To appear in: Proceedings of the 6. East-European Conference on
Advances in Databases and Information Systems (ADBIS'2002),
Bratislava, Slovakia, September 8-11 2002.

OLAP Query Evaluation in a Database Cluster: a Performance Study on Intra-Query Parallelism[*]

Fuat Akal Klemens Böhm Hans-Jörg Schek

Swiss Federal Institute of Technology Zurich, Database Research Group
Institute of Information Systems, ETH Zentrum, 8092 Zürich, Switzerland
{akal, boehm, schek}@inf.ethz.ch

Abstract. While cluster computing is well established, it is not clear how to coordinate clusters consisting of many database components in order to process high workloads. In this paper, we focus on Online Analytical Processing (OLAP) queries, i.e., relatively complex queries whose evaluation tends to be time-consuming, and we report on some observations and preliminary results of our PowerDB project in this context. We investigate how many cluster nodes should be used to evaluate an OLAP query in parallel. Moreover, we provide a classification of OLAP queries, which is used to decide, whether and how a query should be parallelized. We run extensive experiments to evaluate these query classes in quantitative terms. Our results are an important step towards a *two-phase query optimizer*. In the first phase, the coordination infrastructure decomposes a query into subqueries and ships them to appropriate cluster nodes. In the second phase, each cluster node optimizes and evaluates its subquery locally.

1 Introduction

Database technology has become a commodity in recent years: It is cheap and is readily available to everybody. Consequently, *database clusters* likewise are becoming a reality. A database cluster is a network of workstations (PCs), i.e., commodities as well, and each node runs an off-the-shelf database. In the ideal case, a database cluster allows to *scale out*, i.e., it allows to add more nodes in order to meet a given performance goal, rather than or in addition to modifying or tuning the nodes.

Even though its advantages seem to be obvious, it is not clear at all how data management with a database cluster should look like. Think of a cluster that consists of a large number of nodes, e.g., more than 50. How to make good use of such a cluster to work off a high database workload? How to deal with queries and updates, together with transactional guarantees? In broad terms, the concern of our PowerDB research area [20] is to address these issues and to develop a cluster coordination infrastructure. The infrastructure envisioned completely hides the size of the cluster and the states of its nodes from the application programmer. For the time being, we assume that there is a distinguished node (*coordinator*) with 'global knowledge' as

[*] Project partially supported by Microsoft Research.

part of the infrastructure. Clients submit requests, i.e., queries and updates, only to the coordinator and do not communicate directly with other nodes of the cluster.

With respect to data distribution, we apply standard distributed physical design schemes [1]. The design scheme determines the query evaluation. The main design alternatives are full replication, facilitating high inter-query parallelism, and horizontal partitioning, improving intra-query parallelism. Recent investigations [2],[3] have examined these design alternatives for OLAP queries. However, given a large number of nodes, it might not be a good idea to have just one distributed physical design scheme that spans over the entire cluster. Instead, several schemes may coexist, e.g., the data might be fully replicated on three cluster nodes while the remaining nodes each hold a partition of a fact table and replicas of all the other tables. We refer to such a scheme as a *mixed distributed physical design scheme*, as opposed to *pure distributed schemes*. Note that a design scheme may be pure even though the physical layout of tables of the same database is different.

Mixed physical design in turn motivates the investigation of a *two-phased query optimization*: The coordination middleware chooses the nodes to evaluate a given query and derives subqueries, to be evaluated by these nodes. In the second phase, each node finds a good plan for its subquery and executes it. Two-phased query optimization is appealing for the following reasons: The coordinator load becomes less, compared to a setup where the coordinator is responsible for the entire query optimization process. This approach does not require extensive centralized statistics gathering, only some essential information is needed in the first phase.

Within this general framework, this paper is a first step to address the following specific questions.

1. How well can we parallelize a given query? How to predict the benefit of parallelization with little effort, i.e., by means of a brief inspection of the query?
2. How many cluster nodes (*parallelism degree*) should be used to evaluate a given query? What is the limit utility when increasing the number of nodes from n to n+1?

Suppose that a pure distributed physical design scheme is given. Question 1 asks if this scheme allows for faster evaluation of the query, as compared to evaluation on a single database. Question 2 in turn assumes that there is a distributed design scheme that allows to continuously adjust the number of nodes to evaluate the query. To address this question, our study assumes a particular design scheme with this characteristic as given. It is based on the TPC-R benchmark [4]. Answers to this question will allow us to come up with mixed physical design schemes.

The contribution of this paper is as follows: It provides a classification of queries where the different classes are associated with specific parallelization characteristics. In particular, 'parallelization characteristics' stands for the number of nodes that should be used to evaluate the query in parallel. We provide simple criteria to decide to which class a query belongs. Our experiments yield a characterization of the various classes in quantitative terms. The results will allow us to build a query optimizer at the coordination level for the first phase of optimization.

While our study clearly goes beyond existing examinations, it is also preliminary in various respects. First, the focus of this paper is on queries executed in isolation. We largely ignore that there typically is a stream of queries, and we leave the interdependencies that might arise by the presence of other queries to future work. Furthermore, in this paper we do not discuss updates. We plan to adapt results gained from parallel work in our project [21] at a later stage. We also do not address the issue of physical design systematically, but take for granted a meaningful, but intuitively chosen physical design scheme. Nevertheless, we see this study as a necessary and important step towards the infrastructure envisaged.

The remainder of this paper has the following structure: Section 2 describes the PowerDB architecture and briefly reviews the TPC-R benchmark which serves as a running example and experimental platform of this study. Section 3 discusses the physical database design that underlies this study and parallel query evaluation using this scheme, and presents our query classification scheme. Section 4 discusses our experiments. Section 5 presents related work. Section 6 concludes.

2 System Architecture and Preliminaries

Architecture. The object of our investigations in the PowerDB project is a cluster of databases, i.e., a set of off-the-shelf PCs, connected by a standard network, and each such PC runs a standard relational DBMS. Using relational databases on the cluster nodes gives us a relatively simple, easy-to-program, and well-known interface to the nodes. The overall objective of the project is to design and realize a middleware that orchestrates them (see Fig. 1). We have implemented a first version of the middleware that provides a subset of the features envisioned [2],[3]. The version envisioned will also decide which data the individual components store, and it will allow for more flexible query-processing schemes.

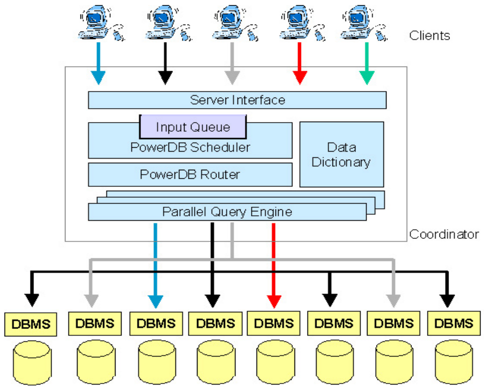


Fig. 1. System Architecture

In the PowerDB architecture, clients only communicate with a distinguished cluster node, the *coordinator*, which is responsible for *queueing*, *scheduling*, and *routing*. Incoming requests first go to the input queue. Depending on performance parameters, the scheduler decides on the order of processing of the requests and ensures transactional correctness. The router comes into play when there is a choice of nodes where a request may execute without violating correctness. To carry out its tasks, the coordinator must gather and maintain certain statistics. The *data dictionary* component contains such information. A given query may be executed on an arbitrary number of components in parallel.

Database. TPC-R is a benchmark for decision support that contains complex queries and concurrent updates. It will not only serve as a test bed for our experimental evaluation, but it will also be our running example. Fig. 2 plots the TPC-R database schema that consists of eight tables. Each table corresponds a rectangle that consists of three parts. The name of the table is at the top, the primary key attributes of the table are in the middle, and the number of table rows with scale factor 1 is at the bottom. The scale factor is an external parameter that determines the database size. Scale factor 1 results in a total database size of roughly 4 GB, including indexes. *LineItem* and *Orders* are the two biggest tables by far. Following the usual distinction in the data-warehousing context, these tables are the *fact tables* of the schema, the other ones the *dimension tables*.

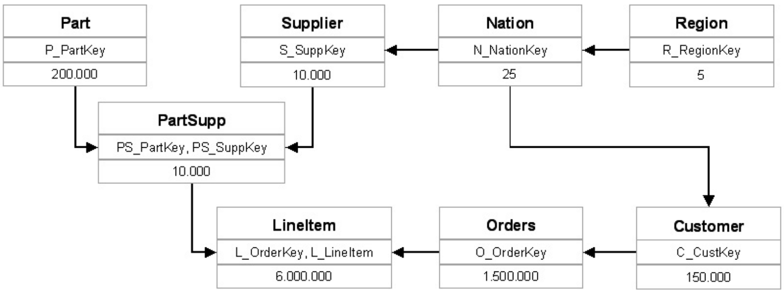


Fig. 2. TPC-R Database Schema

3 Query Evaluation in a Database Cluster

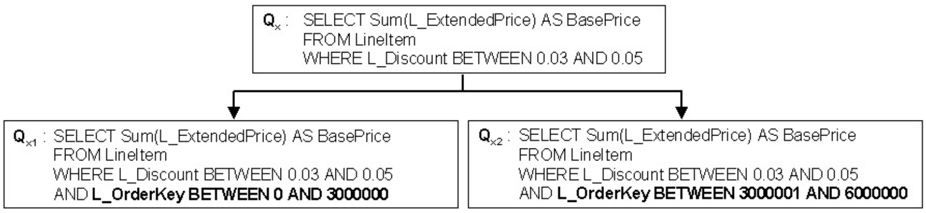
This section describes our investigation of the parallelization characteristics of OLAP queries. More specifically, we discuss the physical design that is the basis of our evaluation. We continue with a description of virtual and physical query partitioning and present our classification of possible queries.

Physical Design. An important requirement to ensure maximum impact of our work is that the middleware should be lightweight. In particular, we try to avoid any non-trivial query processing at the coordinator. That is, simple steps such as uniting disjoint sets or aggregating a set of values are acceptable, but joins or sorting are not.

A promising design scheme is *physical partitioning* [2]. It partitions the fact tables over different nodes, using the same attribute as partitioning criterion, and has replicas of the dimension tables at all nodes. Fact tables are by far the largest tables, so accessing these tables in parallel should yield the most significant speedup. However, the difficult open question remains: How many nodes should be used to evaluate the query in parallel? To be more flexible when it comes to experiments, we have not used physical partitioning, but the following physical design scheme: All tables are fully replicated, and there are clustered primary key indexes on the fact tables, on the same attribute. With TPC-R, this attribute is *OrderKey*[1]. Subsequently, we refer to this attribute as *partitioning attribute*. Having a partitioning attribute allows to 'virtually' partition the fact tables, i.e., generate subqueries such that each subquery addresses a different portion of these tables, and to ship these subqueries to different nodes. The result of the original query is identical to the union of the results of the subqueries in many cases (for a more precise description of the various cases see below). Because of the clustered index, a subquery indeed searches only a part of the fact tables. This is the motivation behind this design scheme, subsequently referred to as *virtual partitioning*.

Query evaluation with virtual partitioning. With virtual partitioning, a subquery is created by appending a range predicate to the where clause of the original query. The range predicate specifies an interval on the partitioning attribute. The interval bounds are denoted as *partition bounds*. The idea behind partitioning is that each subquery processes roughly the same amount of data. Assuming that the attribute range is known, and values are uniformly distributed, the computation of the partition bounds is trivial. The following example illustrates the virtual partitioning scheme based on the uniformity assumption.

Example 1. Fig. 3 depicts a query Q_x that accesses the fact table *LineItem*. It scans all tuples and summarizes the ones that satify the search condition. The figure also shows the partitioning of Q_x, which is decomposed into subqueries by adding *OrderKey* range predicates. The example assumes that *OrderKey* ranges from 0 to 6000000, and the number of subqueries is two. Assuming uniform distribution of tuples regarding the partitioning attribute, each subquery accesses one half of whole data.



```
Qx : SELECT Sum(L_ExtendedPrice) AS BasePrice
     FROM LineItem
     WHERE L_Discount BETWEEN 0.03 AND 0.05
```

```
Qx1 : SELECT Sum(L_ExtendedPrice) AS BasePrice
      FROM LineItem
      WHERE L_Discount BETWEEN 0.03 AND 0.05
      AND L_OrderKey BETWEEN 0 AND 3000000
```

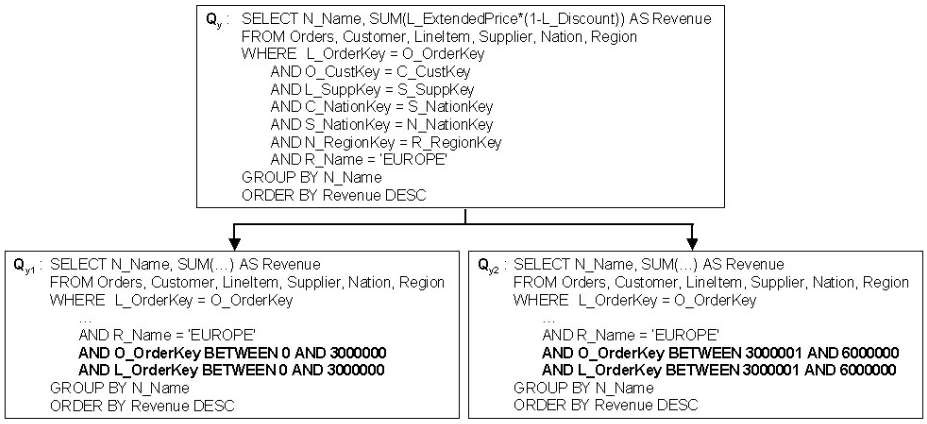
```
Qx2 : SELECT Sum(L_ExtendedPrice) AS BasePrice
      FROM LineItem
      WHERE L_Discount BETWEEN 0.03 AND 0.05
      AND L_OrderKey BETWEEN 3000001 AND 6000000
```

Fig. 3. Partitioning of Simple Queries

[1] Throughout this paper, we use the attribute name *L_OrderKey* or *O_OrderKey* when we want to make explicit the database table of the attribute (*LineItem* and *Orders*, respectively).

After having generated the subqueries, the coordinator ships them to appropriate cluster components. After their evaluation, the coordinator computes the overall result set, e.g., by merging results of subqueries.

Example 2. The query Q_y in Fig. 4 is a complex query containing aggregates and joins. It joins fact tables and other tables, and performs some aggregations, e.g., it groups on *N_Name* column and summarizes the column computed. Figure also shows how Q_y is partitioned. Note that the predicate on *OrderKey* occurs twice within each subquery, once for each fact table. This is to give a hint to the query optimizer. From a logical point of view, it would be enough to append the range for one of the fact tables only. But the current version of SQL Server only works as intended if the range is made explicit for each fact table.



```
Qy :  SELECT N_Name, SUM(L_ExtendedPrice*(1-L_Discount)) AS Revenue
      FROM Orders, Customer, LineItem, Supplier, Nation, Region
      WHERE  L_OrderKey = O_OrderKey
          AND O_CustKey = C_CustKey
          AND L_SuppKey = S_SuppKey
          AND C_NationKey = S_NationKey
          AND S_NationKey = N_NationKey
          AND N_RegionKey = R_RegionKey
          AND R_Name = 'EUROPE'
      GROUP BY N_Name
      ORDER BY Revenue DESC
```

```
Qy1 : SELECT N_Name, SUM(…) AS Revenue
      FROM Orders, Customer, LineItem, Supplier, Nation, Region
      WHERE  L_OrderKey = O_OrderKey
          …
          AND R_Name = 'EUROPE'
          AND O_OrderKey BETWEEN 0 AND 3000000
          AND L_OrderKey BETWEEN 0 AND 3000000
      GROUP BY N_Name
      ORDER BY Revenue DESC
```

```
Qy2 : SELECT N_Name, SUM(…) AS Revenue
      FROM Orders, Customer, LineItem, Supplier, Nation, Region
      WHERE  L_OrderKey = O_OrderKey
          …
          AND R_Name = 'EUROPE'
          AND O_OrderKey BETWEEN 3000001 AND 6000000
          AND L_OrderKey BETWEEN 3000001 AND 6000000
      GROUP BY N_Name
      ORDER BY Revenue DESC
```

Fig. 4. Partitioning of Complex Queries

Determining the partition bounds. A crucial issue in partitioning is to determine the partition bounds such that the duration of parallel subqueries is approximately equal. The computation of good partition bounds is difficult in the following cases:

- The uniformity assumption does not hold, i.e., the values of the partitioning attribute are not uniformly distributed.
- The original query contains a predicate on the partitioning attribute, e.g., '*OrderKey < 1000*'. Another issue is that it is not clear whether intra-query parallelization is helpful at all in the presence of such predicates and will lead to significantly better query-execution times.
- Even if the uniformity assumption holds, it may still be difficult to compute good partition bounds. Consider the query Q_z that accesses several tables connected by an equi-join, and that has a selection predicate on the *Nation* table:

```
Qz: SELECT C_Name, SUM(O_TotalPrice)
    FROM Orders, Customer, Nation
    WHERE O_CustKey = C_CustKey AND N_Name = 'TURKEY'
          AND C_NationKey = N_NationKey
```

```
GROUP BY C_Name
ORDER BY C_Name
```

Let us further assume that there is a strong correlation between the attribute *N_NationKey* and the attribute *O_OrderKey*, e.g. certain *OrderKey* ranges for each nation. This means that the computation of the partition bounds must reflect both the selection predicate on the table *Nation* as well as the correlation. One solution to this problem is to use two-dimensional histograms.

With the TPC-R benchmark, be it the data set, be it the queries, we have not experienced any of these effects. That is, we could conduct our study without providing solutions to these problems. However, taking these issues into account is necessary to cover the general case and is part of our future work.

Classification of Queries. Even if we can leave aside the restrictions from above, the core question is still open, i.e., how many nodes should be used to evaluate a query. We therefore try to classify the set of possible queries into a small number of classes by means of their structural characteristics, such that queries within the same class should have more or less the same parallelization characteristics. For each class, we want to find out how many nodes should be used to evaluate a query. The first phase of query optimization would inspect an incoming query and identify its class. We distinguish three classes of queries:

1. Queries without subqueries that refer to a fact table,
    a. Queries that contain exactly one reference to one of the fact tables, and nothing else.
    b. Queries that contain one reference to one of the fact tables, and together with arbitrarily many references to any of the other tables.
    c. Queries that contain references to both fact tables, all joins are equi-joins, and the query graph is cycle-free.
2. Queries with a subquery that are equivalent to a query that falls into Class 1.
3. Any other queries.

The difference between Class 1.a and 1.b is that a query of Class 1.a accesses only one table, a query of Class 1.b in turn accesses the fact table and one or several dimension tables. With Class 1.c, the query graph must be cycle-free to prevent from the following situation: Think of two tuples in the fact table, and think of a query whose query graph contains a cycle. The query now combines these two tuples by means of a join, directly or indirectly. Now assume that the fact table is partitioned, and the two tuples fall into different partitions. If the query is now evaluated in the way described above, the two tuples will not be combined. In other words, partitioning would lead to a query result that is not correct. For the analogous reason, we require that the join in Case 1.c is an equi-join. Consider the following query that does not fall into Class 1.c, because the join is not an equi-join:

```
SELECT O_OrderKey, Count(L_Quantity)
FROM Orders, LineItem
WHERE O_OrderDate = L_CommitDate
GROUP BY O_OrderKey
```

If a tuple-pair of *Orders* and *LineItem* that meets the join predicate falls into different partitions, then the evaluation of the query on the different partitions in parallel would return an incorrect result.

Some queries may contain subqueries that work on fact tables. The query processor handles nested queries by transforming them into joins. So, extra joins that contain fact tables have to be performed. This sort of queries can be parallelized as long as their inner and outer parts access to same partition of data. This requires that inner and outer blocks of query must be correlated on partitioning attribute. In another words, transformed join must be done on partitioning attribute. The query Q_t meets this condition. So, same *OrderKey* ranges can be applied to inner and outer blocks.

Q_t:

```
SELECT O_OrderPriority, COUNT(*) AS Order_Count
FROM Orders
WHERE O_OrderDate >= '01-FEB-1995'
      AND O_OrderDate < dateadd(month,3,'01-FEB-1995')
      AND EXISTS(SELECT *
                 FROM LineItem
                 WHERE L_OrderKey = O_OrderKey
                       AND L_CommitDate < L_ReceiptDate)
GROUP BY O_OrderPriority
ORDER BY O_OrderPriority
```

It is not feasible that the subquery accesses a partition of the fact table that is different from the one accessed by the outer query. The inner block of the query Q_u has to access all tuples of *LineItem*. So, the same partitioning ranges cannot be applied to inner and outer parts. Hence, this query cannot be parallelized.

Q_u:

```
SELECT SUM(L_ExtendedPrice)/(7.0) AS Avg_Yearly
FROM LineItem L, Part
WHERE L.L_PartKey = P_PartKey AND P_Brand = 'Brand#55'
      AND L.L_Quantity < (SELECT AVG(L_Quantity)*(0.2)
                         FROM LineItem
                         WHERE L_PartKey = P_PartKey)
```

Query Q_x in Example 1 falls clearly into Class 1.a. Since the query Q_z seen in previous subsection refers to only one of the fact tables and includes some other tables than fact tables falls into Class 1.b. Query Q_y in Example 2 joins both fact tables and falls into Class 1.c. Query Q_t above falls into Class 2. Because it has a subquery working on fact table and its query graph is cycle-free.

We expect a linear speedup for queries from Class 1.a and a roughly linear speedup for queries from Class 1.b and Class 1.c. The reason why we are less certain about Class 1.b and Class 1.c queries is that they access fact tables only partly, while accessing the other tables entirely. Nevertheless, we expect a significant speedup since the partially accessed tables are the large ones. However, the overall picture is less crisp, compared to Class 1.a. The parallelization characteristics of the individual queries in Classes 1.b and 1.c are not clear apriori.

Another issue that needs to be addressed is aggregation. The issue of computing aggregates in parallel has been investigated elsewhere [22], and it is largely orthogonal to this current work. Our study just assumes that such parallelization techniques are available, and the effort required by the coordination middleware to compute the overall aggregates, given the aggregate results from the subqueries, is negligible.

Let us now briefly look at Class 2 queries. Again, we can make use of previous work that has shown how to transform such queries into ones that do not have subqueries [19]. These transformations cover the most cases. This is actually sufficient for our purposes: Since we envision a mixed physical design scheme where at least one node holds a copy of the entire database, it is not necessary to parallelize all queries, and it is perfectly natural to leave aside certain queries, as we do by defining Class 3.

4 Experiments

Having come up with classes of queries with presumably similar parallelization characteristics, it is now interesting to evaluate this classification experimentally. We have looked at different partition sizes to investigate the effect of a growing cluster size. Our experimental evaluation consists of two steps. First, we study queries and their parallelization characteristics in isolation. Although our focus is on this issue, the second part of experiments looks at query streams and compares the throughput of a stream of partitioned queries to one of non-partitioned queries.

Experimental Setup. In our experiments, the cluster size ranged from 1 to 64 PCs. Each cluster node was a standard PC with Pentium III 1GHz CPU and 256 MBytes main memory. We have used MS SQL Server 2000 as the component database system, running on MS Windows 2000. Each component database was generated according to the specification of the TPC-R benchmark with a scaling factor 1. In other words, the TPC-R database was fully replicated on each component. When looking at the parallelization characteristics of an individual query, we ran the query repeatedly, typically around 50 times, and our charts graph the mean execution time.

Outcome and Discussion of Experiments. In the following, we look at the different query classes from the previous section successively. The TPC-R benchmark contains two queries that fall into Class 1.a. Fig. 5(a) shows the mean execution times. The different columns stand for different cluster sizes (1, 2, 4, etc.). Since these queries do not contain joins between large tables, we would expect a linear speedup. Query 1 scales perfectly with the cluster size. The speedup of Query 6 is more or less the same for clusters of size 2 and 4. However, the mean execution time sharply decreases with 8 cluster nodes. Our explanation is that all the data accessed by a subquery fits into the database cache, and further executions of the same query exploit this effect. On the other hand, this is also the case for Query 1. However, it computes more aggregates, which requires more CPU time. This explains why the effect observed with Query 6 does not make itself felt in this case as well.

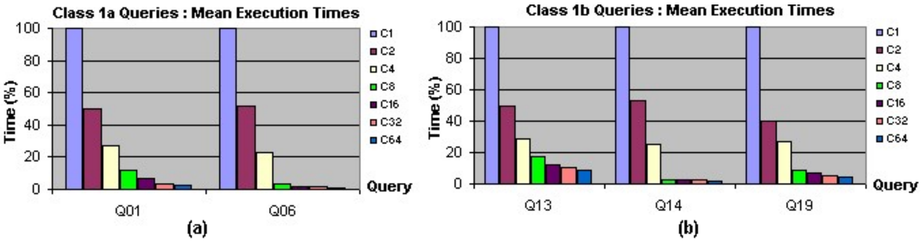


Fig. 5. Mean Execution Times for **(a)** Class 1.a and (b**)** Class 1.b Queries

Fig. 5(b) graphs the mean execution times for Class 1.b queries. Three queries from the benchmark fall into this class. They are more complex than the ones of Class 1.a. However, we can still expect a significant and hopefully almost linear speedup if we can equally divide the load among the cluster nodes. Our results are more or less as expected with a degree of intra-query parallelism of 2 and 4. Query 13 contains a substring match predicate (e.g., *o_comment NOT LIKE '%special%'*) as the only predicate. Since substring matching is very CPU-intensive and time-consuming, the decrease of execution time with cluster size 8 is not exactly sharp with Query 13. Query 19 in turn contains many search arguments and consequently requires much CPU time as well. Its behavior is similar to the one of Query 13.

Class 1.c is the largest class in our classification, with seven members. Fig. 6 shows the mean execution times for Class 1.c queries. The results up to 8 cluster nodes are now as expected, given the observed characteristics of queries from the other classes. The effect with substring predicates occurs again with Query 9.

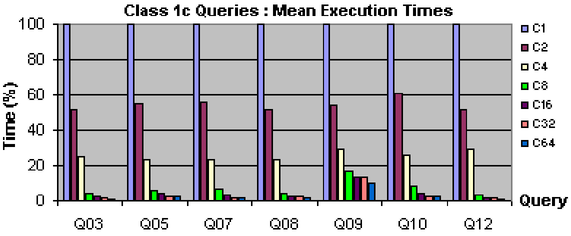


Fig. 6. Class 1.c Queries: Mean Execution Times

For each nested query in Class 2, there is one equivalent flat query that falls into Class 1. Since the queries in this class behave like Class 1 queries, we do not explicitly discuss them. As said before, Class 3 queries represent the remaining kinds of queries that are not considered for parallelization.

Summing up this series of experiments, there are two factors that significantly affect query-execution times: The first one is whether or not data accessed by a subquery fits into memory. With our target database, this is the case with 8 cluster nodes and above. The second one is the nature of the operations, e.g. comparisons such as substring predicates.

As we have seen, after having reached a certain degree of parallelism, a further increase does not reduce the execution times significantly. Consequently, the second part of our experiments looks at the throughput of partitioned vs. non-partitioned queries in the cluster. With non-partitioned queries, there is no intra-query parallelism, only inter-query parallelism. In other words, each query from the stream goes to a different cluster node, one query per node at a time [2]. We used simple *round robin* to assign queries (*QP1*) to the cluster nodes. In the partitioned case in turn, the entire cluster is busy evaluating subqueries of the same query at a certain point of time, and the cluster size is equal to the partition size (*QPN*).

Fig. 7 graphs the throughput for both cases. The *x*-axis is the cluster size with non-partitioned queries and the partition size in the other case. The throughput values are almost the same for cluster sizes/partition sizes 2 and 4. The throughput with partitioned queries is 35% higher on average than in the other case because of the caching effect for cluster sizes/partition sizes 8 and 16. But after that, the caching benefit becomes less, and the throughput of partitioned queries is less. We conclude that the coordination middleware should form groups of nodes of a certain size, 8 or 16 in our example. These node groups should evaluate a given query in parallel; a higher degree of intra-query parallelism is not advisable.

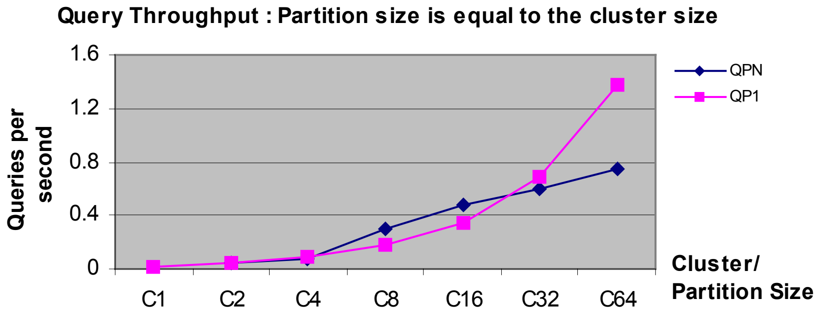


Fig. 7. Query Throughput – Partition size is equal to the cluster size

Having said this, we want to find out in more detail what the right partition size is, e.g., 16-partitioned queries vs. 8-partitioned queries. Fig. 8 shows the throughput with a fixed partition size, but an increasing cluster size. According to the figure, 8-partitioned queries yield the highest throughput for all cluster sizes equal to or larger than 8. This experiment is another view on our result stated earlier. If the partitions become so small that the data accessed by a subquery fits into main memory, it is not advantageous to increase the number of partitions beyond that point.

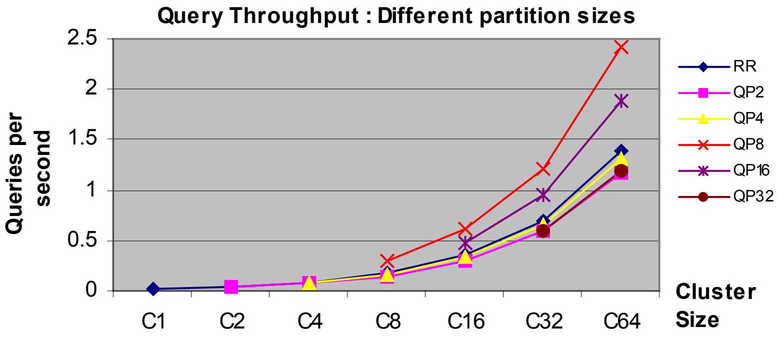


Fig. 8. Query Throughput - Different partition sizes

5 Related Work

Parallel databases have received much attention, both in academia and in industry. Several prototype systems have been developed within research projects. These include both shared nothing [12],[13] and shared everything systems [14]. Most prototypical parallel database systems are shared nothing. They deploy partitioning strategies to distribute database tables across multiple processing nodes. Midas [10] and Volcano [15] are prototypes that implement parallel database systems by extending a sequential one.

Industry has picked up many research ideas on parallel database systems. Today, all major database vendors provide parallel database solutions [6],[7],[8].

In the past, many research projects have assumed that parallel database systems would run on a special hardware. However, today's processors are cheap and powerful, and multiprocessor systems provide us with a good price/performance ratio, as compared to their mainframe counterparts. Hence, using a network of small, off-the-shelf commodity processors is attractive [5],[11]. In [16] and [17], relational queries are evaluated in parallel on a network of workstations. In contrast to our work, these prototypes do not use an available DBMS as a black box component on the cluster nodes.

Our work also goes beyond mechanisms provided by current commercial database systems, e.g., distributed partitioned views with Microsoft SQL Server [9]. These mechanisms allow to partition tables over several components and to evaluate queries posed against virtual views that make such distribution transparent. We did not rely on these mechanisms because our PowerDB project has other facets where they do not suffice. Microsoft's distributed partitioned views do not offer parallel query decomposition. Furthermore, distributed partitioned views are designed for OLTP and SQL statements that access a small set of data, as compared to OLAP queries [18].

Our study in turn investigates which queries can be evaluated in parallel over several partitions, and yield a significant speedup. We provide performance characteristics of OLAP queries when evaluated in parallel, together with extensive experiments.

6 Conclusions

Database clusters are becoming a commodity, but it is still unclear how to process large workloads on such a cluster. The objective of the project PowerDB is to answer this question, and we have carried out some preliminary experiments in this direction described in this paper. This study has looked at a typical OLAP setup with fact tables and dimension tables and complex, long-lasting queries. We have asked how many nodes should be used to evaluate such a query in parallel (intra-query parallelism). To this end, we have used a physical design scheme that allows varying the number of nodes for parallel query evaluation. Our contribution is a classification of OLAP queries, such that the various classes should have specific parallelization characteristics, together with an evaluation and discussion of results.

The work described in this paper is part of a larger effort to realize the PowerDB infrastructure envisioned. In future work, we will address the following issues:

- A more technical, short-term issue is to replace virtual partitioning by physical partitioning. Physical partitioning is more appropriate than virtual partitioning in the presence of updates, since full replication tends to be very expensive.
- It is necessary to deal with situations where easy computation of partitioning bounds is not feasible (cf. Section 3). We conjecture that it will boil down to more detailed meta-data and statistics gathering by the coordinator, capturing the states of the nodes and the distribution of data.
- This paper has focused on queries in isolation, but future work should also look at streams of queries and possible interdependencies between the executions of different queries. Our own previous [3] work has dealt with caching effects in a database cluster and has shown that they should be taken into account to obtain better performance in simpler cases already (no intra-query parallelism).
- It is crucial to deal with other databases and application scenarios as well, not only OLAP. For instance, the case where the database contains a large table, and a large number of queries contain self-joins, is particularly interesting. It is not clear at all how to arrive at a good partition of such a data set. A possible approach might use data mining and clustering techniques – but this is purely speculative since we do not see at this level of analysis what the underlying distance metric should be.
- Given the database and the access pattern, the coordinator should identify a good mixed physical design scheme dynamically. This is a particularly challenging problem, given that self-configuration and automated materialized-view design is hardly solved in the centralized case. Of course, the access pattern does not only contain queries, but also updates. Finally, there is some recent work on physical representation of data that allows for fast, approximate query results in the OLAP context. The drawback of these schemes is that updates tend to be expensive. We

will investigate the self-configuration problem in the distributed case, taking such design alternatives into account as well.

References

1. Özsu T., Valduriez P., *Distributed and Parallel Database Systems*. ACM Computing Surveys, 28(1):125-128, March 1996.
2. Röhm U., Böhm K., Schek H.-J., *OLAP Query Routing and Physical Design in a Database Cluster*. Advances in Database Technology, In Proc. 7th Int. Conf. on Extending Database Technology, EDBT 2000, pp. 254-268, March 2000.
3. Röhm U., Böhm K., Schek H.-J., *Cache-Aware Query Routing in a Cluster of Databases*. In Proc. 17th Int. Conf. on Data Engineering, ICDE2001, April 2001.
4. TPC, TPC Benchmark^{TM} R (Decision Support).
5. Kossmann D., *The State of the Art in Distributed Query Processing*. ACM Computing Surveys, 32(4) : 422-469, September 2000.
6. Baru C.K., et al, *DB2 Parallel Edition*. IBM System Journal, 34(2):292-322, 1995.
7. *Oracle 8i Parallel Server*. An Oracle Technical White Paper. January 20, 2000.
8. *Informix Extended Parallel Server 8.3 XPS*. White Paper, Informix, 1999.
9. Delaney K., *Inside Microsoft SQL Server 2000*. Microsoft Press, 2001.
10. Bozas, G., Jaedicke, Mitschang, B., Reiser, A. Zimmermann, S., *On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project*. TUM-I 9625, SFB-Bericht Nr. 342/14/96 A, May 1996.
11. DeWitt D.J., Gray J., *Parallel Database Systems: The Future of High Performance Database Systems*. Communications of the ACM, 35(6):85-98, June 1992.
12. DeWitt, D.J., et al., *The Gamma Database Machine Project*. IEEE Transactions on Knowledge and Data Engineering, 2(1):44-62, March 1990.
13. Boral H., et. al., *Prototyping Bubba*, *A Highly Parallel Database System*. IEEE Transactions on Knowledge and Data Engineering, 2(1):4-24, March 1990.
14. Stonebraker M., et. al., *The Design of XPRS*. In Proc. 14th Int. Conf. on Very Large Data Bases, pp. 318-330, September 1988.
15. Graefe G., *Volcano - An Extensible and Parallel Query Evaluation System*. IEEE Transactions on Knowledge and Data Engineering, 6(1):120-135, February 1994.
16. Exbrayat M., Brunie L., *A PC-NOW based parallel extension for a sequential DBSM*. IPDPS 2000, Cancun, Mexique, 2000.
17. Tamura T., Oguchi M., Kitsuregawa M., *Parallel Database Processing on a 100 node PC Cluster : Cases for Decision Support Query Processing and Data Mining*. In Proc. SC97: High Performance Networking and Computing, 1997.
18. *Microsoft SQL Server MegaServers : Achieving Software Scale-Out*. White Paper, Microsoft Corporation, February 2000.
19. Ganski R.A., Long H.K.T. *Optimization of Nested SQL Queries Revisited*. In Proc. ACM Conf. on Management of Data, pp. 23-33, 1987.
20. The Project PowerDB, url : http://www.dbs.ethz.ch/~powerdb.
21. Röhm U., Böhm K., Schek, H.-J., Schuldt H., *FAS – a Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components*. In Proc. Int. Conf. on Very Large Data Bases, 2002.
22. Shatdal A., Naughton J.F., *Adaptive Parallel Aggregation Algorithms*. In Proc. ACM Conf. on Management of Data, pp. 104-114, 1995.