel environment. The second phase uses resource utilization and scheduling algorithms to transform the optimal serial QEP into a parallel, schedulable execution plan by taking into account all relevant parameters in a parallel architecture (partitioning information, CPU and communication costs, etc.). The other approach is to have an integrated cost-based query optimizer that understands parallelism and applies that knowledge when costing and generating candidate plans [4]. While initial studies [11] suggested that the two-phase strategy produced good parallel plans, it was found that in practice the approach could lead to query compilation overheads since the same kind of discovery about ordering requirements made by the optimizer would have to be made again during the parallelization phase [4]. Our proposed parallel POP architecture falls under the integrated cost-based optimizer approach.

A large body of work in parallel query optimization has been devoted to the two-phase approach, with particular focus on achieving the best possible parallelizing execution schedule given a serial plan [7, 12, 16]. Extensive research has also been dedicated to re-optimization strategies that explore various dynamic load balancing schemes to allow a parallel or distributed database system to react and adapt to changes in resource configuration and availability [6]. The re-optimization methodology on the execution schedule proposed by [7, 1] aimed at handling irregular and unpredictable data arrival delays from data sources and thus is more applicable to distributed DBMSs rather than parallel DBMSs where our focus is. The aforementioned solutions trigger re-optimization if query execution is detected to have deviated from the optimal resource utilization schedule. In reaction, the schedule is re-optimized to balance the system and better utilize available machine resources. However, the query plan is not re-optimized, and as such they do not address the issue of executing with a suboptimal plan due to cardinality estimation errors.

The work by Ng et al. [17] proposed a method of dynamic plan reconfiguration for parallel DBMSs. However, it primarily focused on operator coordination and state capture/restoration when a plan is reconfigured due to a system change, and did not address the issue of triggering re-optimization due to a suboptimal plan being chosen. Moreover, their proposed system does not allow the reuse of intermediate results after plan reconfiguration, which can lead to significant runtime overheads.

Several solutions for re-optimizing distributed (or federated) database queries have been proposed [9, 10]. They provide calibration functions to help wrappers of remote data sources refresh statistics periodically. However, they are compile-time or just-in-time approaches and hence are still vulnerable to inaccurate cardinality estimates during execution. An extension of POP to federated queries has also been proposed [14]. In federated POP, we can directly use the serial POP solution since a table in a remote site can be treated the same as a local table using a *nickname*. Here, processing at remote resources cannot be targeted for re-optimization. In contrast, in a shared-nothing parallel DBMS, a table is partitioned across a set of nodes, and processing at all the nodes are targeted for re-optimization, and thus, a whole new set of technical problems and challenges are introduced.

General mid-query re-optimization techniques were first studied in [13]. POP and extensions [5, 15] improve upon the work by [13] in terms of re-optimization opportunity and risk metrics. The Redbrick DBMS is a pioneer DBMS that includes a primitive form of progressive optimization in a commercial product. It uses a special strategy to perform star-joins by first computing all the intermediate results of all dimension table accesses, and then uses their cardinalities to choose the optimal access and join methods. While this product uses progressive optimization, it is limited to a very specific execution strategy and does not address the issues of arbitrary checkpoint placement, join reordering, and the reuse of intermediate results.

A drastically different approach to adaptive query optimization is to treat query execution as tuple routing and optimize each tuple individually. In the Telegraph project, adaptive mechanisms are devised to continuously reorder operators in a query plan at runtime[2, 8]. Although their mechanisms can be applied to federated and parallel database systems [2, 19, 18, 8], using per-tuple routing as an re-optimization strategy imposes considerable overhead to query processing which can lead to performance degradation, especially if the initial plan is already optimal and does not require re-optimization. Eddies and STAIRs also cannot reuse intermediate results [8]. Moreover, an eddy uses greedy routing policies and does not consider the overall query execution cost. While this is suitable for Telegraph's interactive and continuous processing metrics, it remains to be seen that, without a traditional optimizer that uses dynamic programming, if this would work for systems with the more common completion time or total work metrics.

10. CONCLUSION

In this paper, we presented a comprehensive and practical solution for progressive optimization in a shared-nothing parallel database to progressively fix sub-optimal plans generated due to the optimizer's estimation errors. We proposed novel placement strategies of the PCHECK operators for parallel query execution plans in Section 4. We presented several variations (synchronous, asynchronous, and hybrid) of the PCHECK operator processing methods and detailed steps using efficient communication protocols in Section 5. We then proposed novel voting schemes (unanimous, majority, averaging, maximum, and weighted) in Section 6 to make a robust global decision to POP. Here, we proposed a combination of eager and lazy voting strategies to ensure the voting mechanism and the overall re-optimization process operate optimally. We proposed a novel method in Section 7 to "virtualize" distributed intermediate results which allows the optimizer to generate view matching structures needed for reuse since the intermediate results are distributed at different processing sites. We have implemented our solution in a leading commercial DBMS. Experiments using our prototype have shown that our solution dramatically improves robustness of query execution, accelerating complex OLAP queries by up to a factor of 22, while incurring only a negligible overhead and a minimal risk of regression.

11. REFERENCES

[1] L. Amsaleg et al., "Dynamic Query Operator Scheduling for Wide-Area Remote Access," DPD, 6(3), 1998.

[2] R. Avnur, J.M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," ACM SIGMOD 2000

[3] C. Baru and G. Fecteau, "An overview of DB2 parallel edition," SIGMOD 1995.

[4] Baru et al., "DB2 Parallel Edition," IBM System Journal, 1995.

[5] S. Babu, P. Bizarro, and D. DeWitt, "Proactive Re-Optimization," SIGMOD 2005.

[6] L. Bouganim et al., "Dynamic Load Balancing in Hierarchical Parallel Database Systems," The VLDB Journal, 1996.

[7] L. Bouganim et al., "Dynamic Query Scheduling in Data Integration Systems," ICDE 2000.

[8] A. Deshpande and J.M. Hellerstein, "Lifting the Burden of History from Adaptive Query Processing," VLDB 2004.

[9] W. Du, R. Krishnamurthy, and M.-C. Shan, "Query Qptimization in Heterogeneous DBMS," VLDB 1992.

[10] G. Graefe and K. Ward, "Dynamic Query Evaluation Plans," SIGMOD 1989.

[11] W. Hasan, D. Florescu, and P. Valduriez, "Open Issues in Parallel Query Optimization," SIGMOD Record, 25(3), 1996.

[12] W. Hong and M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS," PDIS 1991.

[13] N. Kabra, D. DeWitt, "Efficient Mid-Query Re-Optimization of Suboptimal Query Execution Plans," SIGMOD, 1998

[14] S. Ewen et al., "Progressive Query Optimization for Federated Queries ," EDBT 2006.

[15] V. Markl et al., "Robust Query Processing through Progressive Optimization," SIGMOD 2004.

[16] S. Manegold et al., "On Optimal Pipeline Processing in Parallel Query Execution," Technical report, CWI, 1998.

[17] K. W. Ng et al., "Dynamic Reconfiguration of Sub-Optimal Parallel Query Execution Plans," TR, UCLA, 1998.

[18] V. Raman, A. Deshpande, and J. Hellerstein, "Using State Modules for Adaptive Query Processing," In ICDE, 2003.

[19] V. Raman and J. Hellerstein, "Partial Results for Online Query Processing," In SIGMOD, 2002.

[20] V. Markl et al., "Progressive Optimization in Action," In VLDB, 2004 (demo).

[21] A. Shatdal, "Order Based Analysis Functions in NCR Teradata Parallel RDBMS," In EDBT 2000.

[22] D. Wildfogel, and R. Yerneni, "Efficient Testing of High Performanfce Transaction Processing Systems," VLDB 1997.