Databases and Information Systems VIII
H.-M. Haav et al. (Eds.)

doi:10.3233/978-1-61499-458-9-233

Cost-Based Data-Partitioning for Intra-Query Parallelism

Yanchen LIU [a,b], Masood MORTAZAVI [a,1], Fang CAO [a], Mengmeng CHEN [a] and Guangyu SHI [a]

[a] *Huawei Innovation Center, Santa Clara, CA, USA*
E-mail: {yanchen.liu, masood.mortazavi, fang.cao, mengmeng.chen, shiguangyu}@huawei.com
[b] *City College, CUNY, New York, NY, USA*

Abstract. Contemporary DBMS systems already use data-partitioning and data-flow analysis for intra-query parallelism. We study the problem of identifying data-partitioning targets. To rank candidates, we propose a simple cost model that relies on plan structure, operator cost and selectivity for a given base table. We evaluate this model in various optimization schemes and observe how it affects degrees of parallelism and query execution latencies across all TPC-H queries: When compared with the existing naïve model which partitions the largest physical table in the query, our approach identifies significantly better partitioning targets thus resulting in sinificantly higher degree of resource utilization and intra-query parallelism for most queries while having little impact on the remaining queries in the TPC-H benchmark.

Keywords. data-partitioning, intra-query parallelism, multi-core systems, optimization

Introduction

Peforming queries on large data-sets is the norm in many applications [1,2,3]. In analytic query processing [4], the relative advantage of column stores has been well-established [5,6,7]. Column-oriented databases process only the fraction of data relevant to the query; they offer better possibilities to take advantage of compression techniques; they are highly suitable for super-scalar CPU architectures [7,8,9,10].

Chips have integrated increasingly larger number of cores [11], providing an opportunity for innovations in parallel processing of queries. Databases lend themselves to various kinds of parallelism: I/O, inter-query(inter-operation), and intra-query(intra-operation) [12]. Data decomposition is a fundamental technique in parallel computing [13]. Partitioning techniques have been used to provide I/O parallelism through load-balancing across multiple units [14,15]. NoSQL databases take full advantage of this method of I/O parallelism [16] as do tradi-

[1] Corresponding author: Masood Mortazavi, masood.mortazavi@huawei.com

tional and NewSQL databases that rely on "shared-nothing" system architectures [17].

Intra-query parallelism in multi-core environments has always posed scheduling and system design challenges. Because the exhaustive search of all possible schedules is prohibitive even for relatively simple queries, scheduling systems rely on some degree of heuristics [18]. Task to thread assignment requires data partitioning, thread and memory management models [19]. In distributed parallel query processing we need to resolve an even larger set of problems. For example, Scope extends a transformation-based optimizer for distributed systems [20]. It uses different types of data exchange operators to generate parallel execution plans and introduces data exchange operators and rules to generate optimized distributed query plans. The size of the dataset can significantly affect the efficiency of such optimizers. In [21], an Xchange family of operators introduce intra-operator parallelism. This work also discusses strategies and transformation rules to rewrite a given non-parallel query execution tree into its optimal parallel counterpart in multi-core environments. Parallel task scheduling, except for the most trivial cases, has been proven to be an NP-hard problem [22,23].

We propose an approach to improve intra-query parallelism, which fits well within the tradition of parallel processing using the data decomposition model [13]. When deciding which base table to partition, our heuristics rely on the structure of query plans, the effective selectivity of operators, the cost of each such operator given the size of its input, and the size of base tables used in the query. We develop three separate data-partitioning optimizers and investigate the effect of each across a set of standard queries. It is also verifed that our apporach outperformes the "naïve" partitioning method which always partitions the largest table.

The remainder of this paper is organized as follows: The system model is described in section 1. In section 2, we describe our approach to cost-based data-partitioning optimization. In section 3, we introduce the scope of our evaluation and describe the metric we have used to identify the best approach. We also summarize our implementation and testing approach and discuss our experimental results. We conclude this paper with secion 4, where we also review some future directions suggested by our work.

1. System Model

1.1. Problem Statement

We use *(base) table*, *binary association table* or *BAT* interchangeably to refer to the objects of a query plan. The objects include *fields* or *columns* and are stored in a *database*. A *binary relational algebra* (in this paper, also called *binary algebra*) is a set of *instructions* (or *operators*) that operate on binary relationships, or BATs, and produce other BATs. The query evaluation engine sees a query *plan* as a set of instructions, starting with some input BATs and some query parameters and ending in some resulting relation. It is possible to create semantically equivalent plans for a given query. An *optimizer* selects an optimal plan from the set of equivalent plans. It is possible to chain a set of optimizers in an *optimizer pipeline*.

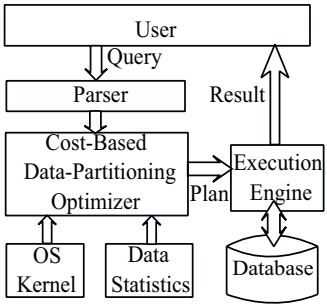

Figure 1. Simplified diagram of the architectural modules

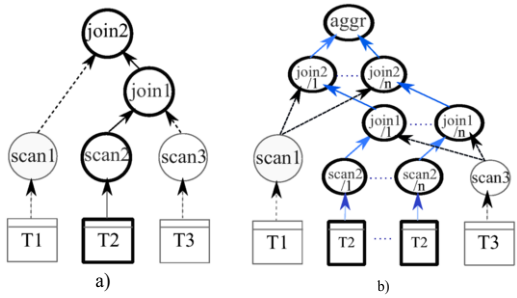

Figure 2. a) Cost accumulation chain in a query plan; b) The impact of data-partitioning plan

Data-flow (or task-dependency) analysis provides a platform for intra-query parallelism. An input can be *data-partitioned* into pieces before it is fed to a chain of instructions. A bounded *sub-plan* needs to be *replicated* in order to perform equivalent operations on all partitions. Data-partitioning allows the operation (or chain of operations) to be performed on parallel threads. As query execution progresses, these parallel threads must produce results towards a common output, and the remaining tasks will potentially be executed with fewer threads.

1.2. System Model

Figure 1 shows the basic architectural modules involved in query formulation and cost-based, data-partitioning optimization. In Figure 2, operators are shown as circles, base tables are shown as squares and data flows are given as arrows. Leaf operators act on tables (or columns) and produce *intermediate* results that are used by operators higher in the plan.

Using our cost model, we rank tables involved in the query. Those with the highest processing costs are ranked highest as potential partitioning targets. Once ranking is complete, we can split one or more of the "costliest" tables. (See Figure 2 a) for a cost accumulation chain, and see Figure 2 b) for the impact of a split on the plan structure.) With a "split" plan, we have a decomposed task dependency graph that allows for parallel processing [13].

There always exists a trade-off between the sophistication of the heuristic cost models *vs.* their degree of robustness on the one hand and the length of time spent in computing these models on the other. In the current model, we have elided issues related to overhead costs such as start-up costs, skew costs, resource contention costs and reassembly costs [12]. The *start-up* costs are generally useful to consider when there is a real choice between parallelism and single-resource execution. For large data sets, the start-up overheads are usually justified. The raw resource *contention* costs, non-uniform access to resources, and how to avoid them are well beyond the scope of this paper. Here, we also ignore issues related to *skew* costs. We assume processing occurs on arrays with relatively even distributions.

Nevertheless, there will always be some amount of skew when dealing with real, as opposed to experimentally generated data. Our argument regarding *assembly* costs is similar to our analysis of start-up costs. They are worth paying when dealing with large data-sets. Our *cost accumulation model* provides a heuristic for determining a potential *critical path* in the *task-dependency graph* [13] in a query plan. By utilizing our data-partitioning optimizer, it is also possible to see how partitioning multiple base tables can affect the performance.

2. Cost-based Data-Partitioning

2.1. Cost Parameters

Our model allows for several factors in cost calculation: Num_i represents the number of records in the i^{th} table. Op_j represents the relative physical cost of the j^{th} operator in a "cost accumulation" chain of operators. (The operators in a given cost-accumulation chain are data-dependent.) $S_{j,k}$ represents selectivity after the k^{th} operator in the operator chain leading to Op_j.

Note that for accuracy, one can maintain and update derived values for Op_j and $S_{j,k}$ based on actual execution of queries on large data-sets. We expect to do more work in this area in the future. Here, we focus on how cost accumulates in the plan and how this accumulated cost can be used to make parallelization decisions.

2.2. Cost Accumulation

We use the following formula to calculate accumulated cost along an operator chain:

$$cost_i = Num_i \sum_{j=1}^{N} (Op_j \prod_{k=1}^{j} S_{j,k-1}) \tag{1}$$

Here, $cost_i$ is the accumulated cost of the i^{th} table in a relational database—it is calculated for a particular plan—and is "accumulated" over a chain of operators in the plan. Variable N is the number of operators in the chain of operators involving the i^{th} table, all the way to the last-considered operation which will benefit from data-partitioning parallelism. As noted earlier, variable Op_j represents the operator "unit cost" per record. We do not put emphasis on modeling this operator-specific cost. In our implementation environment, we can make simplifying assumptions regarding relative costs of operators because they all act on binary association tables. The product in Eq. (1) is the accumulated and effective selectivity up to the operator j. $S_{j,0}$ is assumed to be 1 for all j.

Our algorithms accumulate cost up to an operator. Usually, this "destination" operator is some join operator where the biggest BATs in the query meet. Figure 2 shows an example for a cost accumulation chain. There, we are accumulating the cost of operations on $Table2$ as these costs add up all the way to the top of the operator tree for the query in question.

Once the accumulated *cost* for a table is calculated to be larger than others, this table will be partitioned so that all related executions in the chain of operators can run in parallel on the available compute cores. For example, as indicated in the plan in Figure 2, if *cost* for $Table2$ is larger than the one for $Table3$ before $JOIN1$, and *cost* for $Table2$ is larger than the one for $Table1$ before $JOIN2$, a cost-based data partitioning optimizer would partition $Table2$ before $SCAN2$. It is also possible to use the available processing resources to data-partition additional tables. (We discuss this further in Section 2.4.)

Figure 2 also shows the impact of data-partitioning $Table2$ on the query plan. In this case, $Table2$ is partitioned into n pieces. The number of partitions n is usually determined by the the number of cores or HW threads—or whatever symmetric cores or symmetric resources available in the system. In the plan represented in Figure 2 b), $SCAN2$, $JOIN1$ and $JOIN2$ can execute in parallel. At the end, the sub-results produced after $JOIN2$ operations need to be aggregated together.

Because our optimizers seek to partition the table which represents the greatest cost in the overall plan, the resulting parallel execution will yield less idleness among resources and faster execution times for the query.

2.3. Cost Ranking

Multiple data-partitioning candidates can exist, and one needs to discriminate among these candidates according to some ranking procedure.

Our candidate-tuple ranking model is given by Eq. (2).

$$\begin{aligned} & cost(< T_{1,i} >, ..., < T_{n,i} >) \; < cost(< T_{1,j} >, ..., < T_{n,j} >) \\ & \iff \sum_{k=1}^{n} (cost(< T_{k,i} >)/\alpha_k) < \sum_{k=1}^{n} (cost(< T_{k,j} >)/\alpha_k) \end{aligned} \tag{2}$$

Here, $< T_{k,i} >$ is the k^{th} member of the candidate tuple i whose ranking needs to be compared with another candidate tuple and $\alpha_1, ..., \alpha_n$ represents the data-partitioning "distribution" policy:

$$\mathcal{P} \equiv < \alpha_1, ..., \alpha_n > \tag{3}$$

Note that, in general, we take α_i/α_j to the ratio of the number of partitions in T_i over the number of partitions in T_j. We should emphasize here that our cost model in Eq. (1) only helps identify candidate tuples by evaluating their processing costs. We still need a partitioning policy such as $\mathcal{P}$ to determine the number of partitions for each member of the candidate tuple.

The ranking pattern in Eq. (2) emphasizes additive cost to discover which tuple has the largest effective cost even after each member of the tuple has been partitioned according to policy $\mathcal{P}$. Given the number of cores in our test platform, we have only considered two-member tuples with a data-partitioning "distribution" policy of $< 30, 2 >$. More generally, data-partitioning distribution policy $\mathcal{P}$ can be determined dynamically and per-query, and it can be adaptively adjusted.

2.4. Data-Partitioning Strategies

We explored and compared three separate data-partitioning candidate selection strategies and a naïve strategy.

2.4.1. Naïve (Default)

The *Default* or *Naïve* (but still powerful) strategy selects the largest table as the target for data-partitioning. The naïve approach assumes that any other table on a longer operator chain will have a reduced size after each predicate-constraint has been satisfied. To show that this is not an entirely unreasonable assumption, consider the case where selectivity is uniformly distributed to be 50% across all "predicate" operators. The longer operator chain even if it is long can only amplify the cost of a table by at most $\times 2$ $(= 1 + 1/2 + 1/4 + ...)$, assuming individual operator costs remain constant. So, whenever the second largest table is less than $1/2$ the size of the largest table, one may wonder whether taking plan structure and other large tables into account would make any difference.

2.4.2. First-Join (1^{st}-Join)

There can be two relatively large tables, T_j and T_k, which must be joined prior to any joins with the largest table T_l. Data-partitioning the largest table will lead to sub-optimal distribution of cores to pieces because there will be fewer cores available to do the combined, earlier work on T_j and T_k. To solve this problem, the *First-Join* strategy selects a data-partitioning candidate that is both large enough and that feeds the first join operation. The relative success of this strategy depends on the proper selection of the threshold for "large enough".

2.4.3. 1-Table Cost-Based (1-TCB)

The *1-Table Cost-Based* data-partitioning strategy selects a single candidate for data-partitioning. It uses our heuristic cost model (Eq. (1)). In this case, the data-partitioning optimizer walks through all instructions and accumulates the cost related to each table based on whether it or any of its dependants appear in that instruction. A ranking comparison (Eq. (2), with tuple of order 1) can now produce an ordering of tables with their relative costs.

2.4.4. 2-Table Cost-Based (2-TCB)

Using our heuristic cost model (see Eq. (1)), *2-Table Cost-Based* strategy selects a tuple of two candidates for data-partitioning. Algorithm given in Figure 3 describes how the candidate list of 2-TCB tuples is generated. Eq. (2) is then used to discriminate among these generated tuples and select the two tables that need to be partitioned.

2.5. Core Algorithm for 2-Table Cost-Based Candidate Selection

The core algorithm for 2-TCB (see Figure 3) consists of four parts:

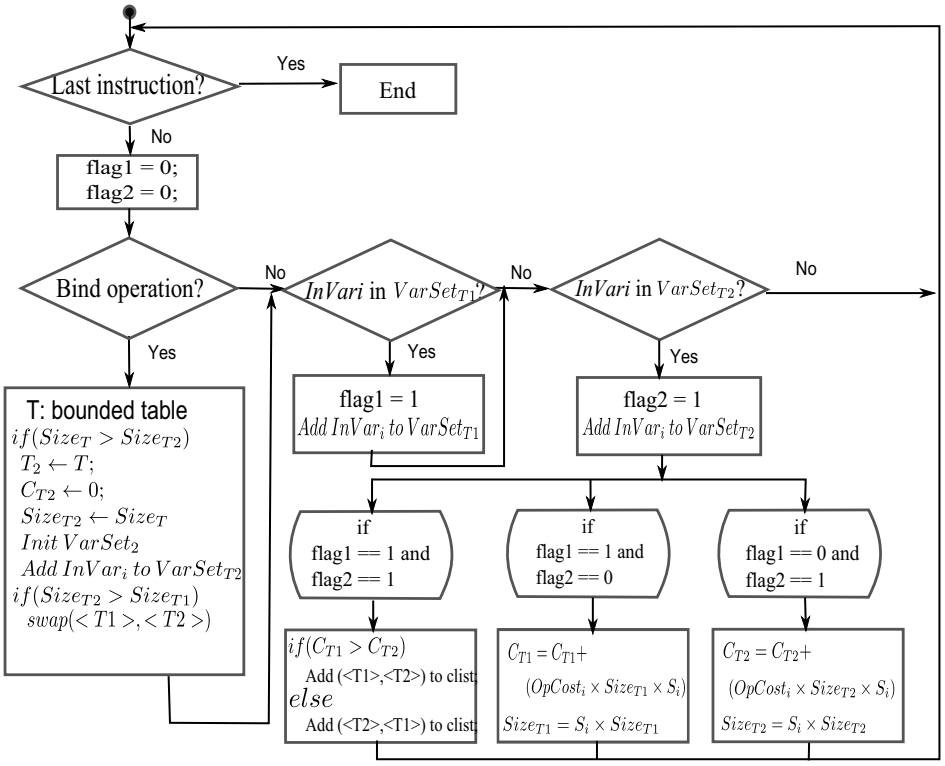

Figure 3. Identifying candidate list for data-partitioning

- *Selecting potential candidates.* Here, physical tables that will be bound to variables in the relational binary algebra are being selected as seeds for cost accumulation chains.
- *Maintaining variable dependencies.* Here, variable dependencies are maintained by ensuring that the "out" variables of an instruction are added to variable set depending on a table as long as the "in" variables of that instruction are also in the same variable set. (Dependent variable sets are constructed per BAT.) It is this part of the algorithm that incorporates the essential structural characteristics of the plan into the data-partitioning optimizer.
- *Detecting joins.* When the operator chain arrives at a join operation, we add the table pair as a candidate tuple to the list. The notation $< T >$ indicates not only the table but an object including its cost value and dependent variable set. The cost value is later used in ranking. (See Eq (2).)
- *Accumulating costs.* Before moving to evaluate the next relational binary algebra instruction, we add the cost of the current operation to the cost of the current candidate pair. We have elided a part of the algorithm which re-initializes the "current" candidate tables after reaching a join operator.

Results in Section 3 are reported for a specialization of this algorithm: Once the first join is detected by the current top-most tuple, we let the exploration for candidates stop. For small plans—most of the type we were dealing with—this policy for terminating the exploration worked well.

3. Evaluation

The heuristics used in modeling costs often leads to lop-sided improvements — some queries may improve but others degrade significantly. In order to reduce the bias in our optimizations, we analyzed all optimizers we implemented on all 22 TPC-H queries. We were interested in variations of 10% or more.

To explore the efficiency of our proposed algorithms and heuristic cost model, we implemented and compared our strategies in MonetDB, an open-source, column-oriented database management systems [6,24]. We coded our partitioning strategies as MonetDB optimizers. Our data-partitioning optimizers are placed in the optimizer pipeline right before MonetDB's "mitosis" optimizer which generates the parallel instruction subgraphs for each piece of the partitioned table. We modified MonetDB's February 2013 SP1 code to implement and test the strategies reported here.

3.1. Testing

While we have studied all TPC-H queries and have reported the results below, we have paid some extra attention to $Q3$, $Q5$, $Q8$, $Q10$, and $Q18$. To test the correctness and performance of our implementation, we used standard TPC-H benchmark at scale factors $SF100$ and $SF200$. This implies approximately 100GB and 200GB of data respectively.

3.1.1. Environment

Our laboratory environment consists of both physical machines and clusters of Linux containers (also known as LXCs [25]). We have built this laboratory environment for both single-node and distributed query processing experimentation and analysis. Most of the experiments related to this paper were single-node experiments executed within a Linux container.

3.1.2. Resources

Physical compute resources, within which we run our experimental Linux containers have the following configuration: Ubuntu Linux 3.5.0-17-generic x86_64, 188 GB of RAM, 2x Intel(R) Xeon(R) CPU, E5-2680 @ 2.70GHz, 32 effective cores @ 1.2GHz, cache size: 20480 KB.

3.1.3. Measurement Tools

We used MonetDB's Tomograph [26] and Stethoscope [27], as well as standard Unix tools to view and measure parallelism and overall execution times. Both Stethoscope and Tomograph provide per-operation execution times—with Stetho-

Table 1. The effect of data-partitioning strategy on TPC-H query latency for $SF100$ (about 100GB of data).

TPC-H	Default	1^{st}-Join	1-TCB	2-TCB	TPC-H	Default	1^{st}-Join	1-TCB	2-TCB
$Q1$	21.01	20.52	21.04	20.44	$Q12$	3.42	3.42	3.36	3.45
$Q2$	0.63	0.78	0.59	0.61	$Q13$	80.85	78.3	78.59	81.31
Q3	18.6	13.29	**13.21**	**13.31**	$Q14$	3.54	3.54	3.41	3.55
$Q4$	6.35	6.74	6.12	7.18	$Q15$	9.29	9.1	9.63	9.63
Q5	7.16	**4.43**	**4.43**	**4.36**	$Q16$	12.43	12.21	11.23	11.97
$Q6$	2.5	2.51	2.55	2.6	$Q17$	3	3.01	2.91	2.96
$Q7$	2.87	2.72	4.01	2.75	**Q18**	**24.26**	38.59	28.62	**23.67**
Q8	12.36	**4.92**	**4.93**	**5.01**	$Q19$	5.4	5.17	5.08	5.17
$Q9$	38.51	39.35	40.98	37.21	$Q20$	4.86	4.85	4.37	4.65
Q10	**10.9**	19.49	**10.48**	**10.38**	$Q21$	21.96	20.26	19.31	22.32
$Q11$	3.78	3.04	2.8	2.87	$Q22$	10.05	9.67	9.6	9.75

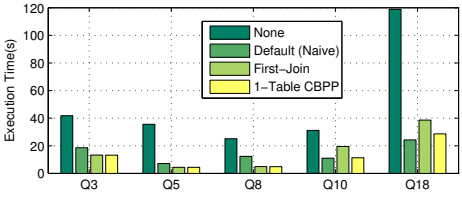

Figure 4. Latency variation with data-partitioning strategies—representative TPC-H queries

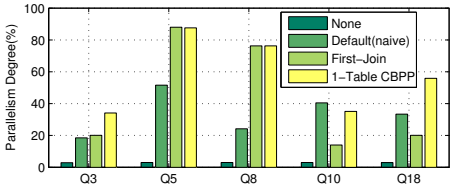

Figure 5. Parallelism variation with data-partitioning strategies—representative TPC-H queries

scope providing even more fine-grained details. We used these more fine-grained reports as well as Tomograph's graphic reports to be very useful tools in suggesting optimization strategies.

3.2. Results

Table 1 presents an overview of the effect of various data-partitioning strategies on TPC-H query latencies. While each of the comparisons in Table 1 can be studied separately, we have highlighted some of the results which will guide the reader through some further analysis we provide below. Figure 4 and Figure 5 show the latency and the degree of parallelism obtained by each of the strategies when applied to TPC-H queries $Q3$, $Q5$, $Q8$, $Q10$ and $Q18$. The degree of parallelism is computed by summing the duration of time each HW thread has been doing useful work and dividing it by the total time of query execution multipled by the number of HW threads available and involved in the query execution.

3.3. Discussion

First, it is worth noting that while adding our data-partitioning optimizer, we are still relying on MonetDB's data flow analysis, its "mitosis" optimizer which produces "replica" sub-plans for patitions, and its thread-to-core assignment system. Given this environment, our data-partitioning optimizers manipulate the effective set of "replica" tasks that sub-plan "mitosis" creates and that dataflow analysis assigns to cores as independent task graphs.

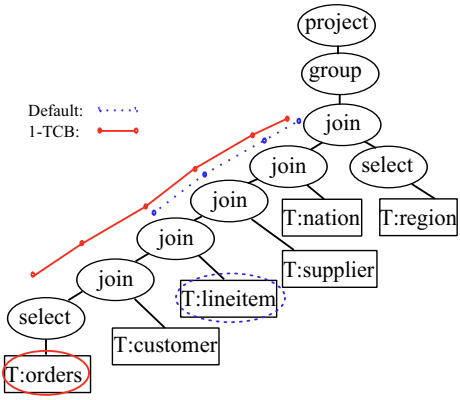

Figure 6. Summary plan for TPC-H $Q5$

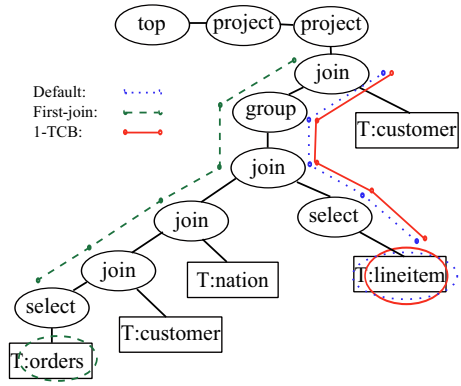

Figure 7. Summary plan for TPC-H $Q10$

With default (or naïve) data partitioning strategy (see 2.4.1) , which chooses the largest table for partitioning, no consideration is given to the structure of the plan and the relative amount of work when there are other relatively large tables. The naïve strategy only guarantees that those operations depending on data from the largest table will be executed in parallel. For those cases where some tasks also depend on other large tables, system resources remain idle and cannot be fully utilized. Given the low levels of parallelism obtained with the naïve partitioning optimizer for a large class of queries, we explored and developed the remaining three optimizers (described in 2.4.2, 2.4.3 and 2.4.4).

For example, table "lineitem" is the largest among the tables referred in the TPC-H Q5. As shown in Figure 6, since the default strategy data-partitions the "lineitem" table, the "select" operation on "orders" table and "join" operation on "customer" cannot be executed in parallel. On the other hand, since cost calculation indicates "orders" table consumes more resource than "lineitem" table does (before the join operation with "lineitem"), our algorithm suggests "orders" table for data-partitioning. A similar analysis can be done for other queries where we see a relatively large difference—queries $Q3$, $Q5$, $Q10$ and $Q8$ being most notable.

Our 1^{st}-Join data-partitioning strategy (see 2.4.2) always targets the first table it encounters in a "join" operation as long as the table's size is above our heuristically selected threshold. The 1^{st}-Join strategy proves effective for both $Q5$ and $Q8$ (see Table 1). Although we don't explicitly calculate the cost in this case, the "estimated" cost here can be said to be highest if the first table to participate in a "join" is above a certain size. However, this 1^{st}-Join estimation of cost proves to be too simplistic. The strategy degrades latencies in $Q10$ and $Q18$ even as it improves them significantly in $Q5$ and $Q8$. (See Table 1 for details.)

In our initial analysis of 1^{st}-Join results, we noticed that, in contrast to the case of TPC-H $Q5$, when the majority of physical tasks are based on the first-joined table (in the case of $Q5$, the "orders" table), the 1^{st}-Join data-partitioning strategy no longer produces sufficient gains when the largest table suffers more processing in some other queries. To elaborate on this concept, we considered the case where 1^{st}-Join data-partitioning was applied to $Q10$, where it degraded the

performance. As shown in the Figure 7, "orders" table is the first base table to suffer a "join" operation in TPC-H Q10 plan. According to the rules of 1^{st}-Join partitioning strategy, "orders" passes the threshold criteria and becomes a target for data-partition. However, the "select" operation on the "lineitem" table costs more than the operations on "orders" table before the first "join" operation. In this case, choosing "orders" table as the partitioned table will lead to reduced parallelism. (It is worth noting that MonetDB's Tomograph [26] was extremely useful in this kind of analysis and in clearly visualizing these trade-offs, e.g. , the relative cost of "select" operation on "lineitem" vs. all operations on "orders" prior to their "join" operation.) Based on this analaysis, we added a cost-based data-partitioning strategy. As can be observed in Table 1, in contrast to 1^{st}-Join, cost-based data-partitioning works well on $Q10$ and $Q18$ queries. Cost-based data-partitioning also does well with respect to those queries where 1^{st}-Join data-partitioning shows significant latency improvements, i.e. $Q5$ and $Q8$. As we made incremental improvements in optimization techniques, we observed that in the majority of the queries, either $1-TCB$ or $2-TCB$ offered the best strategy. Only in $Q6$, were neither of these strategies the best, with a worst case degredation of 4%.

4. Conclusions

We have presented four cost-based data-partitioning strategies for improving intra-query parallelism in a column-oriented database running on multi-core processors. Since optimal parallel scheduling on symmetric resources is NP-hard [22], we have relied on a heuristic model to converge on a data-partitioning target. Cost heuristics tend to have uneven impact on query performance, i.e. improving latency for some queries while degrading it for others. As a simple guard against this bias, we found TPC-H benchmark can be useful in evaluating and comparing our data-partitioning optimizers.

We implemented three separate data-partitioning candidate selection algorithms and compared them with the naïve algorithm which targets the largest table for data partitioning. In contrast to the naïve selection approach, our alternative strategies account for plan structure, selectivity and specific operator cost. The results (even when obtained with default unit values for per-operator cost and selectivity) show that significant room for improvement exists over the naïve models.

In future work, we expect to introduce specific per operator (or at least, per operator class) cost and a dynamic system for updating selectivity statistics. We plan to implement our multi-table scheduling heuristics over larger and varying sets of cores, input data, and parallelism distribution policies. We will also investigate the application of adaptive scheduling and asymmetric resource configurations to optimization strategies for parallel query processing.

References

[1] Melnik S., et al. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 2011, 54(6), 114-123.

[2] Lamb A., et al. The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 2012, 5(12), 1790-1801.
[3] Stonebraker M. Scientific Data Bases at Scale and SciDB. *CERN Computing Colloquium*, May 2013.
[4] Chaudhuri S., Dayal U. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 1997, 26(1), 65-74.
[5] Abadi D. J., Boncz P. A., Harizopoulos S. Column-oriented database systems. *Proc. VLDB Endow.*, 2009, 2(2), 1664-1665.
[6] Idreos S., et al. MonetDB: two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 2012, 35(1), 40-45.
[7] Stonebraker M., et al. C-store: a column-oriented DBMS. *Proc. VLDB Endow.*, 2005, 553-564.
[8] Abadi D. J. Query execution in column-oriented database systems [dissertation]. MIT, 2008.
[9] Boncz P. A., Zukowski M., Nes N. J. MonetDB/X100: hyper-pipelining query execution. *Proceedings of International conference on verly large data bases (VLDB) 2005*, Very Large Data Base Endowment, 2005.
[10] Zukowski M. Balancing vectorized query execution with bandwidth-optimized storage [dissertation]. Universiteit van Amsterdam, 2009.
[11] Intel *Teraflops Research Chip*, `http://www.intel.com/content/www/us/en/research/intel-labs-teraflops-research-chip.html`.
[12] Silberschatz A., Korth H., Sudarshan S. *Database Systems Concepts*, McGraw-Hill, Inc., 2006.
[13] Grama A., et al. *Introduction to parallel computing: design and analysis of algorithms.* Pearson Education Limited, Essex, 2003.
[14] Bischof S., Schickinger T., Steger A. Load balancing using bisectors - A tight average-case analysis. *ESA*, Springer, 1999
[15] DeWitt D., Gray J. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 1992, 35(6), 85-98.
[16] Silberstein A., et al. PNUTS in Flight: Web-Scale Data Serving at Yahoo. *IEEE Internet Computing*, 2012, 16(1), 13-23.
[17] Pavlo A., et al. A comparison of approaches to large-scale data analysis. *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, New York, NY, USA, 2009.
[18] Krikellas K., Cintra M., Viglas S. Scheduling threads for intra-query parallelism on multicore processors. *EDBT*, University of Edinburgh, Edinburgh, Scotland, 2010.
[19] Viglas S. A comparative study of implementation techniques for query processing in multicore systems. *IEEE Transactions on Knowledge and Data Engineering*, 2014, 26(1), 3-15.
[20] Zhou J., Larson P. A., Chaiken R. Incorporating partitioning and parallel plans into the SCOPE optimizer. *Proceedings of ICDE Conference*, 2010. 1060-1071.
[21] Anikiej K. Multi-core parallelization of vectorized queries [dissertation]. University of Warsaw and VU University of Amsterdam, 2010.
[22] Du J., Leung J. Y., Complexity of scheduling parallel task systems. *SIAM J. Discret. Math.*, 1989, 2(4), 473-487.
[23] Ganguly S., Hasan W., Krishnamurthy R. Query optimization for parallel execution. *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, San Diego, California, USA, 1992. 9-18.
[24] MonetDB, `http://www.monetdb.org`
[25] `http://lxc.sourceforge.net/`
[26] Gawade M., Kersten M. Tomograph: highlighting query parallelism in a multi-core system. *Proceedings of the Sixth International Workshop on Testing Database Systems*, New York, NY, USA, 2013.
[27] Gawade M., Kersten M. Stethoscope: a platform for interactive visual analysis of query execution plans. *Proc. VLDB Endow.*, 2012, 5(12), 1926-1929.