OLAP Query Routing and Physical Design in a Database Cluster

Uwe Röhm, Klemens Böhm, and Hans-J. Schek

Database Research Group, Institute of Information Systems
ETH Zentrum, 8092 Zurich, Switzerland
{roehm|boehm|schek}@inf.ethz.ch

Abstract. This article quantifies the benefit from simple data organization schemes and elementary query routing techniques for the PowerDB engine, a system that coordinates a cluster of databases. We report on evaluations for a specific scenario: the workload contains OLAP queries, OLTP queries, and simple updates, borrowed from the TPC-R benchmark. We investigate affinity of OLAP queries and different routing strategies for such queries. We then compare two simple data placement schemes, namely full replication and a hybrid one combining partial replication with partitioning. We run different experiments with queries only, with updates only, and with queries concurrently to simple updates. It turns out that hybrid is superior to full replication, even without updates. Our overall conclusion is that coordinator-based routing has good scaleup properties for scenarios with complex analysis queries.

1 Introduction

Practically any organization has accumulated large amounts of data. Increasingly, such data is becoming subject to complex analysis queries, also referred to as decision support queries or On-Line Analytical Processing queries (OLAP queries). Despite their complexity, users want those queries to be evaluated fast. Typically, materialized views and data warehousing technology are in use to meet this requirement. However, users more and more expect those queries to be evaluated on up-to-date data. View materializations help if updates and their propagation to materialized views do not delay the evaluation of queries. The main objective of our work within the PowerDB project at ETH Zurich is to reduce the slowdown of queries caused by updates. I.e., our vision of a 'next-generation OLAP platform' is that of a system allowing for efficient evaluation of short queries, updates, and complex analysis queries, without being in the way of each other.

The PowerDB architecture is a cluster of databases, i.e., a set of so-called *components* that run a DBMS, together with a coordinator. The approach pursued within the project is to build a simple yet clever coordinator: it can route queries to the least loaded component in case of replication, it can decompose and route complex queries in parallel to several components in case of partitioning and/or replication, and it can route updates to replica in parallel.

This current article investigates the capacities of routing in OLAP scenarios. We consider queries and simple concurrent updates that the component databases handle correctly without a global scheduler. We investigate admission control and routing techniques for complex analysis queries for different data placement schemes and under different update rates. We consider the case that there are streams of OLAP queries, OLTP queries, and simple updates, and we are interested in response times as well as throughput. Our objective is to assess simple techniques in quantitative terms, in order to have results that are sufficiently general.

In more detail, we proceed as follows: the first elementary but important question is under which circumstances queries should be evaluated concurrently, and when they better run after each other. Literature has coined the term *affinity* for two queries whose concurrent evaluation is better [YCDI87]. In the context of OLAP queries, it turns out that the number of cases where affinity occurs is limited. Furthermore, we have observed a phenomenon which we refer to as *obstruction*, i.e., two queries running concurrently execute much slower as if they run one after the other. In the body of the article, we say in which cases obstruction occurs.

Our next step is to evaluate simple data placement schemes and routing techniques for complex analysis queries. The alternative placement schemes considered are the following ones: the first one is full replication. The simplest routing technique is that the coordinator sends queries to the components one-at-a-time in a round-robin fashion. The second alternative called *hybrid* is a combination of replication and partitioning, i.e., we partition the biggest relation and replicate the other ones. Queries from the input stream that refer to the partitioned relation are evaluated one after the other. The rationale behind the second alternative is that most databases contain one relation that is significantly larger than the other ones. With hybrid, we expect a high degree of intra-query parallelism. The first alternative in turn leads to inter-query parallelism. The central question now is under which circumstances the speedup from intra-query parallelism is higher than the throughput improvement from inter-query parallelism. Our first result on affinity has told us that throughput increases linearly with full replication in a readonly environment (as the relative overhead is negligible), and our experiments confirm this. However, it turns out that hybrid is significantly better, e.g., with six nodes its throughput is a factor of 3 higher.

Finally, while complex analysis queries have been the motivation for this study, the more general scenario is that the transaction mix contains updates as well. As mentioned before, this current work considers a special case with regard to updates, namely transactions that contain one single update action. This restricted case gives us an idea regarding the influence of updates on query performance; at the same time, it is the most general case without explicitly taking global correctness into account. Our expectation is that hybrid is also better with updates, and our experiments confirm this. This is because the big relation also tends to be the one that is updated most frequently. The slowdown of the queries due to the kind of updates considered here is relatively moderate

in both cases. For example, the throughput of hybrid decreases by 3% with six nodes and ten updates per second.

While related work has investigated affinity [YCDI87], much of this work has not taken OLAP queries into account. Furthermore, much empirical work on physical design has focused on partitioning without replication [JLS99,MD97], and we have not found any conclusive answers to the important questions addressed above. To investigate these questions, we are building a full system, the PowerDB engine. For our evaluation, we have used data and queries from the TPC-R benchmark [TPC99], which targets at platforms for OLAP.

This current work concentrates on query admission and routing issues and leaves aside scheduling. Arbitrary complex transactions with the need for a global scheduler are subject to future work. However, our own previous work has shown that scheduling at the coordinator level with parallelization yields good scalability [GBS99,KS96,RNS96]. — Furthermore, it should be clear that this article does not address query processing; instead, we rely on the query processing capabilities of off-the-shelf database technology.

The remainder of this article has the following structure: Section 2 gives an overview of our PowerDB architecture. In Section 3.1, we describe our physical design alternatives in more detail, followed by the corresponding query evaluation and routing strategies in Section 3.2. Section 4 contains our affinity study and the results of our performance evaluation of the two alternatives, full replication and hybrid design.

2 PowerDB System Architecture

2.1 Overview

The parallel architecture investigated in this current work is subject of a larger project of the Database Research Group at ETH Zurich called PowerDB. In

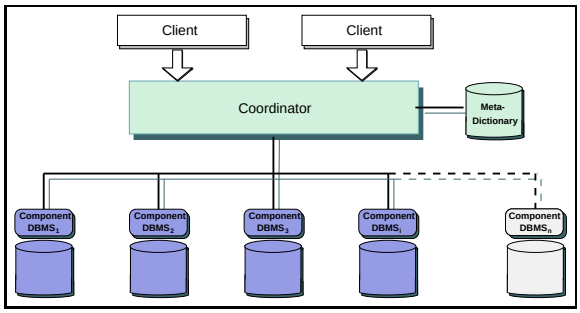


Fig. 1. System Architecture of PowerDB.

short, PowerDB is a "database of databases". It uses relational databases for storage management. Figure 1 serves as an illustration. There is a distinguished node, the *coordinator*, and a number of other nodes, the *components*. Clients

only interact with the coordinator. From the client point of view, there is one database schema. The coordinator passes updates and queries to the components and collects the results. In more detail, the coordinator parallelizes, schedules and routes the queries, depending on the actual physical design. In case of updates, the coordinator is responsible for consistency of the data. A scheduler (cf. Figure 2) decides in which order to execute the queries. If several nodes can evaluate a query, a router chooses the target node. The coordinator has query-processing capabilities and is capable of evaluating several queries in parallel.

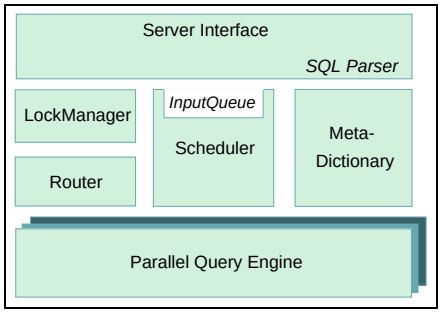


Fig. 2. Internal Structure of PowerDB.

Summing up, the coordinator is a database management system on top of database systems in that it consists of a SQL frontend, a metadictionary, a scheduler, a lock manager, a router and an execution engine (cf. Figure 2). We use the terms 'coordinator' and 'PowerDB engine' interchangeably. The hardware platform is a cluster of PCs; each PC runs an off-the-shelf relational DBMS. For the evaluation, the PowerDB cluster consists of up-to 6 Pentium-II computers (266 MHz), each equipped with 128 MBytes RAM and two 4 GBytes harddisks. We have used Oracle 8.0.4 as component DBMS on all PCs. A 100 MBit Ethernet LAN connects the nodes. The coordinator runs on a separate PC.

2.2 PowerDB Routing

This subsection describes the routing module and its interaction with other modules at the coordinator level in more detail. Clients communicate with PowerDB through a message-based interface. More precisely, clients issue SQL92-compliant statements. After parsing and validating an SQL statement, the SQL frontend generates a simple execution plan (see Example 1). The objective is to delegate the evaluation of the statement as much as possible to the component DBMSs. The execution plans do not specify the components that evaluate the plan. This is feasible since we will consider only simple placement alternatives that do not distinguish between components. So far, no further query optimization takes place. The SQL frontend then inserts the action into the scheduling queue.

To facilitate that the coordinator can accept new client requests while processing other ones, the PowerDB coordinator is multithreaded. The input queue is processed periodically, or as soon as it has a certain length. The scheduler

determines a set of candidate components that can execute the current action from a correctness point of view. The router then selects the actual components from among the candidates. In the context of this current evaluation, we consider only simple query transactions consisting of a single SQL `SELECT`-statement and simple updates. Consequently, the scheduler does not impose any restrictions.

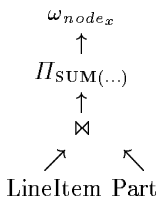
The execution engine is capable of executing multiple actions concurrently at different components. The main query execution strategy is query shipping: the component DBMSs are responsible for evaluating an SQL statement locally. The query engine at the coordinator carries out any query processing functionality necessary to combine partial results of a query accessing several nodes, e.g., set unions, or aggregate computation. The results are sent directly to the callback interface of the client divided into chunks of a specific size.

Example 1: Query 14 from the TPC-R benchmark looks as follows:

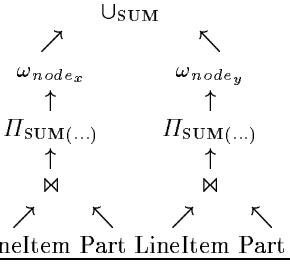
```
SELECT   SUM(...)
FROM     LineItem, Part
WHERE    L_PartKey = P_PartKey
         AND L_ShipDate BETWEEN '01-Mar-95' AND '01-Apr-95'
```

Plan A is the execution plan for full replication. ω is the wrap-operator known from middleware infrastructures for data integration [RB99]. The target node which shall execute the subtree with root ω_{node_x} is left unspecified. Now consider a data placement alternative that partitions relation `LineItem` over all components, and the components hold full replica of the other relations. With this design, the generated plan depends on the number of nodes. For instance, Plan B is for two nodes. It specifies that different nodes must execute the subqueries, and that the PowerDB query engine has to combine the partial results. Both with Plan A and Plan B, the router will fill in the placeholders for the target nodes.

Execution Plan A



Execution Plan B



---

3 Physical Design and Routing of Complex Analysis Queries in a Database Cluster

3.1 Physical Design

Primitives for Physical Design. A fundamental problem in this context is the physical organization of data that yields good performance both with regard to

queries and updates. The two primitives for physical design of individual relations that are specific to distribution are partitioning and replication [OV91]. *Partitioning*, i.e., data from a relation goes to different nodes, typically results in intra-query parallelism. *Replication* in turn leads to inter-query parallelism, as different nodes can evaluate queries in parallel. With replication, query-processing functionality at the coordinator is not necessary. But partitioning in general requires such functionality, at least if data is not shipped from one component to another.

Basic Alternatives Considered in this Study. In the following, we describe the basic alternatives for physical design that we consider in this context. Recall that we want to quantify the benefit from simple, elementary techniques for complex analysis queries. While literature has proposed other schemes for physical organization of databases as a whole, e.g., collocated joins [BFGe95] or multi-attribute declustering [GDQ92], we see such techniques as refinements of the basic alternatives described in the following.

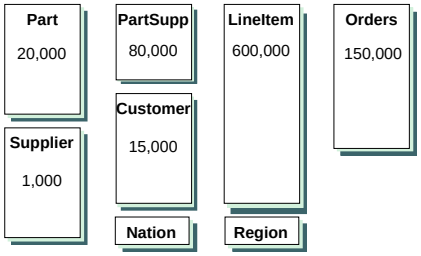


Fig. 3. TPC-R with Full Replication.

Full Replication. Our first basic alternative is full replication, i.e., each component contains a copy of each node. The coordinator does not process queries submitted by clients in any way; it routes the query to a component of its choice, and the component returns the result to the client. In Section 3.2, we discuss the different routing techniques considered. Updates are processed in an eager, "update everywhere" fashion — Figure 3 serves as an illustration. The boxes correspond to the relations from the TPC-R database, the numbers are the numbers of tuples in the relation, and the different shades tell us which components hold a copy of the relation.

Hybrid Design. The second alternative, subsequently called *hybrid design*, is as follows: it partitions the biggest relation over all n nodes, and each node holds a copy of all the other relations (cf. Figure 4). With regard to query evaluation, there is a distinction between queries that refer to the partitioned relation and those that do not. In the first case, all components process the original query, and the coordinator computes the overall result. If the query does not contain aggregation, the overall result is simply the disjoint union of the intermediate results; computing the overall result in the other case

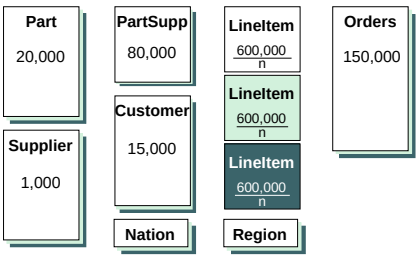


Fig. 4. TPC-R with Hybrid Design.

is not difficult as well. If the query does not refer to the partitioned relation, their evaluation is as with full replication.

The rationale behind the second alternative is twofold: first, related work on physical organization of individual relations shows that partitioning is of advantage as long as partitions do not become too small [MD97]. In other words, we should partition the largest relation to obtain the optimal speedup. The other motivation is the heuristic that big relations are subject to frequent updates. With hybrid, each update of the big relation goes to only one component, as opposed to n components in the case of full replication.

3.2 Query Routing

The previous section has described the alternatives of physical design used in this study. Both approaches replicate at least some relations over all nodes. Hence, there are several components in the general case that can evaluate a query. To cope with this situation, different query routing algorithms are available [CLL85,YCDI87,Rah92,FGND93]. In the following, we describe the routing schemes used in this study.

Balancing-Query-Number Routing. A common approach to query routing in a database cluster is round robin. We use the variant that tries to have the same number of active queries at each node. Figure 5 contains the algorithm. It routes a newly arrived query to the component with the lowest number of active queries. By using a stable sort algorithm it considers components in a round-robin way. [CLL85] calls this routing strategy *BalanceQNr* ('Balance-Query-Number').

```
function BalanceQNr_Routing ( NodeList nodes, Action new_query ) : Node
begin
   // sort nodes by ascending load; if two nodes have the same number of queries,
   // the stable sort algorithm keeps them in the given order
   stable_sort( nodes, NumberOfQueriesComparison );
   return nodes[0]; // choose node with least load
end
```

Fig. 5. Balancing-Query-Number Routing Algorithm.

Affinity-Based Query Routing. The idea behind affinity-based routing is to assign queries which access the same data to the same component [YCDI87,FGND93]. The rationale is that affinity-based routing yields an increased buffer-hit ratio and thus reduces I/O costs. In the following, we briefly look at existing affinity-based routing schemes and then describe a new algorithm targeted at our specific scenario.

[YCDI87,FGND93] are simulation studies on affinity-based routing for OLTP scenarios. To decide if there is affinity between two queries, they only consider the data accessed by the queries, but not the nature of the access, i.e., 'scan'

vs. 'random access'. But this differentiation is important. For example, consider two queries which do an aggregation over a relation. As they access exactly the same data, a simple affinity-based routing algorithm would assign them to the same node. However, assume that evaluation of the queries is scan-based. In order to avoid flushing the database buffer, current database systems do not use the normal database buffer for pages read during a scan [Ora97]. The effect is that two queries with scan-based evaluation do not benefit from each other. Even worse, scanning the same disk may lead to obstruction[1].

```
function AffinityBased_Routing ( NodeList nodes, Action newquery ) : Node
begin
  stable_sort( nodes, NumberOfQueriesComparison );
  foreach node in nodes do
    boolean affinity := true;
    foreach activequery in node.activeQueries do
      affinity := affinity and (AffinityMatrix[newquery][activequery]>0);
    od
    if ( affinity = true ) then
       return node;
  od
  return NULL;
end
```

Fig. 6. Affinity-Based Routing Algorithm.

Consequently, the nature of the data access matters. Our improved affinity-based routing strategy reflects this. Figure 6 contains it in pseudo-code. The core of the algorithm is the *affinity matrix*, which states how much the concurrent execution of two queries is faster than their sequential evaluation. The values incorporate both data affinity and nature of access. The algorithm routes a newly arrived query to a component whose active queries have affinity with the new one. We will present the matrix in Subsection 4.1.

"Short Queries ASAP". Our affinity investigations will motivate another routing algorithm called *Short-Queries-As-Soon-As-Possible*, which we will describe at the end of Subsection 4.1.

4 Performance Evaluation

In this section, we present the results of an extensive evaluation of our data placement and routing alternatives with the TPC-R benchmark and the PowerDB system. Subsection 4.1 contains our findings regarding affinity. Subsection 4.2 compares different query routing algorithms. It shows that affinity-based

[1] With the TPC-R benchmark, queries Q_1 and Q_6 both scan the central fact table (LineItem) in order to compute certain aggregates. The concurrent execution of both queries at the same node is 25% slower than the sequential execution.

routing improves throughput only marginally, and that sequential evaluation of complex queries is superior. Therefore, Subsection 4.3 compares the two physical design alternatives using sequential routing only. Subsection 4.3 investigates the performance impact of simple updates.

Experimental Setup. The TPC Benchmark R [TPC99] is the successor of the TPC-D benchmark. It consists of 22 OLAP queries and two data modification streams, called "refresh functions". Given a scaling factor sf, the first one (RF1) inserts $sf * 1500$ new orders and corresponding lineitems into the database, while the second refresh function (RF2) deletes them. We have used the query stream corresponding to the query sequence O(00) of the specification. Each component has the complete data set of the TPC-R benchmark with scaling factor 0.1. The data and the indexes sum up to a database size of about 300 MBytes for each component. With the hybrid approach, the `LineItem` relation has been partitioned over the components according to the `L_OrderKey` attribute.

4.1 Affinity

This subsection presents and explains the so-called *affinity matrix* of the TPC-R benchmark used in the affinity-based routing algorithm as described in Figure 6. In a nutshell, each cell of the affinity matrix contains the affinity value of a pair of queries. To come up with the affinity matrix, we have looked at the characteristics of the queries, i.e., size of relations, selectivity of predicates, execution plan, obtained from the query optimizer. We have evaluated query pairs for which we

Query	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}
Runtime	53s	11s	190s	123s	327s	181s	345s	57s	211s	127s	31s
Query	Q_{12}	Q_{13}	Q_{14}	Q_{15}	Q_{16}	Q_{17}	Q_{18}	Q_{19}	Q_{20}	Q_{21}	Q_{22}
Runtime	235s	25s	43s	77s	16s	14s	106s	20s	64s	79s	13s

Table 1. Runtimes of single TPC-R Queries.

have expected a relatively high degree of affinity. We have run such queries both sequentially and concurrently. In more detail, the affinity matrix gives us the effect of concurrent query evaluation on mean response time (MRT):

$$affinity(Q_x, Q_y) = 100 - \frac{MRT(Q_x || Q_y) * 100}{MRT(Q_x; Q_y)}$$

The affinity value represents the improvement (in percent) of mean response time by concurrent execution of two queries. The faster the concurrent execution is compared to the sequential one, the higher is the affinity value. In the formula, $MRT(Q_x || Q_y)$ denotes the mean response time of Q_x and Q_y executed concurrently, and $MRT(Q_x; Q_y)$ stands for the mean response time of Q_x and Q_y where Q_y is executed after Q_x. I.e.,

$$MRT(Q_x; Q_y) = \frac{ExecutionTime(Q_x) + ResponseTime(Q_y, [Q_x])}{2}$$

$ResponseTime(Q_y, [Q_x])$ is the response time of Q_y whose execution starts after evaluation of Q_x. In other words,

$$ResponseTime(Q_y, [Q_x])) = ExecutionTime(Q_x) + ExecutionTime(Q_y)$$

Table 2 is the affinity matrix for the TPC-R queries we have measured (cells of unmeasured pairs are left blank). To compensate runtime fluctuations, we only consider affinity values higher of at least 2%, otherwise, there is a '-'. As can been seen, the matrix is only sparsely populated, i.e., affinity rarely occurs.

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}	Q_{16}	Q_{17}	Q_{18}	Q_{19}	Q_{20}	Q_{21}	Q_{22}	
Q_1	-		10%			-												-		-	-		Q_1
Q_2																							Q_2
Q_3	-		32%	15%	6%	-	4%		-	-		-	3%		-		-	-	-	-	-		Q_3
Q_4			3%	2%	35%	15%	26%	-	10%	10%		23%	-		-		3%	4%	-		-	-	Q_4
Q_5			-	-	29%	-	-		-	9%		-			-		-	-	21%		-		Q_5
Q_6	-		-	-	7%	32%	-		-	-		26%						-		-	-		Q_6
Q_7			-	-	-	-	31%		-	-		-	-		-		-	-			-		Q_7
Q_8			6%	15%	25%		30%										6%		-				Q_8
Q_9			-	-	-	-	-		10%	-		-						-	-				Q_9
Q_{10}			8%	8%	32%	10%	19%	-	-	-		19%	-					-	-		-		Q_{10}
Q_{11}																							Q_{11}
Q_{12}			-	-	6%	18%	-		-	-		33%					2%	-			-		Q_{12}
Q_{13}			31%	37%	36%		-			2%													Q_{13}
Q_{14}						-											8%						Q_{14}
Q_{15}						2%									-								Q_{15}
Q_{16}								4%			6%						7%		-		33%		Q_{16}
Q_{17}		5%	47%	47%	48%		48%	4%		2%					5%				9%		44%		Q_{17}
Q_{18}	-		-	-	-	-	-		-	-		-						-	-	-	-		Q_{18}
Q_{19}			44%	42%	46%		46%	4%		-							12%						Q_{19}
Q_{20}	-		23%	20%	32%	24%	36%	-		2%	4%					8%	6%	17%	3%	2%	3%		Q_{20}
Q_{21}	-		-	5%	13%	-	-					-						-	-	-	8%		Q_{21}
Q_{22}			42%	44%	45%		-	4%		2%													Q_{22}
	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}	Q_{16}	Q_{17}	Q_{18}	Q_{19}	Q_{20}	Q_{21}	Q_{22}	

Table 2. Affinity Matrix of TPC-R Queries.

The affinity matrix is not symmetric. To explain this, we have a closer look at a relatively high value in the matrix. For instance, it is of advantage to assign Q_{17} to a component that currently evaluates Q_3, but the opposite is not. There is a large runtime difference between these queries, i.e., Q_3 with 160 seconds versus Q_{17} with 4 seconds (cf. Table 1). In other words, the improvement is not the consequence of query evaluations favoring each other, but of a reduced waiting period. Hence, mean response time does not adequately reflect affinity if there is such a large runtime difference. As the effect described above is typical, we conclude that exploiting affinity systematically should not bring any significant improvement. Subsequent experiments with affinity-based routing policies will validate this claim.

"Short Queries ASAP." TPC-R consists of a mix of queries some of which are complex analysis queries, some of which are short queries. As a consequence of our evaluation on affinity, we suggest the so-called "Short-Queries-As-Soon-As-Possible" routing policy (ShortQueriesASAP) given in Figure 7. In order to minimize the waiting period of short queries, the *ShortQueriesASAP* policy allows them to be executed concurrently with one other query. Only long queries have to wait in the input queue until a free node is available. Note that this is indeed a routing algorithm. It does not schedule the queries, i.e., sort the input queue.

if (*query in the head of the input queue is a long query*)
evaluate query on a component that currently does not evaluate any query
else if (*query in the head of the input queue is a short query*)
evaluate query on a component that either is not evaluating any query, or that is currently evaluating a long query, but no other short query

Fig. 7. Short-Queries-ASAP Routing Policy.

4.2 Routing Algorithms

To evaluate our alternative routing algorithms, we have executed a stream of TPC-R queries on the database cluster with full replication. The query stream corresponds to the query sequence O(00) of the TPC-R specification. The three routing strategies are *Balance-the-Number-Of-Queries*, *Short-Queries-ASAP* and *Affinity-Based-Routing*. With regard to Balance-the-Number-of-Queries, we have measured the following variants: one without a maximum load per node, subsequently referred to as *BalanceQNr*, and one with at most one query per node, referred to as *Sequential*.

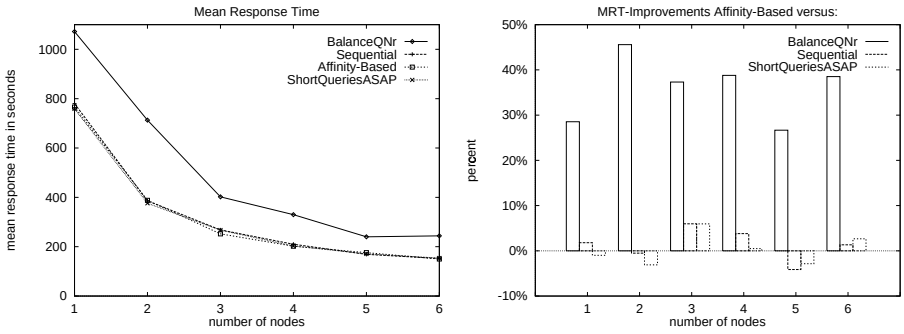


Fig. 8. Comparison of Routing Strategies with Full Replication.

As shown in Figure 8, sequential evaluation is among the best strategies. As expected, affinity-based routing improves the mean response time only slightly, e.g., for three components by 6% compared to sequential routing. The simple routing policy Short-Queries-ASAP introduced in the previous Subsection performs quite reasonable, but there is no general performance gain versus sequential or affinity-based. As the values for Sequential, Short-Queries-ASAP and Affinity-Based routing are always very close, the right diagram in Figure 8 "zooms in" and displays the relative difference between the alternatives. Figure 8 also tells us that the concurrent execution of multiple queries at one node (BalanceQNr) is clearly suboptimal. The mean response time of BalancingQNR-routing is 30% to 40% slower than sequential or affinity-based routing.

4.3 Physical Database Design Alternatives

After investigating the different routing algorithms, we now compare the two design alternatives full replication and hybrid design. We first study query per-

formance in isolation, we then investigate the effect of concurrent updates on query streams. From now on, the routing strategy is always the sequential one.

Query Performance For this experiment, we have issued the set of the 22 TPC-R queries as one batch to the coordinator, and have measured the runtime of each query separately. Figure 9 shows the mean response time and throughput of the TPC-R queries with the two design alternatives for different numbers of components.

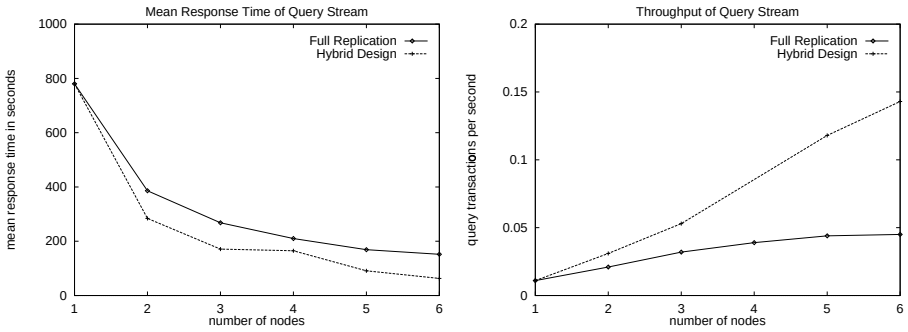


Fig. 9. Querying Performances of Physical Database Design Alternatives.

The hybrid design improves throughput significantly: With a system size of six nodes, the throughput is three times better compared to full replication. The reason is that due to the partitioning of the `LineItem` relation 17 of the 22 TPC-R queries gained intra-query parallelism.

The throughput improvement of the PowerDB cluster with full replication is linear with the number of nodes (see Figure 9) as one might have expected. E.g., with six components the mean response time of the TPC-R is approximately a sixth of the response time with one node. The same is true for the throughput, which is about five times higher. However, the performance improvement with hybrid is better, and the reason is as follows: the `LineItem` relation is partitioned into smaller fragments. This allows the component to use more efficient execution plans. We have verified this for some queries. Oracle has taken the reduced size of the `LineItem` partitions into account; the smaller partitions allow for in-memory sorting. Hence, the system has generated execution plans with merge joins whenever possible. The reduced cost of the join and the better caching behaviour of smaller partitions together with increased intra- and inter-query parallelism yield a throughput improvement of 12 with six components.

Query Performance with Concurrent Updates Finally, we study the effect of concurrent updates on query performance. In this experiment, we have measured the throughput of the 22 TPC-R queries with different rates of concurrent updates. We have configured the coordinator to run up to one query and update concurrently at each component. Since we do not consider concurrency control

issues, we have not used the TPC-R refresh functions for this experiment. Instead, we have used a stream of single-update transactions on `LineItem`. With full replication, such updates affect all components, while queries run on a single component. With hybrid, the situation is the reverse. Due to the single-action transactions, no conflict check is necessary. The results presented here are valuable to assess the performance with both designs under concurrent updates. To allow for easier comparison, we have included the runtimes of the pure query streams from Section 4.3 (no concurrent updates) in Figure 10 (cf. Line A).

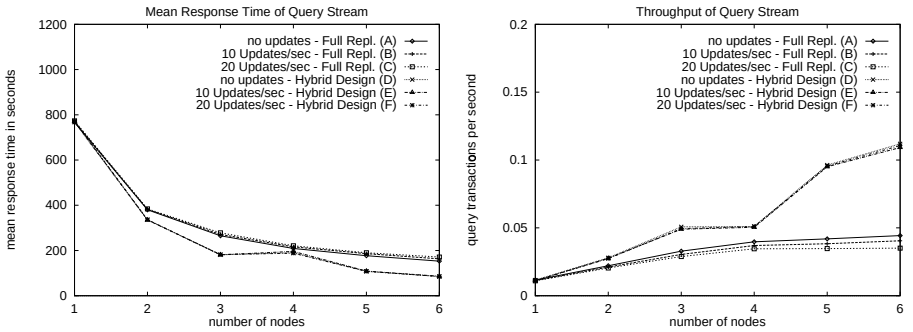


Fig. 10. Querying Performances with Concurrent Updates.

With both design alternatives, query performance decreases by about 10% with a concurrent stream of 10 updates per second. Increasing the update rate to 20 updates per second did not yield a much bigger deterioration. The slowest mean response time was about 20% slower than without updates. The overall results are the same as the ones for the scenario without updates. With a system size of six nodes, the throughput with the hybrid design is about three times higher, compared to full replication. Furthermore, the hybrid design proved more robust with regard to concurrent updates: the slowdown of the query stream is much less than with full replication. For example, with six nodes and 20 updates per second, the query performance of hybrid design (F) is 98% of the performance without updates (D), while it is at 79% with full replication (C).

5 Related Work

The bottomline of this section is that, even though admission control and routing are well-known concepts that have been investigated some time ago, previous research has not addressed these issues for OLAP queries.

Transaction Routing. In literature, the term 'transaction routing' normally means 'query routing': only with queries there typically is a choice of sites where to execute the transaction. — Carey et al. [CLL85] address the problem of routing queries in a distributed (shared-nothing) database system with full replication. Based on the classification of queries into either CPU-bound or I/O-bound, they propose two algorithms: *BNQRD* ("Balance the Number of Queries by Resource

Demands") routes queries to the site with the smallest number of queries of the same class, while *LERT* ("Least Estimated Response Time") uses estimates of the I/O and CPU demand to route a query such that the estimated response time is minimized. The improvement of both algorithms is in the $10\% - 20\%$ range. [Tho87] refines this approach by classifying queries not only into CPU- or I/O bound, but also into a finite set of types based on their resource demands, i.e., CPU, memory, and I/O considered separately for each disk. However, such classifications do not help in scenarios where all queries are I/O bound, e.g., queries that carry out scans over relations.

[YCDI87] presents an *affinity-based* approach to transaction routing. Incoming transactions are classified into affinity groups based on their pattern of page references. With this routing scheme, there is a component assigned to each affinity group, and each transaction goes to the component of its group. The classification is static, and does not consider changes of workload patterns. Several works extended this approach, e.g., [FGND93] additionally introduced average response time goals for each class. However, all the classifications do not take the kind of access into account, but only the data referenced. Thus, if two queries scan the same large table, obstruction occurs. Finally, for a general classification framework of different routing strategies see [Rah92].

Admission Control. Other related work has investigated admission/load control for a single database system. [MW91] presents a conflict-driven approach for automatic load control. Its algorithm prevents data-contention trashing by monitoring a performance metric, the so-called *conflict rate*, and by reacting to critical system states by postponing the execution of newly arriving transactions or aborting transactions currently running.

6 Conclusions

This work has shown that simple techniques for physical design and for routing in a database cluster architecture lead to significant performance improvements. Our scenario included OLAP queries, OLTP queries, and updates taken from the TPC-R benchmark. It has produced results that confirm our work on PowerDB, but that are also interesting in themselves: we have found out that affinity of queries is not larger than in scenarios others have considered earlier, i.e., OLTP queries only. On the other hand, we have observed that complex analysis queries may obstruct each other. This means that the system should not try to evaluate queries concurrently. Regarding a parallel architecture, we have evaluated two basic alternatives for data organization, full replication and hybrid, together with different routing schemes. It turns out that hybrid is superior to full replication in all the scenarios considered, and its speedup is more than linear. However, we should point out that the size of the biggest relation in the database limits the speedup obtained with hybrid. For a very large number of components, a combination of our elementary alternatives is probably appropriate. Finally, updates consisting of only one update per transaction lead to a very moderate slowdown of the query stream, e.g., 2% in one particular setting. This leads us to

the overall conclusion that coordinator-based routing has good scaleup properties for scenarios with complex analysis queries. For the evaluation, we have used the PowerDB prototype, an implementation of a 'database of databases' based on off-the-shelf hardware and software components. In the future, we will investigate the effects of combining scheduling with routing. This includes full transaction support as well as reordering of the input queue.

References

[BFGe95] C.K. Baru, et al. Dbs2 parallel edition. *IBM Systems Journal*, 34(2), 1995.

[CLL85] Michael J. Carey, Miron Livny, and Hongjun Lu. Dynamic task allocation in a distributed database system. In *Proceedings of the 5th IEEE Int. Conf. on Distributed Computing Systems (ICDCS), Denver, Colorado*, May 1985.

[FGND93] D. Ferguson, et al. Satisfying response time goals in transaction processing systems. In *Proceedings of the 2nd Int. Conf. on Parallel and Distributed Information Systems, San Diego*, 1993.

[GBS99] T. Grabs, K. Böhm, and H.-J. Schek. A document engine on a db cluster. *High Performance Transaction Systems Workshop (HPTS)*, Sept. 1999.

[GDQ92] S. Ghandeharizadeh, D.J. DeWitt, and W. Qureshi. A performance analysis of alternative multi-attribute declustering strategies. In *Proceedings of the 1992 ACM SIGMOD Conference, San Diego, California*, pages 29–38, 1992.

[JLS99] H.V. Jagadish, L.V. Lakshmanan, and D. Srivastava. Snakes and sandwiches: Optimal clustering strategies or a warehouse. In *Proceedings of the 1999 ACM SIGMOD Conference, Philadelphia*, pages 37–48, June 1999.

[KS96] H. Kaufmann and H.-J. Schek. Extending tp-monitors for intra-transaction parallelism. In *Proceedings of PDIS'96, Miami*, December 1996.

[MD97] M. Mehta and D.J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal*, 6(1):53–721, 1997.

[MW91] Axel Moenkeberg and Gerhard Weikum. Conflict-driven load control for the avoidance of data-contention trashing. In *Proceedings of the 7th IEEE Int. Conf. on Data Engineering (ICDE), Kobe, Japan*, pages 632–639, 1991.

[Ora97] Oracle Corporation. *Oracle8 Server Concepts, Release 8.0, Chapter 5*, 1997.

[OV91] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*, chapter 7–9. Prentice Hall, 1991.

[Rah92] Erhard Rahm. A framework for workload allocation in distributed transaction processing systems. *Systems Software Journal*, 18:171–190, 1992.

[RB99] U. Röhm and K. Böhm. Working together in harmony — an implementation of the corba object query service and its evaluation. In *Proc. of the 15th IEEE Int. Conf. on Data Engineering, Sydney, Australia*, March 1999.

[RNS96] M. Rys, M. C. Norrie, and H.-J. Schek. Intra-transaction parallelism in the mapping of an object model to a relational multi-processor system. In *Proc. of the 22nd Int. Conf. on Very Large Databases, Mumbai, India*, 1996.

[Tho87] A. Thomasian. A performance study of dynamic load balancing in distributed systems. In *Proceedings of the 7th IEEE Int. Conf. on Distributed Computing Systems (ICDCS), Berlin, Germany*, 1987.

[TPC99] Transaction Processing Performance Council. Tpc-r benchmark specification rev. 1.0.1. Technical report, July 1999.

[YCDI87] P. S. Yu, et al. Analysis of affinity based routing in multi-system data sharing. *Performance Evaluation*, 7:87–109, 1987.