Cache-Aware Query Routing in a Cluster of Databases

Uwe Röhm Klemens Böhm Hans-Jörg Schek
Swiss Federal Institute of Technology
ETH Zentrum, 8092 Zurich, Switzerland
{roehm,boehm,schek}@inf.ethz.ch

Abstract

We investigate query routing techniques in a cluster of databases for a query-dominant environment. The objective is to decrease query response time. Each component of the cluster runs an off-the-shelf DBMS and holds a copy of the whole database. The cluster has a coordinator that routes each query to an appropriate component. Considering queries of realistic complexity, e.g., TPC-R, this article addresses the following questions: Can routing benefit from caching effects due to previous queries? Since our components are black-boxes, how can we approximate their cache content? How to route a query, given such cache approximations? To answer these questions, we have developed a cache-aware query router that is based on signature approximations of queries. We report on experimental evaluations with the TPC-R benchmark using our PowerDB database cluster prototype. Our main result is that our approach of cache approximation routing is better than state-of-the-art strategies by a factor of two with regard to mean response time.

1. Introduction

A cluster of databases is becoming a cost-effective alternative to multiprocessor database management systems on mainframes. Instead of scaling up the idea is to "scale out" [8] by adding more database components to the cluster. The challenge with such an architecture is to coordinate the cluster components in order to provide a simple database system view to the clients.

The object of this present investigation is a cluster of relational databases, i.e., each *component* of the cluster runs an off-the-shelf RDBMS. The architecture is coordination-based (cf. Figure 1). Clients communicate only with a distinguished node, the *coordinator*. Incoming queries arrive at the coordinator via an *input queue*. The coordinator decides on the execution order and the routing of the queries. This paper assumes full replication. This gives us maximum flexibility where to execute a query. Given this, our objective is to base the routing decision on the states of the component caches. This brings up a number of questions:

- How to obtain or at least approximate the state of a component cache? In a cluster architecture, with components being black-boxes, one cannot access the cache directory and look up the content of the cache.
- How to quantify the benefit from evaluating a given query on a particular component, given an approximation of the cache state? With 'low-level' data objects, i.e., disk pages and page access operations, the caching benefit is either 1 (= page is in the cache) or 0 (= page is not in the cache). When looking at queries that possibly refer to a number of relations, we need a more sophisticated model.
- Having estimated that benefit, how to actually route queries? Are the cache states the only parameter of the routing decision?
- Which performance improvements can be achieved using such a *cache approximation* router? How does such a router compare to one that has apriori knowledge regarding the caching benefit?

Our interest is to learn how much we can achieve in quantitative terms with simple mechanisms addressing these questions. We investigate OLAP queries, i.e., queries that are I/O intensive and long running. Our general idea is to derive the cache content or at least an approximation of it from the queries that a component has evaluated most recently. Throughout this paper, we will work with various approximations of the content of a cache with

different degrees of sophistication. We will refer to such an approximation briefly as *cache state*. We will also discuss and evaluate alternative definitions of the *benefit of a cache state for a query*. Based on the benefit values, the coordinator routes the queries. This may also include reordering of the input queue.

In more detail, we proceed in two steps: first, we consider the case that the set of queries is fixed. Note that other work on query routing limits itself to this restrictive case [14, 1]. We for our part see this case as an upper bound of the benefit of cache-aware routing. It allows to pre-compute benefits of cache states for queries. As a second step, we consider the general case where queries are arbitrary. In this case, we dynamically compute the caching benefit by approximating the sets of tuples accessed by queries. We have run extensive experiments using a prototype system. By using all queries of the TPC-R benchmark (and not only a subset), our experimental setup is realistic. We have evaluated the different approaches using two metrics: mean response time and throughput. The experiments study the effect of varying cache sizes as well as of different numbers of components. This present investigation does not consider update statements.

Our main results are as follows: Cache approximation based query routing significantly reduces mean response times, as compared to routing schemes that are not cache-aware. Different definitions of 'benefit' fare differently with the two evaluation metrics. We obtained the best results when we approximated the cache content using bit string signatures of queries and normalized the benefit values with the execution costs of statements. Such a routing algorithm more than halves mean response time, as compared to non-cache-aware strategies. Since our approach deals with conjunctive SQL queries and black-box components, we believe that it is very general, and that our results are remarkable.

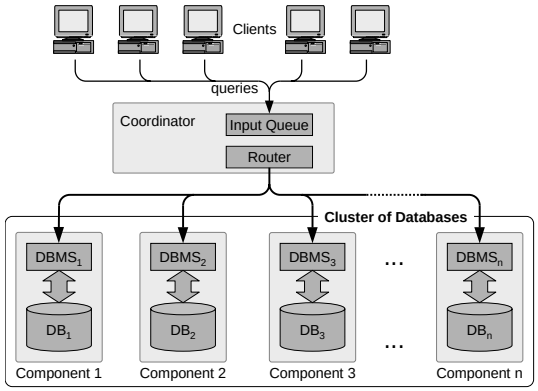

Figure 1. Architecture of PowerDB.

This work is part of the *PowerDB* project presently conducted at ETH Zurich. The objective is to build a high-performance DBMS using off-the-shelf hardware and software components as much as possible. Another research interest of *PowerDB*, besides query routing, is scheduling at the coordinator level. However, this is not the topic of this paper.

The remainder of this paper has the following structure: The subsequent section discusses related work. In Section 3, we give an overview of query routing in a coordination-based architecture. Section 4 describes different routers compared in this paper, signature-based approximation of query results, and corresponding benefit models. Section 5 describes the results of our performance evaluations. Section 6 concludes the paper.

2. Related Work

An important issue that seems to be closely related to the topic of this paper is distributed caching [15, 13]. There, the underlying assumption is that each component has a page cache. An important observation is that it may be cheaper to fetch pages from a remote cache than from disk [13]. Work on distributed caching addresses the questions where to fetch the pages and how appropriate replacement policies look like. For example, instead of just dropping a page from a cache, we might transfer it to another cache [15]. Distributed caching is also an issue in WWW proxy management [6] and with regard to operating systems (global memory management) [7]. However, these investigations evaluate cache maintenance policies. Their basic assumption is that the cache manager can be implemented from scratch or that at least the cache directory is accessible. We in turn realistically assume that the cache directory is part of a component and hence is not readable from outside. In consequence, caching strategies for distributed systems cannot be implemented on top of off-the-shelf database components. Another difference between our approach and other work on distributed caching is as follows: we consider complex OLAP queries instead of simple access operations to 'low-level' data objects, i.e., page reads.

Another recent issue with regard to caching is refined cache admission and replacement strategies, known as *semantic caching*: [11] caches query results as a whole based on a sophisticated cost model. But it relies on an exact query match. [12] refines this work in that it looks for cached query results which subsume a submitted query. [4] goes into a similar direction by introducing the notion of *remainder query*. Such a query retrieves only data that is not already in the cache. [5] explicitly considers

aggregate relationships for cache management and computation of remainder queries. Work on semantic caching is usually based on assumptions different from our current ones. However, there is a relationship between semantic caching and this work here. Consider a DBMS that is a component in a composite system. Furthermore, assume that it implements semantic caching. In this case, cache approximation could take the particular relationship between queries and cache content into account as well in order to yield better results. But as far as we know, this is currently not the case for the components considered in this study.

"Transaction routing" [1, 14, 16] means "query routing" in many cases. For a good overview and classification we refer to [9]. Previous work on transaction routing again assumes the cache manager to be open and manipulable. Furthermore, it only considers page-level access operations, but not SQL queries. In a previous investigation [10], we have compared several non-cache-aware routing strategies for OLAP queries in a black-box scenario. We have concentrated on the question whether OLAP queries should be evaluated in parallel or consecutively. It has turned out that parallel execution of such complex queries leads to I/O trashing in most cases. This deteriorates performance. Hence, in this work, we restrict the degree of multiprogramming for OLAP queries to one. Another finding of [10] has been that rearranging the input queue based only on estimated execution times does not yield significant improvements in an OLAP scenario.

3. Basic Aspects of Query Routing

Before we describe cache approximation routing in Section 4, we review query routing in a coordination-based architecture. We discuss basic aspects of query routing which are independent of the actual routing algorithm and we describe one specific routing algorithm for illustrative purposes. This algorithm also serves as reference in our evaluation.

3.1. Routing in a Coordination-Based System

In a coordination-based architecture like PowerDB, clients issue their queries to a distinguished node of the database cluster, the coordinator (cf. Figure 1). More precisely, the clients place incoming queries into the input queue of the coordinator. In the context of this paper, client queries are decision-support SQL queries. We also assume full replication of the data. Hence, different components are able to execute queries in parallel. In our terminology, *routing of a (single) query* is the decision which component shall execute a query.

The coordinator processes the input queue periodically or if the queue exceeds a certain threshold size. Given this, *routing of a set of queries* consists of two steps: first, the coordinator decides in which order to route the queries in the input queue. Second, it routes the individual queries in this order. We illustrate both steps by means of one specific routing algorithm: *First-Come-First-Free-Server (FCFFS)* routing.

3.2. First-Come-First-Free-Server Routing

With *FCFFS* routing, the coordinator iterates through the input queue according to a first-come-first-served policy. For each query, it invokes the routing algorithm – as long as there are free components. The coordinator also passes the list of components which can execute the query to the router. In

```
function FCFFS_Routing ( list<Comp> comps,
                         Query q ) : Comp
begin
  stable_sort(comps, load_comparison());
  if ( comps[0].load == 0 ) then
    return comps[0];
  else
    return nil;
end
```

Figure 2. *FCFFS* **routing algorithm.**

step two, the *FCFFS* algorithm routes the query to the first free component. It does so by sorting the list of components by the number of queries currently active on the components, i.e., their load (cf. Figure 2). The load of the components is a typical runtime statistic maintained by the coordinator. The first component with load 0, i.e., the first free component is finally chosen. *FCFFS* corresponds to *Balance-the-Number-of-Queries* routing [10, 1] with a degree of multiprogramming of one. Obviously, the routing decision in the case of this simple algorithm is independent of the current query. This is the starting point of our present work.

4. Cache Approximation Query Routing

The objective of query routing is to reduce query response time. The execution times of queries – especially OLAP queries – are dominated by I/O costs. While caching plays a major role for performance of query evaluation, a simple routing algorithm like

FCFFS is not aware of caching effects. In the following, we investigate *cache approximation (CA)* routing strategies. In our terminology, cache approximation routing approximates the cache contents for routing to achieve maximum benefit from the component caches.

Our investigation of cache approximation routing consists of two steps. The first step relies on precomputed benefits. The second step dynamically estimates the benefit of a cache state for a query. The treatment of the input queue is the same for both approaches. We will discuss this particular aspect first.

4.1. Input Queue Reordering

Cache approximation routing estimates the benefit of a cache state for a query. The idea is that the execution time of a query is minimal at the component whose cache contains the largest subset of data that will be accessed by the query. As we explained in Subsection 3.1, query routing for a set of queries consists of two steps: the coordinator first decides on the processing order for the queries in its input queue, before it routes the individual queries.

In case of cache approximation routing, in Step I the coordinator no longer processes its input queue in a first-come-first-served manner. Instead, the coordinator reorders the queries in the input queue according to the estimated benefit values. The query for which the routing algorithm estimates the highest benefit is executed first. However, the coordinator has to ensure that all queries are still processed. That is, it must avoid starvation of queries with low benefit values. Consequently, the coordinator tags queries with an age and reorders the input queue according to both benefit and age of the queries.

In Step II, the routing algorithm decides on the target component to execute the current query. In the following, we discuss several approaches to such cache approximation routers.

4.2. Cache-Approximation Routing using Pre-Computed Caching Benefits

For this subsection, we assume that the set of possible queries is known in advance. This allows to make use of previously recorded runtime statistics on these queries. In more concrete terms, we define the cache state of a component as the query most recently executed there. The hypothesis is that OLAP queries typically access more data than fits into the cache. Hence, we only need to consider the most recent query because all unnecessary data loaded by previous queries will be swapped out. We define the benefit of a cache state for a query as follows: it is the reduction of disk accesses, as compared to the execution on a flushed cache. The assumption that the set of possible queries is fixed allows to determine the benefit values in advance. They are stored in a static benefit matrix. The matrix may comprise absolute or relative benefits. Consequently, we distinguish two router variants, which we call *CAabs* and *CArel*.

Caching Benefit Matrix. In order to determine the benefit matrix for our TPC-R scenario, we investigated the queries with fixed parameters. Note that this is a very restrictive assumption. We first measured the cache misses of each query when executed on a flushed database cache (i.e. containing no page of the database). We denote this with $CacheMisses_{FlushedCache}$. Then, we measured the disk accesses of the query when it executed after another query. This value is called $CacheMisses_{After}$. For *CAabs*, the benefit matrix contains the absolute difference between both numbers:

$$CacheMisses_{FlushedCache} - CacheMisses_{After}$$

In the case of *CArel* routing, the entries specify the relative improvement of cache misses in percent, calculated by the following formula:

$$(1 - \frac{CacheMisses_{After}}{CacheMisses_{FlushedCache}}) * 100$$

The higher these values, the more the current query should benefit from the cache content. Figure 3 shows the relative benefit matrix for the TPC-R queries and a cache size of 40000 pages. The higher the benefit values, the darker the corresponding matrix cell.

Figure 3 shows that most queries are faster when being executed a second time. This is what one would expect. But the benefit matrix is not symmetric. High values typically occur if the subsequent query is small with regard to its total number of disk accesses. Consider, for example, the TPC-R queries Q13, Q14, Q15, Q17, or Q19. These are less I/O intensive queries. Therefore, their execution time strongly varies depending on the overlap of data accessed by them and the previous query. On the other hand, an I/O intensive query like Q7 or Q18 cannot benefit much when run after such a short query.

Routing Algorithm. Both *CAabs* and *CArel* keep track of the last query executed at the components. They choose the component promising the least I/O effort based on caching benefit values. The general algorithm is illustrated in Figure 4. The pre-

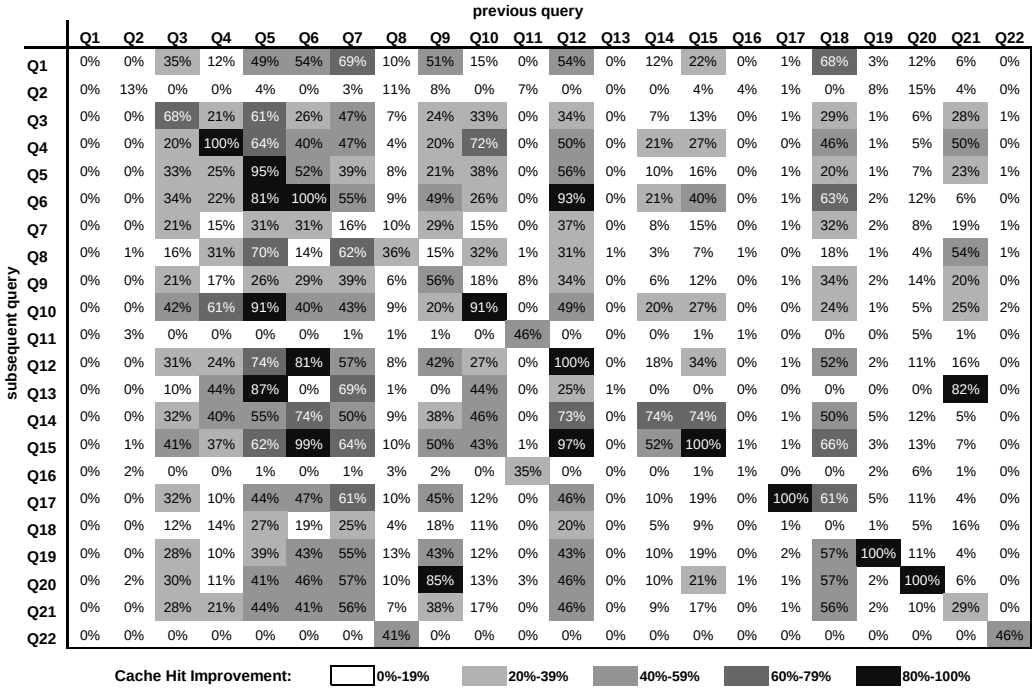

Figure 3. Relative caching benefit matrix of *CArel* router for TPC-R queries.

```
function CA_Routing ( list<Comp> comps,
                      Query q ) : Comp
var  ca_benefit : matrix[22,22] of int:=(...);
     choice : Node;  benefit : int;
begin
  benefit := -1;
  for c in comps do begin
    if ca_benefit[c.last][q]>benefit then
      benefit:= ca_benefit[c.last][q];
      choice := c;
    fi
  end; return choice;
end
```

Figure 4. *CArel* routing algorithm.

computed caching benefits allow for optimal routing decisions. The router can predict the number of cache misses of the queries. Hence, cache approximation routing based on pre-computed benefit values will serve as an orientation guide for evaluating the different cache approximation schemes.

4.3. Dynamic Cache Approximation

In the previous subsection, we have assumed a known set of possible queries. We drop this assumption in this subsection. In contrast to *CAabs* and *CArel*, which only keep track of the most recent query, we will describe cache states more explicitly.

The typical granularity of component caches are disk pages. Any data needed for query execution is loaded into a component cache in form of disk pages. Ideally, the state of a cache would be defined as its set of pages. But with the black-box assumption, the coordinator has no access to the cache directory of the components. Hence, we must rely on information collected separately. Intuitively, the set of tuples accessed by a query indirectly defines the set of data pages read from disk during execution. Consequently, our idea is to approximate cache states by approximations of the set of tuples accessed by a query. This can be done with different degrees of precision. The following subsections reflect this.

4.3.1. Cache Approximation via FROM-Clauses

A straightforward approach to approximate the set of tuples accessed by a query is to keep track of the relations accessed. This is simply achieved by analyzing the FROM clause of the query. We approximate the state of component caches by the set of relations accessed by the most recent n queries. Let Q_i^C denote the ith most recent query executed at component C, i.e., Q_1^C is the last query. We define the cache state as follows:

$$rels(Q) := \{R \mid R \text{ contained in FROM clause of query } Q\}$$

$$state(C,n) := \bigcup_{1 \le i \le n} rels(Q_i^C)$$

Further, we define the benefit of the cache state of component C for query Q by the size of intersection of the cache state and the FROM clause of the query:

$$benefit(Q, C, n) := |state(C, n) \cap rels(Q)|$$

Subsequently, we refer to a router that implements this strategy as *CAF* router.

4.3.2. Cache Approximation via Query Signatures

The above approach relies on a very rough approximation of queried data. It does not take into account which portions of the relations are actually accessed by a query. These are described by each query predicate. However, quantifying the exact overlap of the sets of tuples specified by two different predicates having common attributes is difficult. Consequently, we further approximate the set of tuples specified by a predicate using bit string signatures. By doing so, we reduce the calculation of benefits to bit string comparisons. This can be done efficiently.

Query Signature Generation. Our approach works with three different kinds of signatures: the *query signature*, which in fact is a set of signatures, each together with a query attribute, *attribute signatures* for each attribute occurring in a predicate of the query, and for each occurrence of an attribute in a query predicate a *predicate signature for the attribute*. The bits set in a query signature represent the data range possibly being accessed by the query. We build a query signature using disjoint coding [2], i.e., we generate one attribute signature for each query attribute. This in turn is based on the *query graph*: Each node of the query graph represents a relation accessed by the query. Each edge of the query graph stands for a join predicate between relations. A select predicate is represented by an edge that forms a cycle. Note that a query graph is defined only for conjunctive queries. Given a query graph, we first generate a predicate signature for each occurrence of an attribute in a query predicate. Then, we intersect the predicate signatures for an attribute to obtain its attribute signature. In more detail, we proceed in four steps:

Selection Predicate Signatures. In a first step, we consider each selection predicate p_{sel} of the form $(a\ \Theta\ const)$ of query Q. The corresponding predicate signature is a bit string, denoted by $predsig(p_{sel}, a, Q)$. It is generated according to the following scheme: The hash function $h()$ transforms constant $const$ into a bit position. Depending on the comparison operator Θ, only the corresponding bit or a whole bit range starting or ending at this position is set in the bit string (cf. Table 1). The hash function must be order-preserving, i.e., $x < y \Rightarrow h(x) \leq h(y)$.

Θ	$predsig(p_{sel}, a, Q)$	bits set
$=$	0 ... 010 ... 0	bit $h(const)$
$<, <=$	0 ... 011 ... 1	$[h(const)..0]$
$>, >=$	1 ... 110 ... 0	[last bit .. $h(const)$]
$\neq$,¬,like	1 ... 111 ... 1	all bits set

Table 1. Selection predicate signatures.

Join Predicate Signatures. In a second step, we generate signatures for join predicates p_{join} of the form $(a_R\ \Theta\ a_L)$ of query Q. This will result in two predicate signatures $predsig(p_{join}, a_R, Q)$ and $predsig(p_{join}, a_L, Q)$. They approximate the two sets of tuples accessed in the joined relations L and R, respectively. Our approach to join-predicate signature generation makes a distinction between two cases, namely equi-joins along foreign key relationships and all other join predicates.

The second case is simple: predicate signatures have all bits set. This also holds for all other kinds of predicates, e.g., comparisons with sub-queries. With regard to the first case, w.l.o.g., let a_R be the key attribute. Let $\wedge_b$ denote a *bitwise and* operation, $rel(a)$ the relation in which the attribute a occurs and $attrs(x)$ the set of attributes occuring in x. x can either be a relation, a predicate, or a query. Finally, let $selects(a, Q)$ denote the set of selection predicates of query Q over attribute a. With this, the generation of the predicate signatures for equi-joins along foreign key relationships is defined as follows:

$$predsig(p_{join}, a_R, Q) :=$$
$$\begin{cases} \bigwedge_{b\ a \in attrs(rel(a_R))} \left(\bigwedge_{b\ p \in selects(a,Q)} predsig(p, a, Q) \right) & \text{if } \alpha, \\ 1\ldots111\ldots1 & \text{otherwise.} \end{cases}$$
$$predsig(p_{join}, a_L, Q) :=$$
$$predsig(p_{join}, a_R, Q), a_R = attrs(p_{join}) \backslash \{a_L\}$$

where $\alpha = (a_R \in attrs(p_{join}) \wedge \exists a \in attrs(rel(a_R)) : selects(a, Q) \neq \emptyset)$

We first generate $predsig(p_{join}, a_R, Q)$, the signature for the key attribute a_R. We make use of the signatures already generated in the first step: we intersect all existing signatures of selection predicates (innermost bitwise and) for attributes of the relation of a_R (outermost bitwise and). If there is no selection predicate on any attribute of R, i.e., α does not hold, we set all bits. The rationale is to approximate the set of tuples selected in R. We use the same bit string for the signature of the foreign key attribute, too. This models the access on the corresponding join tuples in L.

Attribute Signatures. In the next step, we combine the predicate signatures to attribute signatures. Let $preds(a, Q)$ denote the set of all predicates of query Q in which attribute a occurs. An attribute signature for an attribute a is a bit string denoted by $attrsig(a, Q)$. We generate it by intersecting all predicate signatures for attribute a:

$$attrsig(a, Q) := \bigwedge_{p \in preds(a,Q)}^{b} predsig(p, a, Q)$$

Query Signatures. Finally, the query signature for query Q, denoted by $sig(Q)$, is the set of $(attribute, bitstring)$ pairs defined as follows:

$$sig(Q) := \{ (a, attrsig(a, Q)) \mid a \in attrs(Q) : preds(a, Q) \neq \emptyset \}$$

Example 1. Consider TPC-R query Q4:

```
SELECT O_OrderPriority, COUNT(*)
FROM  Orders
WHERE O_OrderDate >='01-JAN-1995'    (P1)
  AND O_OrderDate < '01-APR-1995'    (P2)
  AND EXISTS (SELECT * FROM LineItem
  WHERE L_OrderKey = O_OrderKey      (P3)
  AND L_CommitDate < L_ReceiptDate)  (P4)
GROUP BY O_OrderPriority ...;
```

The query contains four predicates: two selection predicates P_1 and P_2 on `O_OrderDate`. P_3 is a join predicate between the two relations `Orders` and `LineItem`, and P_4 a self-join on attributes of `LineItem`. Assume that $h(\text{'01-JAN-1995'}) = 4$ and $h(\text{'01-APR-1995'}) = 5$. Then, with a signature size of 8 bits[1], the following six predicate signatures will be generated:

$$\begin{aligned}
predsig(P_1, O_OrderDate, Q4) &= 11110000 \\
predsig(P_2, O_OrderDate, Q4) &= 00111111 \\
predsig(P_3, O_OrderKey, Q4) &= 00110000 \\
predsig(P_3, L_OrderKey, Q4) &= 00110000 \\
predsig(P_4, L_ReceiptDate, Q4) &= 11111111 \\
predsig(P_4, L_CommitDate, Q4) &= 11111111
\end{aligned}$$

The signature of the join attributes are derived from the two predicate signatures for `O_OrderDate`. The last predicate P_4 does neither contain a constant nor a key attribute. We can only 'approximate' it by two signatures with all bits set. Next, all predicate signatures are intersected into single attribute signatures:

$$\begin{aligned}
attrsig(O_OrderDate, Q4) &= 00110000 \\
attrsig(O_OrderKey, Q4) &= 00110000 \\
attrsig(L_OrderKey, Q4) &= 00110000 \\
attrsig(L_ReceiptDate, Q4) &= 11111111 \\
attrsig(L_CommitDate, Q4) &= 11111111
\end{aligned}$$

[1] Actually, *PowerDB* uses 128 bit signatures.

Hence, the query signature of Q4 looks as follows:

$$\begin{aligned}
sig(Q4) = \{ \quad & (O_OrderDate, 00110000), \\
& (O_OrderKey, 00110000), \\
& (L_OrderKey, 00110000), \\
& (L_ReceiptDate, 11111111), \\
& (L_CommitDate, 11111111) \ \} \quad \blacksquare
\end{aligned}$$

Cache State Approximation. How do these predicate signatures facilitate dynamic cache approximation? Recall that bits set represent the data range possibly accessed by a statement. Therefore, we define the approximated cache state of components by means of the predicate signatures of the recent n statements executed:

$$state(C, n) := \bigcup_{1 \leq i \leq n} sig(Q_i^C)$$

The parameter n is the history length which allows to adjust the degree of approximation.

Benefit Model. As a final step, we define the benefit of the cache state of a component C for query Q. The underlying hypothesis is as follows: the more bits in signatures of the same attribute coincide, the more data accessed by the query is already cached by the component, because the previous queries have accessed similar data. Let $\#_b()$ be a function which counts the number of bits set in a bit string. Then, the benefit is: $benefit(Q, C, n) :=$

$$\sum_{\substack{(a, bs_a) \\ \in sig(Q)}} \sum_{\substack{(a', bs_{a'}) \\ \in state(C,n)}} \begin{cases} \#_b(bs_a \wedge_b bs_{a'}) & \text{if } a = a', \\ 0 & \text{otherwise.} \end{cases}$$

We will refer to a cache approximation router which uses this benefit model as a *CAS* router.

Refined Benefit Model. The benefit model of *CAS* routing yields higher benefit values the more bits of the signatures match. This approach tends to prefer queries with signatures where many bits are set. These are normally multi-join queries with few selective predicates. In order to minimize mean response time, another idea is to execute short queries as soon as possible. To achieve this, we refine our benefit model. It now takes the actual execution costs of a query into account. To do so, the coordinator records the execution times of each query. If the same query is executed another time, its benefit value is finally normalized with the recorded execution time:

$$normalized\_benefit(Q, C, n) := \frac{benefit(Q, C, n)}{execution\_time_Q}$$

In case a query is executed the first time, we deliberately set its execution time to 1. We will refer to this variant as *CASweighted* router.

Discussion. While having developed a number of cache approximation methods by now, more sophisticated ones are still conceivable. For instance, the router could access the query optimizers of the component DBMSs and take the actual query plans into account as well. The rationale is to avoid concurrent scan and random access operations. Another refinement could consider attribute selectivities and the data distribution. As a further example, the router could try to differentiate between index-based access methods and full scans. Finally, we could try to capture the case that the amount of data retrieved to process a query exceeds the cache size.

However, our intention was to develop simple, but efficient cache approximation schemes for black-box components. The possible refinements just mentioned do not exactly meet this criterion. For instance, analyzing the query plan is highly platform-specific and induces much additional complexity. Instead, we now see the necessity to evaluate the routing algorithms developed so far. In particular, we must compare the routing schemes based on realistic assumptions to the 'good' ones, serving as orientation guides. If the difference is not too big, further refinements will not yield significant improvements.

5. Evaluation

We present the main results of an extensive evaluation of the following routing strategies:

FCFFS. *First-Come-First-Free-Server* routing which is not cache-aware. Its performance serves as an orientation point for the other strategies.

CAabs. Cache approximation routing, whose static caching benefit matrix contains absolute cache miss improvements.

CArel. Cache approximation routing based on a relative caching benefit matrix.

CAF. Dynamic cache approximation routing based on the FROM clause of queries.

CAS. Dynamic cache approximation routing based on predicate signatures.

CASweighted. Like *CAS*, but with a refined benefit model using normalized benefits.

5.1. Experimental Setup

All measurements have been carried out on a cluster of PCs (Pentium II, 400 MHz, 128 MBytes) under Windows NT 4.0. The coordinator ran on a separate PC. All computers were interconnected by a switched 100 MBit Fast-Ethernet LAN. We used ORACLE 8.0.4 as component database system with a cache size of 40000 pages. Each component database was generated and populated according to the specification of the TPC-R benchmark [3] with a scaling factor of 0.1. We fully replicated the data and the indexes on all nodes.

The client issued a stream consisting of 220 TPC-R queries to the coordinator with an inter arrival time with random distribution of up to 5 seconds. The execution order of the queries in the stream corresponds to the permutation orders 0 to 9 of the TPC-R specification. The client measured both the query throughput and the mean query response time for such a stream. Each data point represents the average value of at least five measurements. The standard deviation typically has been around 6% of the mean value which is sufficiently low.

5.2. Influence of Cache History Length

First, we are interested in the influence of the cache history length on the accuracy of the approximation schemes. Therefore, we fix the cluster size (8 nodes) and vary the number of previous queries recorded for cache approximation from one up to ten. The results are shown in Figure 5. We see that the

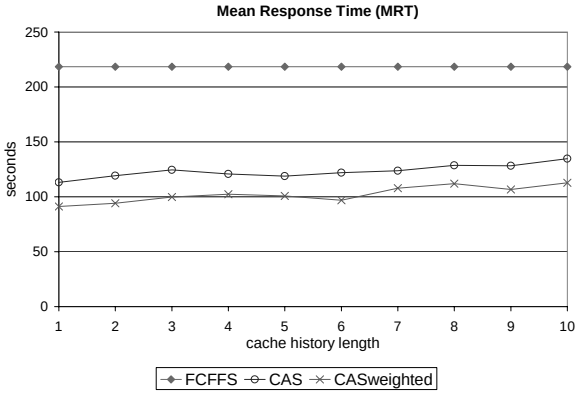

Figure 5. MRT Influence of history size.

dynamic cache approximation routers yield a general improvement of mean response time between 45% and 55%, as compared to *FCFFS*. However, with our present scenario of OLAP queries, increasing the cache history maintained by the dynamic cache approximation routers does not improve their performance. On the contrary, the mean response times of both *CAS* and *CASweighted* routing become slightly worse with longer histories. This is due to the complex queries we are investigating: about half of the TPC-R queries access more database pages than

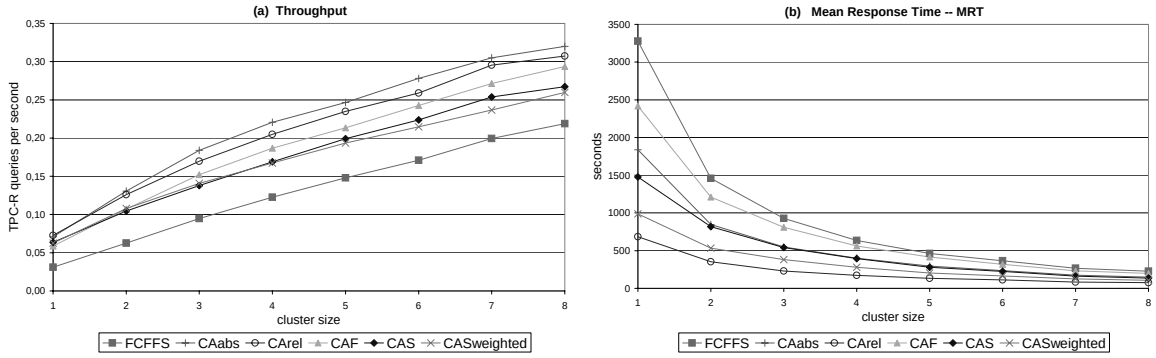

Figure 6. Throughput (a) and mean response times (b) with different routers.

can be cached. Therefore, it is not surprising that the cache state is sufficiently described by the most recently executed query. Consequently, we will use a cache history length of one in the following experiments. However, note that this consequence is not a general one, e.g., as if we would experiment with a workload consisting of short OLTP queries.

5.3. Scalability of Routing Algorithms

Next, we are interested in the overall performance of the different routing algorithms for different cluster sizes (up to 8 nodes). Figure 6 shows the throughput and mean response times of the query stream with the different routing algorithms and increasing cluster size. First, these experiments confirm our results reported in [10]. All routing algorithms show a nearly linear increase in throughput with increasing cluster size. Subsequently, we will use the straightforward approach of *FCFFS* as a basis for our comparisons. All cache approximation routers yield better results than *FCFFS*. However, there is a tradeoff between throughput and mean response time with cache approximation routing. For example, *CAabs* routing achieves higher throughput than *CArel* routing. In contrast, *CArel* yields shorter mean response times than *CAabs*. The reason is that *CAabs* routing prefers longer queries due to their higher benefit even if the improvement relative to the overall query length might be smaller. The opposite is true for the *CArel* router relying on the relative improvements. Now short queries are preferred. The same is true for *CAS* and *CASweighted* routers, respectively.

An interesting observation is that cache approximation routing is already faster than FCFFS with a 'cluster' consisting of only one node (cf. Figure 6(b)). This is due to the reordering of the input queue. In Section 4.1, we explained that the coordinator uses a *cheapest-action-first* policy in conjunction with cache approximation routers. With *CArel* routing, this reordering of the input queue cuts down mean response time on a one node 'cluster' by 80%.

In Figure 7, we 'zoom in' to make the differences of the five cache approximation routers more clear by scaling to the results of *FCFFS*. The relative throughput of the cache approximation routers strongly varies between 120 and 240 percent of the *FCFFS* throughput. With a cluster size of eight nodes, the static cache approximation routers still yield a throughput that is better by about 40%. The dynamic approximation routers achieve a slightly lower throughput than the routers based on precomputed benefit values. They end up with around 20% improvement as compared to FCFFS.

We can distinguish the different routers even better if we look at the improvement of mean response times (cf. Figure 7(b)). Due to its pre-computed knowledge of the benefits for all queries, *CArel* routing yields the lowest mean response times. They are between 20% and 30% of the one of *FCFFS*. That is, *CArel* routing improves the mean response time by a factor of 3. This is the yardstick for the dynamic approaches. The straightforward *CAF* router, which approximates cache states by the FROM clause of the queries, achieves about 10% better mean response times than *FCFFS*. This is the worst mean response time of all cache-aware routers. It is inferior to the cache approximation routers which use predicate signatures: *CAS* routing improves mean response time by around 40% as compared to *FCFFS*. This is even better than *CAabs* routing which uses previous knowledge about the absolute cache miss improvements of the queries. With the refined benefit model of *CASweighted*, we nearly reach the optimal result of the static *CArel* router. *CASweighted* yields around 55% faster mean response times, i.e., it is better than *FCFFS* by a factor of 2. This is remarkable, as in contrast to *CArel* it fully conforms to the black-box principle, with no unrealistic assumptions (e.g., no fixed parameters) on the queries.

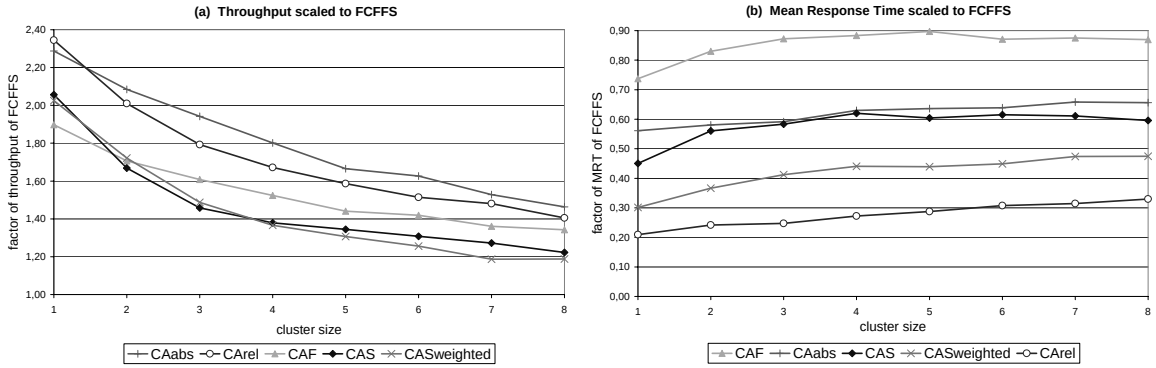

Figure 7. Throughput (a) and mean response time (b), as compared to *FCFFS*.

6. Conclusions

A cluster of databases is an interesting cost-effective alternative to mainframes. In our *PowerDB* project, we pursue a coordination-based approach: a central node, the coordinator, coordinates a cluster of database systems. The coordinator decides what to store at each component and where to evaluate queries. Assuming full replication, we are interested in query routing strategies to decrease query response time. The underlying observation is that query evaluation performance strongly relies on caching. But in a composite system with *black-box* components it is impossible to directly manipulate or retrieve the content of the component caches. In this paper, we presented a new approach to query routing over black-box components, namely signature-based cache approximation routing. We make use of predicate signatures, i.e., bit string approximations of the data ranges accessed by a query. We base the routing decision on these signatures: the coordinator approximates the caches of the components by the signatures of the recently executed queries. The coordinator arranges its input queue so that the query promising the highest benefit from caching effects is processed first. It routes this query to that component with the largest signature overlap. In a quantitative evaluation using a realistic set of OLAP queries, we have shown that cache approximation based query routing can more than halve mean response times. Note that this result is a very general one. On the one hand, the signatures can be generated for arbitrary conjunctive SQL queries. On the other hand, our approach respects that components may be black-boxes.

References

[1] M. J. Carey, M. Livny, and H. Lu. Dynamic task allocation in a distributed database system. In *Proceedings of the ICDCS'85*, pages 282–291, May 1985.

[2] W. W. Chang and H.-J. Schek. A signature access method for the starburst database system. In *Proceedings of VLDB'89*, pages 145–153, 1989.

[3] T. P. P. Council. TPC-R benchmark specification rev. 1.0.1. Technical report, Transaction Processing Performance Council, July 1999.

[4] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of VLDB'96*, pages 330–341, 1996.

[5] P. M. Deshpande and J. F. Naughton. Aggregate aware caching for multi-dimensional queries. In *Proceedings of EDBT2000*, pages 167–182, 2000.

[6] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of SIGCOMM'98*, 1998.

[7] M. J. Freely et al. Implementing global memory management in a workstation cluster. In *Proc. 15th ACM Symp. on Operating System Principles*, 1995.

[8] J. Gray. How high is high performance transaction processing? Presentation on the 1999 HPTS Workshop, Asilomar, CA, Oct. 1999.

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

[10] U. Röhm, K. Böhm, and H.-J. Schek. OLAP query routing and physical design in a database cluster. In *Proceedings of EDBT2000*, pages 254–268, 2000.

[11] P. Scheuermann, J. Shim, and R. Vingralek. Watchman: A data warehouse intelligent cache manager. In *Proceedings of VLDB'96, Bombay, India*, 1996.

[12] J. Shim, P. Scheuermann, and R. Vingralek. Dynamic caching of query results for decision support systems. In *Proc. of SSDBM*, pages 254–263, 1999.

[13] M. Sinnwell and G. Weikum. A cost-model-based online method for distributed caching. In *Proceedings of ICDE'97, Birmingham, UK*, 1997.

[14] A. Thomasian. A performance study of dynamic load balancing in distributed systems. In *Proceedings of ICDCS'87, Berlin, Germany*, pages 178–184, 1987.

[15] S. Venkataraman, J. F. Naughton, and M. Livny. Remote load-sensitive caching for multi-server database system. In *Proceedings of ICDE*, 1998.

[16] P. Yu, D. Cornell, D. Dias, and B. Iyer. Analysis of affinity based routing in multi-system data sharing. *Performance Evaluation*, 7:87–109, 1987.