A Document Engine on a DB Cluster

Torsten Grabs Klemens Böhm Hans-Jörg Schek

Database Research Group – Institute of Information Systems
ETH Zentrum, CH-8092 Zürich, Switzerland
{grabs,boehm,schek}@inf.ethz.ch

Abstract

Requirements on document engines include low query response time and freshness of index data. Our solution to the problem is based on a DB cluster consisting of PCs, each of them running an off-the-shelf DBMS. The specialty of our approach is to allow concurrent execution of insertion and retrieval on the same data. Our technique is based on decomposing and parallelizing insertion and retrieval requests, while at the same time guaranteeing correctness by an additional high-level transaction manager. An important design decision with our architecture is how to assign data to the different components, and we compare several such alternatives. The speed-up obtained is surprisingly good. The cluster-based architecture nicely defuses the bottlenecks occurring with a single-component system.

1 Introduction

Document search engines must cope with very large document collections, both on the internet or in enterprise-wide intranets. These document collections keep growing at a high rate in size and number. At the same time, when querying such collections, users more and more expect up-to-date information that reflects recent modifications to the document collection or related meta-data. Thus, requirements on document search engines are not only low query response time, but also freshness of the index data [9] by allowing concurrent document insertion and retrieval. The common approach that there are two versions of a search engine one of which is used to answer queries while the other one is updated, does not exactly meet both of these requirements. But so far this has been the only means to ensures acceptable query response times.

Within the PowerDB project, the Database Research Group at ETH Zurich offers a solution to this urgent problem. The PowerDB project aims at investigating the power of a DB cluster, i.e. a cluster of commodity hardware plus software components together with off-the-shelf database systems. Previous work of our group [8] has shown that composite multi-level transactions [1, 10] lead to significant performance gains for document search engines based on a multi-processor relational database system. As opposed to [8], we deploy those techniques in clusters of workstations, where an instance of a relational database system is running on each node of the cluster. Furthermore a "coordinator" node – without sacrificing serializability – decomposes client requests into short (sub-)transactions that are routed to the appropriate component node in the cluster. That prevents us from the performance drawbacks that normally occur with managing redundant data [5].

In a case study, we have implemented those request decomposition and parallelization techniques for document search engines supporting boolean retrieval. We have investigated alternatives how to distribute the data among components of the cluster. We evaluate these alternatives for different numbers of components and expose these configurations to different workload patterns.

Our experimental findings from the prototype are very promising, they are better than what one would expect. Consider a request whose processing lasts one time unit in a single-component system. One might expect that processing this request on n components in parallel lasts $1/n$ time units at best. But in the case of high workloads, the speed-up is even better than $1/n$.

Another positive observation is that both document retrievals and insertions can run concurrently on the same system. Nevertheless, both services yield response times that allow for inter-active usage. That means that the problem of outdated index data literally does not exist: a retrieval request operates on an index that represents the state of the document collection at the start time of the request.

The idea to enhance search engine performance by using clusters of workstations has been applied before: The HotBot search engine ([3, 6]) statically partitions the data among the workstations. But global insertion and retrieval transactions remain unchanged, there is no decomposition and parallelization of smaller subtransactions. The latter techniques, in combination with full ACID properties for concurrent insertion and retrieval on the same system, constitute the major contribution of our approach. Already [4] has identified that it is crucial for commodity clusters to 'break the computation into small jobs' at the semantic level and to execute these in parallel. Furthermore, it has been known for a long time that transaction boundaries are crucial with regard to system performance [2]. Another approach contrary to our proposal is to gain performance by sacrificing serializability. [7] applies this to concurrent document retrieval and insertion.

The remainder of this paper is organized as follows: In Section 2, we discuss the document search engine architecture. Section 3 then reports on the application of PowerDB decomposition and parallelization techniques to news messages as an example of documents. Section 4 discusses the data placement alternatives that we investigated. We present our experiments in Section 5 and draw conclusions and sketch future work directions in Section 6.

2 Architectural Overview of the PowerDB Document Search Engine

In our previous work [8], we have shown that composite transactions lead to significant performance gains for document search engines based on a *multiprocessor relational database system*. In the PowerDB project we are investigating these techniques in a *cluster of workstations*, where an instance of a relational database system is running on each *node* of the cluster.

A coordinator-component architecture hides this system topology from clients. Clients connect to a distinguished node – the *coordinator*. The other nodes – the *components* – only interact with the coordinator. Only the components store (document-related) data, not the coordinator. The coordinator offers a service interface for document insertion and retrieval processing. This processing is as follows: after submission of a request, the coordinator decomposes it into sub-requests. All sub-requests of the same service invocation can run in parallel at different components if the data is distributed appropriately in the cluster. Thus, after decomposition and parallelization, the coordinator routes the sub-requests to the components. After processing the sub-requests, the components return their (partial) results. The coordinator composes an overall result and returns it to the client.

Within this process, the coordinator ensures atomicity of each service invocation and isolation of concurrent service invocations at the semantic level. In this current context, isolation means that a document insertion and a query cannot run concurrently if they conflict, i.e. the document inserted is in the query result. With a query with a conjunctive combination of query terms (boolean AND combination), this is only the case, iff this document contains all terms from the query. Applying this notion of conflict allows to start concurrent sub-requests of non-conflicting requests even if conflicts at the database page level occur.

On the component side, the relational database system ORACLE ensures durability of the sub-requests processed. We administer the distributed processing on the cluster with the transaction processing monitor (TP monitor) product TUXEDO from Bea Systems. Actually, we only apply a small subset of TUXEDO features, namely asynchronous remote service invocations, FML buffers (fielded buffers for data transmission) and routing capabilities. We have found that applying a full-fledged TP monitor product does not introduce much overhead, compared to a conventional two-tier solution with ORACLE Net8 connections. Our decomposition and parallelization techniques lead to small request sizes where performance differences between the two alternatives are not significant.

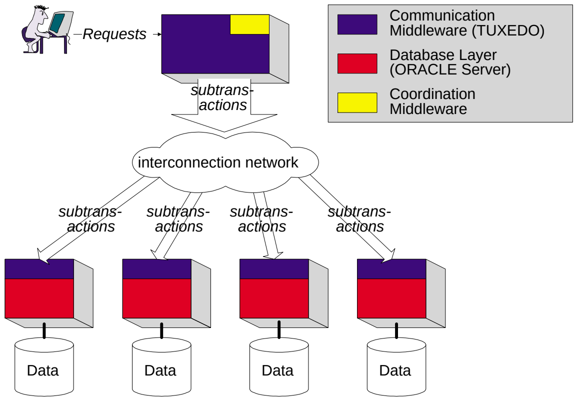

Figure 1: System architecture

Note that we do not apply XA's 2PC protocol delivered with TUXEDO. As explained before, atomicity and isolation at the service level are proprietary extensions of the TP monitor at the coordinator. Figure 1 depicts the coordinator-component topology and the software systems used.

3 Request Decomposition and Parallelization

We will use news messages as an example for documents in order to describe the mapping and decomposition. More general types of documents can be mapped in a similar way.

Database Mapping News messages contain the fields *author, date, ..., subject* and *body*. Then a relation A that stores the raw document text has the attributes `docid, author, ..., body`. For the *subject* and the *body* fields, a relation B_1 resp. B_2 with the attributes `termid` and `docid` represents the index. Hence, A and B_i compose a $1 : n$ relationship.

Given this mapping to a relational schema, we now explain how to express document insertion and retrieval with SQL.

Example 1: Suppose that the relations A and B_1 reside in a single database. Then the following SQL statement retrieves documents that contain the words d_1 and d_2 in the subject field:

```
SELECT * FROM A WHERE docid IN
(SELECT docid FROM B(1) WHERE term = d1)
   INTERSECT
(SELECT docid FROM B(1) WHERE term = d2)
```

The expression for insertion is as follows:

```
BEGIN TRANSACTION
INSERT INTO A VALUES(a)
INSERT INTO B(1) VALUES(docid, d(1))
INSERT INTO B(1) VALUES(docid, d(2))
END TRANSACTION
```

◇

Request Decomposition Generally, decomposition of these services expressed in SQL is as follows: The retrieval for each term of the query constitutes a separate SQL sub-statement. Decomposition means now that each of these SELECT statements has its own transaction boundaries. The overall result of relevant documents is the intersection of the docid sets returned from these sub-requests.

Decomposition of insertion service calls is as follows:

1. For each relation modified by an insertion there is a separate subtransaction.

2. To enhance the parallelism (explained below), we can further decompose each of those subtransactions into smaller subtransactions. Each of those sub-subtransactions inserts a number of tuples into the relation. This number is a parameter that depends on system characteristics such as the commit cost.

Example 2: Consider an insertion of a document with text t where t contains the terms d_1 and d_2 in the subject field. Then, in principle, there are the following subtransactions:

```
BEGIN TRANSACTION
BOT;INSERT INTO A VALUES(t);EOT
BOT;INSERT INTO B(1) VALUES(docid,d(1));EOT
BOT;INSERT INTO B(1) VALUES(docid,d(2));EOT
END TRANSACTION
```

◇

In this example, we have decomposed the insertion into three SQL insert transactions. As said before, in the practical evaluations we did not push decomposition to such extremes.

Sub-Request Parallelization Having accomplished the decomposition into sub-requests, it is easy to see that (nearly) all sub-requests of the same global service invocation can run in parallel. This leads to significantly higher degrees of intra-transaction parallelism compared to the long transaction without these techniques. Furthermore, our semantic conflict test in combination with this decomposition and parallelization leads to higher inter-transaction parallelism because less lock contention occurs.

To facilitate true parallelism for these sub-requests, data placement is crucial. This means that different disks and processors process these parallel requests. We discuss this in the following section.

4 Data Placement

Data placement is an essential issue to ensure balanced workloads on the nodes of the cluster. This is because the coordinator routes sub-requests depending on the data placement. Hence, a bad placement decision may lead to skewed workload distributions and increase response times.

Recall the $1:n$ relationships that characterize the schema of our document-to-database mapping. For this schema, we have identified three different placement alternatives:

DISTAB: DIStributing complete TABles,

HASHLOC: HASHing tuples with dependency LOCality, and

HASHCONS: HASHing tuples CONSecutively to component databases.

The first placement scheme assigns each relation in the schema to a different component database and is easy to implement. The following placement alternatives (HASH*) distribute tuples at the data level, as opposed to the schema level with DISTAB. The component where a tuple is stored depends on a hash value h of the document identifier. HASHLOC preserves dependency locality. That means that the document and its index data reside at the same component system. HASHCONS assigns tuples consecutively to component systems. In other words, if the tuples for relation A of a given document reside at a component h, then the index entries in relation B_i of this document are stored at component $h + i \bmod n$ (n : number of components).

5 Experiments

This section describes the experiments we have carried out in order to evaluate the different placement alternatives with our news search engine prototype. The three following independent dimensions characterize our experiments:

Placement Alternatives: We have considered the placement alternatives DISTAB, HASHLOC and HASHCONS.

Configuration: In order to assess the speed-up properties of the placement alternatives, we tested the system with 3 workstations (2 components and the coordinator) and with 5 workstations (4 components and the coordinator). To have a reference point, we have run experiments on a monolithic configuration with only one workstation.

Workload Patterns: Performance of the placement alternatives depends on the workload pattern. We denote the system workload with a vector (a, b) where a and b represent the number of concurrent insertion and retrieval services, resp.

Our current system configuration uses PCs with one 233 MHz Pentium Processor, 128MB main memory and an interface to a network with a data transmission rate of 10 Mbit/sec. All of those workstation systems are equipped with the Microsoft Windows NT Server 4.0 operating system software. We configured the database systems with tablespaces on an IDE and a SCSI disk drive. The database buffer holds a maximum of 550 blocks of 2K size. Each measurement started with an initial collection size of 2000 documents in the database. With high workloads, we have inserted roughly another 2000 documents. Concurrently to the insertions, we have run the query streams with one to five arbitrary query terms from the complete collection.

In our discussion, we focus on response times at the client interface. Figure 2 depicts the response times for a monolithic configuration with only one node. This node is both the coordinator and the only component of the system. Hence, data placement is trivial with this configuration.

The values from Figure 2 show that response times increase approximately linearly from low to high workloads. With increasing workloads, more services compete for restricted system resources, especially disk I/O. Note that for high workloads insertions last on average unbearable 45 seconds per document. But even for the low workload $(1, 1)$ insertion response times are rather high with 5 seconds on average. This is because the restricted disk capacities do not allow for parallel processing of the many sub-requests generated from the service call.

Figure 3 and 4 show the insertion response times for a large configuration with five nodes – one coordinator and four component nodes. The three series denote the different placement alternatives previously discussed. The values for insertions show that the absolute response times decrease from 45 seconds in the monolithical case to less than 8 seconds for DISTAB. Both HASH placement alternatives show an even better behavior that yields insertion response times of less than 4 seconds per document for high workloads. The reason is that now the same workload distributes among four component nodes of the cluster. Recall our initial consideration from the introduction that for an increase n of nodes, response times decrease at best by a factor of $1/n$. Then we would expect a response time around $45/4 = 11$ seconds. Hence, our experiments show that with HASH placement one can achieve even better performance improvements by increasing the number of components in the cluster when this increase significantly reduces workload contention on the nodes. We assume that this is due to reduced I/O contention on the relatively small number of disks in each workstation. Another plausible explanation could be that the log latch constitutes the bottleneck in the monolithical case. In the clustered configuration, we have as many logs as there are component systems. Nevertheless, this is still an open issue and we will explore this.

Response times for retrieval decrease by a similar factor. Hence, this configuration now allows for an inter-active usage of the services. Both HASH

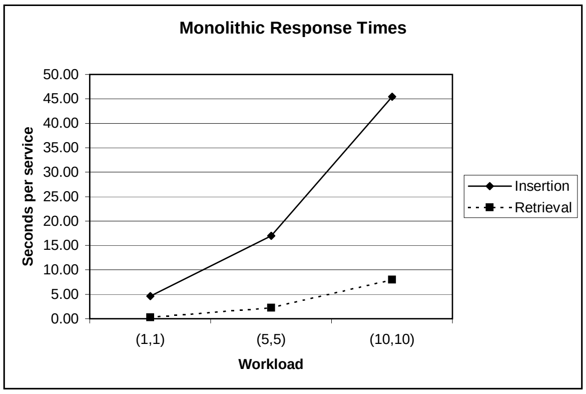

Figure 2: Response time values in a monolithic configuration

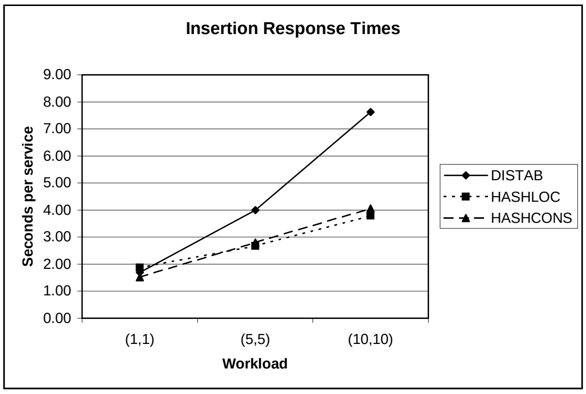

Figure 3: Insertion response time with 4 component subsystems

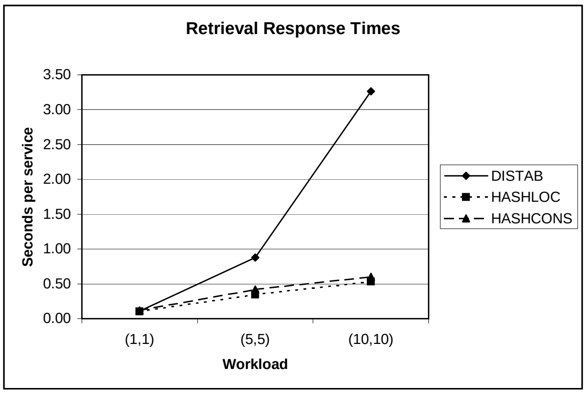

Figure 4: Retrieval response time with 4 component subsystems

workload	DISTAB	HASHLOC	HASHCONS
(1, 1)	2.8	2.5	3.1
(5, 5)	4.2	6.3	6.1
(10, 10)	6	12	11.2

Table 1: Insertion speed-up from monolithic to large configuration

placement alternatives outperform the DISTAB alternative because table sizes are not equal with DISTAB. Hence, DISTAB's distribution of complete tables leads to skewed workloads on the cluster. Both HASH placement alternatives do not suffer from this drawback.

In addition to the figures discussed above, Tables 1 and 2 show the speed-up factors achieved by increasing the number of components from 1 in the monolithical case to 4. Note that for large workloads this decreases response times by an order of magnitude for both HASH placement architectures. Further, we observe that the retrieval speed-up with the DISTAB alternative does not increase with higher workloads due to its skewed data distribution.

6 Conclusions and Future Work

In the PowerDB project we combine concurrency control at the semantic level and sophisticated request decomposition and parallelization mechanisms in DB clusters.

In a case study, we have implemented a news document search engine with PowerDB techniques and tested this engine with different data placement alternatives. Our findings show, that HASH placement yields the best response time. For high workloads, these placement alternatives reduce response times by an order of magnitude when increasing the number of components by 4.

Our evaluation has not yet covered all interesting aspects. Currently, we investigate whether and when the coordinator can become a bottleneck. We further want to compare our approach to the conventional XA/2PC solution in quantitative terms. Finally, in parallel to document support, we apply our PowerDB architecture to the traditional TPC-C and TPC-D benchmarks.

workload	DISTAB	HASHLOC	HASHCONS
(1, 1)	2.7	2.7	2.5
(5, 5)	2.5	6.4	5.3
(10, 10)	2.5	15	13

Table 2: Retrieval speed-up from monolithic to large configuration

References

[1] G. Alonso, A. Feßler, G. Pardon, and H.-J. Schek. Transactions in stack, fork, and join composite systems. In *Proc. ICDT'99, Jerusalem*, pages 150–168, 1999.

[2] M. J. Carey, R. Jauhari, and M. Livny. On transaction boundaries in active databases: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):320–336, 1991.

[3] A. Fox, S. G. Y. Chawathe, E. Brewer, and P. Gaulthier. Cluster-based scalable network services. In *Proc. of the SOSP'97, St. Malo, France*, 1997.

[4] J. Gray. Super-servers: Commodity computer clusters pose a software challenge. In *Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 30–47, 1995.

[5] J. Gray, P. Helland, P. O'Neill, and D. Shasha. The dangers of replication and a solution. In *Proc. of the ACM SIGMOD Conf.*, pages 173–182, 1996.

[6] Inktomi Corp. The Inktomi technology behind HotBot. Technical report, Inktomi Corp., http://www.inktomi.com/Tech/-CoupClustWhitePap.html, 1996.

[7] M. Kamath and K. Ramamritham. Efficient transaction support for dynamic information retrieval systems. In *Proc. of ACM SIGIR*, 1996.

[8] H. Kaufmann and H.-J. Schek. Extending tp-monitors for intra-transaction parallelism. In *Proc. of the 4th Int. Con. on Parellel and Distributed Information Systems*, 1996.

[9] S. Kirsch. Infoseek's experiences searching the internet. *SIGIR Forum*, 32(2):3–7, 1998.

[10] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, 1991.