Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institute of Information Systems *Database Research Group*

CS Technical Report No. 340, April 25, 2000

A Parallel Document Engine Built on Top of a Cluster of Databases

– Design, Implementation, and Experiences –

Torsten Grabs, Klemens Böhm, Hans-Jörg Schek

Department of Computer Science
Institute of Information Systems
ETH Zurich
CH-8092 Zurich, Switzerland
email: {grabs, boehm, schek}@inf.ethz.ch
http://www-dbs.inf.ethz.ch

A Parallel Document Engine Built on Top of a Cluster of Databases – Design, Implementation, and Experiences –

Torsten Grabs Klemens Böhm Hans-Jörg Schek

April 25, 2000

Abstract

We report on the implementation and evaluation of a document engine that supports many parallel search and concurrent insertion requests efficiently and that is scalable to growing numbers of such requests. We use a cluster of commodity database systems in a shared nothing architecture. We deploy previous results on multi-level transactions and decompose a service request into short parallel database transactions. A coordinator, implemented as an extension of a transaction processing monitor, routes the short transactions to the appropriate database system in the cluster, depending on the data distribution that we have chosen. We have paid much attention to the design and implementation of the coordinator to avoid that it becomes a bottleneck. That means that we implemented auxiliary functionality such as term extraction as services and distribute them over the cluster. Extensive experiments show the following: (1) A relatively small number of components already suffices to cope with high workloads. (2) The coordinator of the database cluster has minimal impact on CPU resource consumption and on response times. E.g., the response time overhead of the coordinator is in the order of milliseconds while the response time for retrieval and insertions remains within seconds even with 100 parallel search or insertion streams. This is rather unexpected since the coordinator performs signature-based predicate locking and writes additional logging information. We conclude that a database cluster with a coordinator on top is a good scalable infrastructure for complex application services.

1 Introduction

Administering collections of documents efficiently is an important issue. According to Kirsch, chairman of Infoseek, four citeria are crucial for future search engines [18]. These are: query response times, index sizes, cost of hardware and freshness of data in the index. Our objective is to build a document search and indexing engine that meets these criteria. In particular, it is difficult to reconcile the criteria 'query response time' and 'freshness of index data'. To tackle this problem, we apply database technology in an innovative way: we investigate in quantitative terms a solution that combines multi-level transactions and parallelization techniques, and that is built on top of a cluster of databases. In our terminology, a cluster of databases is a PC cluster where each PC runs an off-the-shelf DBMS. Building a document engine on top of such a cluster broadens the way database technology is used. In previous work [17], we have already shown that multi-level transactions lead to significant performance gains in a multiprocessor database system. This is because a single document request is decomposed into many parallel database transactions. But, with such a monolithic system, opportunities for scaleup and speedup are limited. A cluster-based architecture in turn is promising with respect to *"scale-out"* [20], that is, adding a new component to achieve better performance.

In more detail, we pursue a *coordination-based architecture*, i.e., a cluster node either is the *coordinator* or a *component*. As part of a transaction, a client submits requests for *document services* to the coordinator. Bundling services into transactions reflects that several documents may belong together on a logical level, and such a group of documents should only be visible as a whole. Examples of document services are insertion of a new document, deletion, update, keyword-based search or search using complex operators such as the proximity operator. With a straightforward design of a cluster-based document engine, the coordinator would accept the requests and execute the *document-specific functionality* such as term extraction, stopword elimination and stemming. Then, the coordinator would map the document services to SQL statements and submit them to the different component database systems. Two-phase-locking at the components in combination with a two-phase-commit protocol at the coordinator would implement serializability and atomicity. But such a design imposes a heavy load on the coordinator. It would soon become a bottleneck. Instead, a coordination-based approach requires as little centralized processing as possible. Our system architecture reflects this. We have investigated which processing needs not be done at the coordinator, and our design has shifted it to the components. The only centralized tasks are concurrency control and logging of global transactions. Their implementation in turn is very efficient, as our experiments show.

We have built a document engine with such a thin coordinator, and we have evaluated it extensively. The implementation uses off-the-shelf components such as a transaction monitor and relational database systems. In more detail, the contributions of this paper are the following ones:

- We show how to decompose the functionality of a document engine into transaction monitor services. Our approach allows to replicate the services so that they run in parallel on an arbitrary number of components.

- We present the design of the coordinator, guaranteeing atomicity and correctness of concurrent execution of document service transactions. We emphasize the efficient implementation of a signature-based two-phase-locking protocol for document services.

- With a coordinator-based architecture, the coordinator eventually is a bottleneck. We have run experiments to find out if this is the case in practice. It turns out that concurrency control and logging within the coordinator impose only minimal overhead even with very high request loads.

- We evaluate different data placement alternatives for different numbers of components and for different workload patterns. An important observation is, that response times are interactive even for high workloads with relatively few components already.

- We show experimentally that bundling service requests to transactions does not reduce performance. This means that a discussion whether this current application scenario requires transactions is simply superfluous, as the performance impact of such a feature is minimal.

The remainder of this paper is organized as follows: Section 2 discusses related work. In Section 3, we discuss the service model, parallelization, concurrency control, logging and different data placement alternatives. The section also describes the architecture and discusses the implementation in detail. Section 4 presents the experimental setup. Section 5 discusses our experiments. Section 6 concludes.

2 Related Work

In [15, 11], the authors investigate generic aspects of implementating different services on a PC cluster. They conclude that PC clusters are well suited when correctness is not important. Correctness corresponds to the notion of serializability from transaction theory [5]. An efficient way to implement correctness in combination with high parallelism are multi-level transactions [1, 24, 25]. [2] as well as [13, 17] describe document services and architectures of document engines that are based on multi-level transactions. [2] does not provide an evaluation, and the article does not address scalability issues. This in turn is one of our main contributions.

Another approach is to weaken the correctness criterion of serializability. [16] shows that lock contention in traditional database transaction management unnecessarily restricts the parallelism of insertion and retrieval requests and this leads to performance degradation. In [16], insertions happen in batch update runs. But a transaction reordering technique ensures that queries take also the most recent updates in account. Queries are processed after the relevant document has been inserted in a concurrent update. Hence, the query retrieves the relevant document. Their approach does not decompose and parallelize document service requests. [19] discusses the concurrency control mechanism of the SPIDER search engine. The authors argue that leaving aside correctness is acceptable to some degree with probabilistic retrieval. Hence, this work can be considered as a “non-boolean” version of a conflict test in multi-level transactions.

There is a vast amount of literature about storage structures for information retrieval. [12] gives an overview about this topic. [6, 23] address ’freshness of index data’ with techniques that allow for incremental updates of the index. Conceptionally, any transactional subsystem that implements efficient indexing can become a component of our system. Similar to our approach, [14] maps the index onto relational database tables. [14] has shown that the boolean and the extended boolean retrieval model with proximity search and phrase operators nicely map to standard SQL. This also holds for the vector-space model [21]. The authors also show that this yields competitive response times. But their work as well as the full-text document retrieval benchmark [9] is restricted to read-only and single query workloads. We for our part take updates into account explicitly and provide the infrastructure to process them efficiently. Moreover, our approach allows insertions of new documents to run concurrently to retrieval without sacrificing the ACID properties.

Research on parallel database systems has investigated hash partitioning for data placement [3, 7, 10, 26]. We take over hash partitioning techniques to distribute document and index data.

3 Architecture of the Document Engine

To explain our architecture, Subsection 3.1 shows how multi-level transactions increase parallelism. The following subsections then describe our search engine in more detail. Our running example are newsgroup postings. But our approach is applicable to arbitrary document types.

3.1 Overview of the Concurrent Service Execution

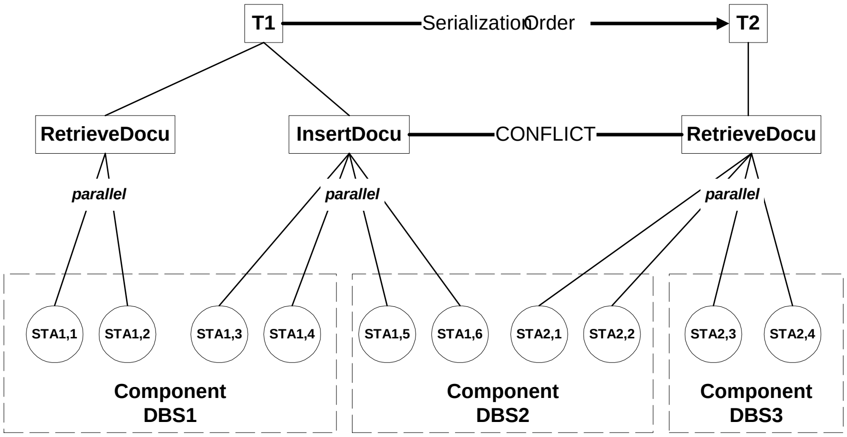

Figure 1: Inter-transaction- and intra-transaction parallelism for document services

With our approach, services of different transactions should run concurrently if they do not conflict, i.e., if a document to be inserted does not qualify (is in the result set) for any query currently executed. In case of conflicts, services are executed sequentially, as Figure 1 shows. This ensures serializability. In addition to this, our approach guarantees atomicity by logging. Services in turn consist of subtransactions (STA). STAs can execute in parallel on different machines, as the bottom half of the figure shows. In our terminology, inter-transaction-parallelism is concurrency of services that belong to different transactions, and intra-transaction-parallelism are subtransactions, e.g., of a service request, running in parallel. The hypothesis is that both inter-transaction-parallelism and intra-transaction-parallelism yield significant performance gains. Essentially, our approach deploys two-level transaction management to transactions that comprise document services as operations. In our terminology, the *second layer transaction manager* implements the transactions for document services.

3.2 Document Services

We briefly introduce the retrieval model and the document services. Then we show how we have mapped this model to a relational database scheme. Such a mapping has the advantage that access with standard SQL to relational attributes can coexist with access by document services to the textual data. Document services are for example insertion of a new document, update of a document, deletion by document id or deletion of all documents that qualify for a given query. Search services are keyword-based, or comprise complex operators such as the proximity operator. Search services may also support different retrieval models such as the vector-space model or probabilistic retrieval. We restrict the discussion to the typical services of insertion and keyword-based boolean retrieval. This focuses on the central problem of processing insertions concurrently to retrieval.

Retrieval Model. Our document engine supports the boolean retrieval model [21]. With boolean retrieval, documents consist of fields. For instance, an e-mail document contains a sender field. A boolean query is a conjunction of predicates of the form *contains(field,pattern)*, with the following semantics: each document that qualifies must contain *pattern* in its *field*. This model is sufficient for most queries users normally pose against search-engines. With Infoseek, the average length of queries is 2.2 words [18], i.e., users do not favor complex queries. Hence, we have left aside advanced query features such as word distance operators, disjunctive forms of the pattern or non-boolean query facilities. [14] discusses mappings to standard SQL for more complex query functionality, including vectorspace model queries that we can support in our architecture as well.

Service Interface. The insertion and retrieval services have the following interfaces:

- `Insertion Service:` $integer\ InsertDocu(text)$
  The input parameter is the document text. It returns a status flag.

- `Retrieval Service:` $\{doctitles\}\ RetrieveDocu(\{(field, term)\})$
  The input parameter is a set of predicates. It returns the titles of qualifying documents.

Both services comprise term extraction, stemming and stopwording as well as decomposition into SQL-subtransactions. Decomposition into SQL-subtransactions consists of two steps, as described subsequently.

Database Mapping. We now explain the database mapping by means of our running example. The documents contain the fields *author, date, subject* and *body*, among others. With our mapping, relation A stores the raw document text. It has the attributes `docid, author, ..., body`. For each field that may occur in a predicate, there is a relation B_i with attributes `termid` and `docid`. The attribute `termid` is the integer representing the stem of a word contained in document `docid`. We map the words to integers because this allows for more efficient index structures [14]. In other words, relation B_i implements the index over the field.

Given this mapping to a relational schema, we now explain how to express document insertion and retrieval with SQL. Recall that insertion and retrieval invocation occur in a global transaction. In the following example, we denote the transactions boundaries with explicit `BEGIN/END GLOBAL TRANSACTION` (`BGT`/`EGT`).

Example 1: Suppose that the relations A, $B_{subject}$, and B_{body} – subsequently referred to as B_1 and B_2 – reside in a single database. Then the following service call and its corresponding SQL statement retrieve documents that contain the words d_1 and d_2 in the subject field and d_3 in the body field:

```
BGT
res=RetrieveDocu(
       (’subject’,d1),
       (’subject’,d2),
       (’body’,d3))
...
EGT
```

```
BGT
SELECT * FROM A, B1 b1a, B1 b1b, B2
WHERE b1a.termid = d1
AND b1b.termid = d2
AND B2.termid = d3
AND A.docid = b1a.docid
AND b1a.docid = b1b.docid
AND b1b.docid = B2.docid
...
EGT
```

The expression for insertion of such a document with text t is as follows:

```
BGT
res=InsertDocu(
      ’subject: d1, d2’)
...
EGT
```

```
BGT
INSERT INTO A VALUES(t)
INSERT INTO B1 VALUES(docid, d1)
INSERT INTO B1 VALUES(docid, d2)
...
EGT
```

◇

Note that the `INSERT INTO` statements in the latter expression can appear in any order since there is no flow of information between the statements. In fact, they could also execute in parallel. Based on this observation, we describe the decomposition of the insertion service into parallel subtransactions in the following.

Service Parallelization and Decomposition. For insertion service calls, parallelization is as follows:

1. For each relation modified by an insertion there is a separate subtransaction.
2. We can further decompose each of those subtransactions into smaller subtransactions. Each such sub-subtransaction inserts a number of tuples into the relation. Those sub-subtransactions can run in parallel, too.

We implement these subtransactions as independent database transactions that may commit or abort independently. Subsection 3.4 explains how we guarantee atomicity.

Example 2: Consider again Example 1. Assume that an insertion inserts a document with text t. Let this text contain the terms d_1 and d_2 both in the subject and the body fields. Then the parallel subtransactions – bracketed by `BOT` and `EOT` – are as follows, according to Parallelization Steps 1 and 2:

```
BGT
DO IN PARALLEL {
 BOT; INSERT INTO A VALUES(t); EOT
 BOT; INSERT INTO B1 VALUES(d1); EOT
 BOT; INSERT INTO B1 VALUES(d2); EOT
 BOT; INSERT INTO B2 VALUES(d1); EOT
 BOT; INSERT INTO B2 VALUES(d2); EOT
 ...
}
EGT
```

◇

3.3 Ensuring Correctness of Concurrent Document Service Transactions

This subsection describes the concurrency control mechanisms of our system.

With document engines, global transactions consist of one or more document service requests. The system cannot execute transactions concurrently in an arbitrary manner because of conflicts [25]. For our document engine, an insertion service s_1 and a retrieval service s_2 conflict if and only if the document of s_1 qualifies for the query of s_2 and both services belong to different transactions. Serializability of a schedule yields correct executions.

We have applied a semantic two-phase-locking protocol. Each service invocation of a transaction dynamically acquires the needed locks and releases them at the end of the transaction. For documents, this works as follows: we represent each request by a bitstring signature [8]. In more detail, we map each non-trivial stem of the document or the query to a signature bit. The bit is then set. Recall that a query is a conjunction of predicates. Hence, all words from all patterns of the query must appear in a qualifying document. If an insertion service `InsertDocu` is invoked then we compute the signature sig_{docins} for this service. Now, let $\wedge_2$ denote the bitwise AND operator and $=_2$ the bitwise comparison. If the equation

$$sig_{docins} \wedge_2 sig_{query} =_2 sig_{query}$$

holds for any $query$ currently running then we assume a conflict. `InsertDocu` is delayed until the transactions with conflicting services have finished. The concurrency control component keeps delayed requests in waiting queues, one for insertion and one for retrieval. When a service terminates, the concurrency control component checks the queues and schedules execution of requests that are not longer in conflict with a concurrent request. The analogous considerations hold if a client submits a new query to the system. Note that there may be false alarms for conflicts. This happens if different words from the query and the document being inserted are mapped onto the same bit position. In practice, long signatures help to prevent from such situations. We use signatures with $2^{14} = 16384$ bits. We work with these huge signatures in order to not underestimate the concurrency control overhead. Even this conservative approach does not lead to a performance problem, as our experiments from Section 5 on the scheduling overhead show.

Given this semantic concurrency control at the service level, we do not need conflict detection at the level of SQL-subtransactions (bottom level in Figure 1). While conflicts at the SQL-statement level are possible (pseudo-conflicts), the semantic conflict test ensures that they do not lead to incorrect executions at the document service level.

3.4 Atomicity of Document Service Transactions

Logging is necessary to ensure atomicity of insertion services in case of failures. Note that we decompose one single insertion request into many parallel database transactions that may commit or abort independently. Our system must guarantee atomicity on two levels: (1) atomicity of a single service request, and (2) atomicity of global transactions that comprise more than one service request. To this end, different techniques have already been applied: [22] implements atomicity by compensation at the

SQL-level. That implies that for each insertion statement the log manager has to log the compensating SQL delete statement for each tuple inserted. With such an approach, logging yields a big overhead. In [17], the authors tackle this problem as follows. Their implementation first inserts the document text and then it updates the index data. Maintenance of the index tables starts only if the insertion of the document has been successful. This allows for forward recovery of insertion services, but parallelism is restricted. Our approach favors parallelism, there is no restriction on the order of subtransactions, but the information that we write to the log is minimal. In detail, we apply the following technique.

To guarantee atomic service executions, i.e., (1), we assume that the components are subsystems with atomic transactions. This is the case with nearly all off-the-shelf database systems. Futhermore, we do not distinguish between semantic aborts due to constraint violations and technical aborts because of deadlocks, e.g. The critical case in our setup is now the one where one subtransaction of the service successfully commits and a second subtransactions aborts. In that case, the system has to compensate the effects of the already committed subtransaction. To this end, the insertion service writes a $(BOT, tid, docid)$-triple to the log before submitting any subtransactions. We will explain the tid parameter in the following paragraph. When all subtransactions terminate successfully at the components or are successfully compensated, the service logs a $(EOT, tid, docid)$-triple. Both these triples bracket the set of independent SQL subtransactions that an insertion service starts. Each tuple that is inserted carries this unique $docid$. Hence, compensation for a service simply deletes all tuples with that $docid$. A missing EOT marker for some $(BOT, tid, docid)$-triple means that such a compensation is required eventually. Logging BOT and EOT is the responsibility of each insertion service. Distributing these services among components also allows to distribute the log. Section 3.6 provides more information.

Now we discuss atomicity of global transactions that comprise one or more service requests, i.e., (2). To make a transaction tid atomic, the concurrency control logs (BGT, tid) and (EGT, tid)-pairs when the transactions starts and ends, respectively. The identifier tid links the service level log to the global transaction log at the coordinator. This allows to search the service level logs for this tid and to retrieve the set D of all $docid$s that have been inserted with that global transaction. If necessary, the coordinator then uses this information to rollback tid. To this end, it compensates each insertion of a document $docid$ that is in D.

These logging mechanisms provide information that also suffices for complete crash recovery. This article omits details on crash recovery for lack of space.

3.5 Data Placement Alternatives for Document and Index Tables

The parallelization techniques discussed above allow for any data placement scheme where a subtransaction runs on one component only. This avoids a distributed commit protocol such as 2PC. Ideally, we would like to route each subtransaction to a different component. This maximizes the degree of parallelism and helps to balance workloads of the component database systems. The placement alternatives that we discuss are the following ones:

- `DISTAB:` DIStributing complete TABles,

- `HASHLOC:` HASHing data items with dependency LOCality, and

- `HASHCONS:` HASHing data items CONSecutively to component databases.

The first placement scheme assigns each table to a different component database. That means all data items of entity type A are stored on the first component and all B_1 data items reside on the second component. Figure 2(a) illustrates this mechanism. This alternative is rather easy to implement. But with tables of different sizes workload distribution is expected to be suboptimal. Furthermore, with a limited number of tables this alternative does not scale for component numbers bigger than that. Hence, for lack of space, we do not present results for DISTAB in the following sections.

The other placement alternatives (HASH*) distribute data items at the data level, as opposed to DISTAB that distributes at the schema level. A hash value h of the document identifier identifies the component where a data item is stored. The first alternative HASHLOC preserves dependency locality. That means that the document and its index data reside at the same component h, as Figure 2(b) shows. The third alternative HASHCONS (Figure 2(c)) assigns data items of a single document to different components: 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 $(h + i) \bmod n$ where n is the number of components.

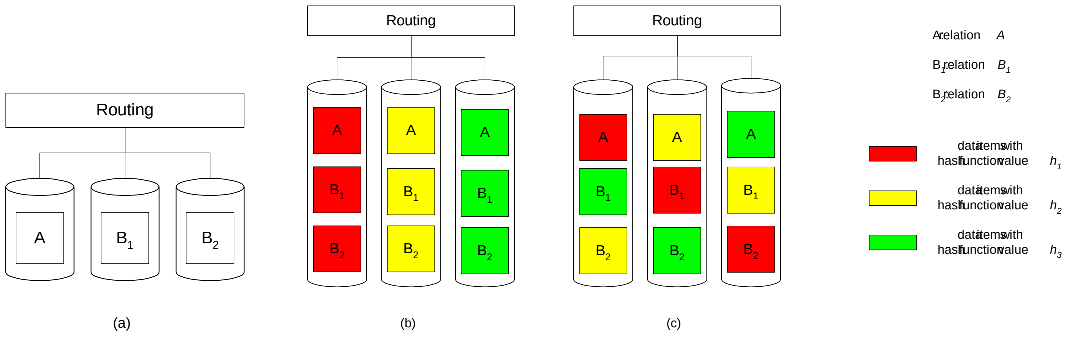
Routing
A
B1
B2
(a)
Routing
A
B1
B2
(b)
Routing
A
B1
B2
(c)
A: relation A
B1: relation B1
B2: relation B2
data items with hash function value h1
data items with hash function value h2
data items with hash function value h3

Figure 2: (a) Distributing tables (DISTAB) (b) Preserving dependence locality with Hashing (HASHLOC), (c) Hashing with consecutive distribution of data items (HASHCONS)

Routing of Subtransactions. For insertion services, data placement affects the routing of subtransactions to components as follows. Consider a subtransaction that inserts into a relation R.

1. With HASHLOC, all insertion subtransactions are sent to one component.

2. With HASHCONS, the hash value of the document ID identifies the component of a subtransaction that inserts into relation A. Other subtransactions that insert into relation B_i are sent to component $(h + i) \bmod n$.

Consider now the routing of subtransactions of retrieval services. With both HASH placement alternatives, the system sends the same subtransaction to each component.

3.6 Design and Implementation using Off-the-shelf Components

Components of our Architecture. We have implemented our system by extending a standard middleware product. In our setting, middleware functionality additionally comprises document processing, data dependent distribution of requests as well as semantic concurrency control and logging. Hence, we distinguish between standard middleware and our proprietary middleware extensions in the following.

In more detail, the *standard middleware* transmits requests and data items from the coordinator to the components. Furthermore, it helps to administer the cluster in terms of remote process startup etc. This functionality is part of various state-of-the-art middleware products.

Our *extensions of the middleware* in turn consist of two separate modules. The first module ir_mod provides the application logics as discussed above: term extraction from texts, stopwording, stemming and signature generation for texts. As decomposition of insertion requests is application-specific, it is also part of ir_mod. ir_mod also implements the different placement alternatives and does the logging at the service level.

The second module $coord\_mod$ carries out the actual coordination of transactions and service requests, i.e., the semantic conflict test. Furthermore, the $coord\_mod$ module maintains the global transaction log. An off-the-shelf software product that implements this functionality does not exist. So both modules are proprietary implementations.

With respect to component database systems, any relational database system that supports a C interface for SQL is applicable.

Services and Processes. To explain parallel execution of services, we distinguish between services and processes subsequently. A *service* is the implementation of a specific functionality, e.g., signature generation for texts, in the form of a binary executable. Starting such an executable at a component, yields a *process* of such a service. There can be many processes for one service, and the processes can even run in parallel at different components. There is a single process for $coord\_mod$ running at the coordinator. This is necessary in order to keep a global locking table for signatures. In contrast, there may be an arbitrary number of processes at different components for ir_mod functionality. We stress that signature generation is not part of the coordinator. Instead, any component can perform this task.

Table 1 and Table 2 summarize the most important services of our system. We distinguish between *inbound calls* and *outbound calls*: the first kind of calls invokes methods of our proprietary middleware extensions. With outbound calls, the middleware extensions invoke methods provided by other logical components.[1]

Interaction of Processes for Service Execution. Figure 3 shows the interaction of the processes for a global transaction that comprises a single call to the document insertion service.

The boxes at the top of the figure denote the different processes that participate in the processing of our example transaction. Note that the concurrency control process (”CC”) – the process for $coord\_mod$ – must reside at the coordinator node. All other processes can run at arbitrary nodes. For each process

[1] We use the following shorthands: TX – transaction, appl. –application.

module	*inbound service name*	*description*
coord_mod	INSSERV$(signature)$	Conflict test for an insertion
coord_mod	RETSERV$(signature)$	Conflict test for a retrieval
coord_mod	BGT(tid)	Begin of a global TX tid
coord_mod	EGT(tid)	End of a global TX tid
ir_mod	RETRIEVEDOCU$(\{(term, field)\})$	Retrieval appl. logic interface routine
ir_mod	INSERTDOCU(doc)	Insertion appl. logic interface routine
ir_mod	REQMANAGER	Manages requests for clients

Table 1: Inbound services

module	*outbound service name*	*description*
ir_mod	MRETDOC$(docid)$	Call SQL sub-TX retrieving a document text
ir_mod	MINSDOC$(docid, doc)$	Call SQL sub-TX inserting a document text
ir_mod	MRETDESC$(\{desc\}, field)$	Call SQL sub-TX retrieving descriptors $\{desc\}$
ir_mod	MINSDESC$(\{desc\}, field, docid)$	Call SQL sub-TX inserting descriptors $\{desc\}$
ir_mod	GENSIGN$(text)$	Generate the signature for $text$

Table 2: Outbound services

there is a bold vertical line when the service processes a request or waits for response from other services. The interaction of the processes is now as follows:

1. The client begins a new global transaction and requests a new transaction identifier tid from the concurrency control. The concurrency control writes the BGT marker to its log before it delivers the tid to the client.

2. The client submits the document together with the tid to a so-called *request manager process* (REQMANAGER) running at an arbitrary component. The request manager transparently manages all subsequent processing of the service request.

3. The request manager process submits the text to a signature server (SignServer) that generates the signature of the text (service GENSIGN). Signature generation requires to parse the complete document text, to extract terms, to eliminate the stopwords and to do the stemming.

4. The request manager submits the signature and the tid to the concurrency control (routine INSSERV). The CC checks for conflicts and blocks the process as long as there are conflicts with other ongoing document service transactions.

5. After admission by the scheduler, the request manager process forwards the request to an `InsertDocu` service process.

6. The `InsertDocu` service decomposes the insertion into parallel subtransactions, as described in

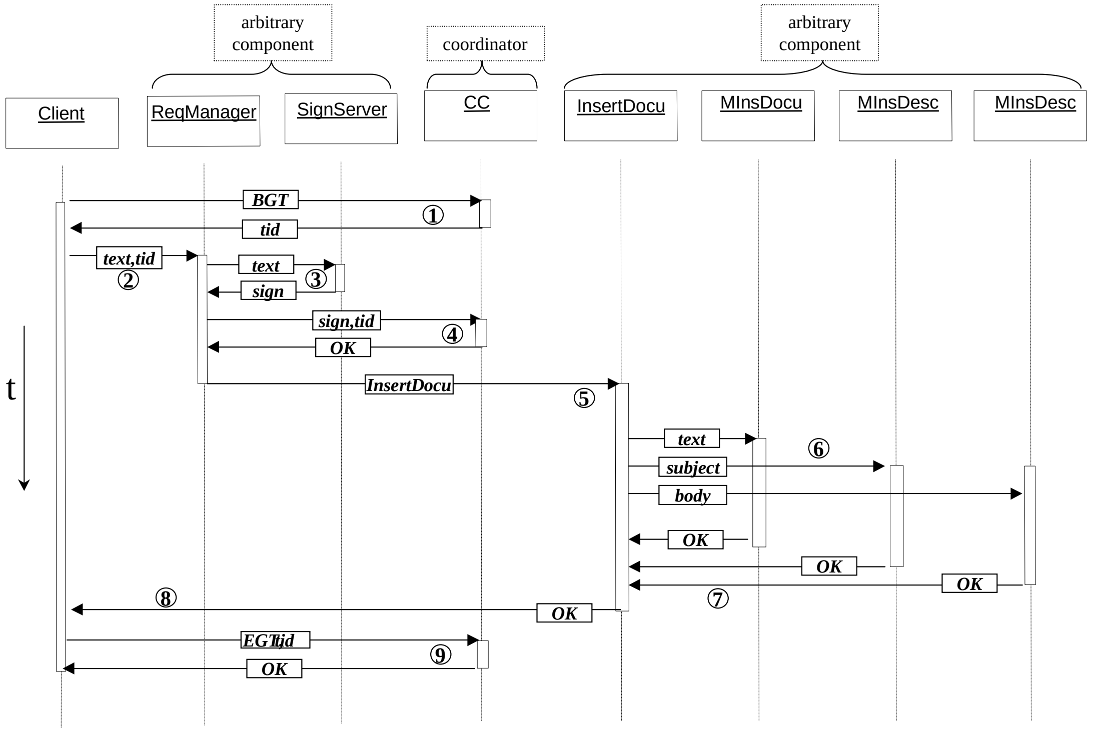

Figure 3: Interaction diagram for an insertion service transaction

Subsection 3.2. Before submitting the subtransactions to the corresponding services, `InsertDocu` writes a BOT log marker. Then it starts the subtransactions.

7. After completing all subtransactions, `InsertDocu` logs the EOT.

8. Then, the service returns its result to the client.

9. The client may now submit further requests within the same transaction. In our example, it simply states the end of its transaction and sends this message to the concurrency control. The concurrency control in turn releases all locks for the transaction and writes the EGT marker to the log.

4 Experimental Setup

This section describes our experiments to assess the scalability of the system and to evaluate the different placement alternatives. In all experiments, we have generated streams of insertions and queries. These client streams were run at a workstation that was not part of the database cluster. In most of our experiments, every `InsertDocu` or `RetrieveDocu` invocation of such a stream is a transaction, i.e., signature locks are required and logging is performed. These artificially short transactions minimize the time that a service request must wait for a signature lock grant because transactions are short and locks released with the end of transaction. This simplification is sufficient, when interacting with search engines in the common way. We have conducted further experiments, where we investi-

table name	*description*	*size in tuples*
TP_LIB	document text table	139,064
IL_TP_LIB_COMBINED	index on the combined field	13,714,066
IL_TP_LIB_BODY	index on the body field	12,745,188
IL_TP_LIB_SUBJECT	index on the subject field	483,207

Table 3: Number of tuples for the complete collection

gate global transactions that may consist of more than one request. We present some of our findings in Section 5.

Documents. The document collections inserted into the system consist of news messages from various newsgroups. For our experiments, we took a complete snapshot of the usenet newsgroups server at the Computer Science Department of ETH Zurich as of end of October 1999. After eliminating messages that had only binary content, the collection consisted of 139,000 textual news documents. The complete amount of this textual data was 800MB of ASCII text. Inititally, we have loaded our document engine with these documents. Each of the following experiments started from this database state. The database size without indexes was 1GB and 1.7GB with indexes. This more or less corresponds to a scaling factor of 1 for the text retrieval benchmark discussed in [9]. Table 3 shows the sizes of the biggest tables when the complete collection is loaded into a single component database system.

Another 250MB of textual data with news messages has been used for the insertion streams. In our experiments, the insertion streams selected new documents at random from this data.

Queries. Queries for the query streams have been synthetically generated from the terms of the initial document collection, in a similar manner as described in [9]. Queries comprise one to five different query terms. The average length of a query is 2.1 terms which corresponds to the average query length at InfoSeek [18]. The size of the result set of our queries was restricted to $1/1000$ of the database size. Hence, queries do not yield more than 140 hits in the collection. Moreover, each query at least yields one hit. The distribution of queries to different indexes represents the sizes of the indexes in terms of postings stored. Hence, the biggest index gets most of the queries. We use different numbers of terms per query for our measurements:

- 50% of the queries to the subject index have one terms, 40% have two terms and 10% three terms. For more than 3 terms, there is nearly no posting. Hence, queries to the subject index with more than 3 terms do not occur.

- For both the body and the combined index, 40% of the queries comprise one term, 25% two terms, another 25% have three terms, 5% four terms and again 5% have five terms. This yields an average query length of 2.1 terms.

Queries are randomly taken from a set of 155,187 queries. Figure 4 gives a more detailed view of the query set. For the different index types, it shows the number of queries with a given number of terms.

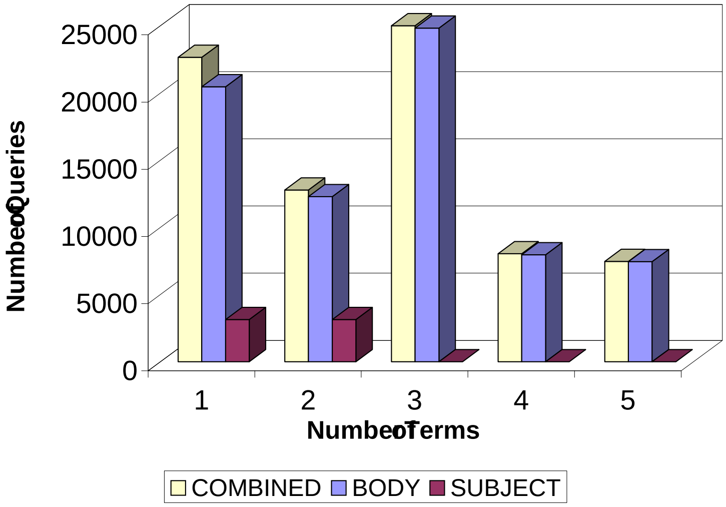

Figure 4: Distribution of queries

As performance metrics, we use response time, throughput and CPU consumption. In our measurements, global response time is the time interval from the request submission to the presentation of the results at the client.

Hardware and Software. We currently use PCs with one 400 MHz Pentium Processor and an interface to a network with a data transmission rate of 100 Mbit/sec (switched duplex Ethernet). All PCs run the Microsoft Windows NT Server 4.0 operating system software. We use the transaction monitor Bea Systems TUXEDO V6.5 [4] as the standard middleware product. It is well suited to transmit high volumes of data and it is available for nearly all operating systems. Note that we actually use only a very small subset of its features. We use FML buffers (fielded buffers for data transmission) and asynchronous invocation methods. We do not use TUXEDO's commit coordination functionality for XA-compliant databases, because this protocol is redundant with multi-level transactions; it incurs additional overhead, as shown in [17]. Our system uses ORACLE 8.0.4 to store the application data at the components. We configured the database systems with tablespaces on an IDE and an SCSI disk drive. The database buffer holds a maximum of 20,000 blocks of 2K size.

4.1 Performance of Retrieval and Insertion Services

One of the main issues of this current work is to provide interactive response times for concurrent document service executions. We have measured response times for insertion and retrieval requests.

The space of experiments now has three orthogonal dimensions:

- Placement Alternatives: We have considered the placement alternatives HASHCONS and HASHLOC.
- Cluster Size: In order to explore the speedup with the different placement alternatives, we have tested the system with 1, 2, 4 and 8 workstations. One of these components was the

coordinator at the same time. The monolithic configuration with only one workstation serves as a reference point.

- `Workload Patterns:` The performance of the placement alternatives depends on the workload pattern. A vector (a, b) denotes the system workload, where a and b represent the number of concurrent insertion and retrieval requests, respectively. That is, there are a clients having submitted an insertion and b clients with a retrieval request. Each client immediately submits a new request when it has received the response from the previous one. Subsequently, we refer to these clients as streams. We call an experiment that we conducted for a particular vector, such as $(5, 5)$, a *run*. The workload vectors of the experiments are $(0, 1)$, $(1, 0)$, $(1, 1)$, $(5, 5)$, $(10, 10)$, $(20, 20)$, $(30, 30)$, $(40, 40)$, $(50, 50)$, $(60, 60)$ and $(70, 70)$. Each insertion request inserts around 400 tuples to the index tables and inserts the document text. Note, that 70 insertion requests concurrently with 70 retrievals is a heavy workload, especially, because both request types basically operate on the same data. On another level, we conducted additional runs for the read-intensive workloads $(1, 100)$ and $(1, 150)$.

4.2 Coordination Overhead

Our experiments so far do not state how many resources each coordination specific task consumes. In particular, they do not reveal whether the coordinator is a bottleneck.

A second series of experiments quantifies this overhead. We conducted these experiments with the HASHCONS placement alternative and considered the full range of workloads as discussed previously. The metrics that we use in these experiments are as follows: we determine the CPU consumption of the coordinator process and compare it to other processes. Additionally, we determine the fraction of the average response time of a service invocation that the coordinator process spends for conflict test and logging. We compare this to the coordination overhead for long transactions that we introduce in the following Subsection.

4.3 Evaluation of Global Transactions

Our system has been developped to support transactions of document service requests. With our previous setups, we have not investigated this aspect in quantitative terms. With the experiments described in this subsection, we investigate how complex transactions with multiple service invocatoins perform compared to short transactions with only one service invocation. To this end, we have changed the client streams. For the experiments with the short transactions, clients declare a transaction, submit a request, and end the transaction. In the setting for long transactions, insertion clients now declare transactions that comprise two insertion requests. Retrieval transactions – as with short transactions – only comprise a single retrieval request. Additionally, we have applied a wound-and-wait protocol [5] in the long transaction setting. That is, insertion requests wait for a lock grant. Retrieval instead does not wait, the coordinator refuses request and the client has to repeat the request. This avoids deadlocks. In the experimental section, we will report on response times for long and short transactions.

5 Outcome and Discussion of Experiments

5.1 Experiments for Response Times and Throughput

Insertion Response Times. Figures 5 and 6 show the graphs for the different cluster sizes and the HASHCONS placement alternative. The response times in 5 increase nearly linearly from low to high workloads for all cluster sizes. This is what we would expect: with increasing workloads, more concurrent transactions compete for the restricted system resources, mainly disk I/O. This leads to poor response times: insertion response times reach an unbearable average of 93 seconds per service invocation for high workloads with 1 component. As a rule of thumb, doubling the workload doubles the response time for an insertion. Increasing the number of components leads to significantly better response times.

> With 8 components, workloads such as (20,20) do not exceed 5 seconds in average response time.

A similar observation holds for throughput. All curves in Figure 6 are nearly constant with workloads of (5,5) and higher. That is, already with (5,5) the system comes very close to its point of saturation. At that point a further increase of the workload does not lead to an increase in throughput. Another observation is that admitting a single reader concurrently to an insertion reduces throughput only for cluster sizes up to 4 nodes. Figure 6 nicely shows that this is no longer the case for big cluster sizes: with 8 components, throughput does not decrease when admitting retrieval – that is moving from the workload (1,0) to (1,1). The other placement alternative HASHLOC leads to similar response times as Figure 9 and 10 shows.

Retrieval Response Times. The general behaviour of retrieval response times for increasing workloads is similar to that of insertion requests: with increasing workloads, response times increases as well – for the same reasons as with insertions. Figures 7 and 8 shows the respective curves. The other observation is that retrieval response times are generally about half of the response times for insertions with high workloads.

> A cluster of 8 nodes yields average retrieval response times of less than 5 seconds for relatively high workloads such as (30,30) or (40,40).

This facilitates working with the system interactively.

5.2 Experiments for Speedup

Table 4 and Table 5 show the speedup of response times in relation to a setup with only one component.

The speedup for low workloads is less than linear for both placement alternatives. Ideally, one would expect a speedup of 4 when going from 1 to 4 components. In the case of the low workload (1,1) the speedup is below 3 for 4 components and below 5 for 8 components. For the workload (20,20),

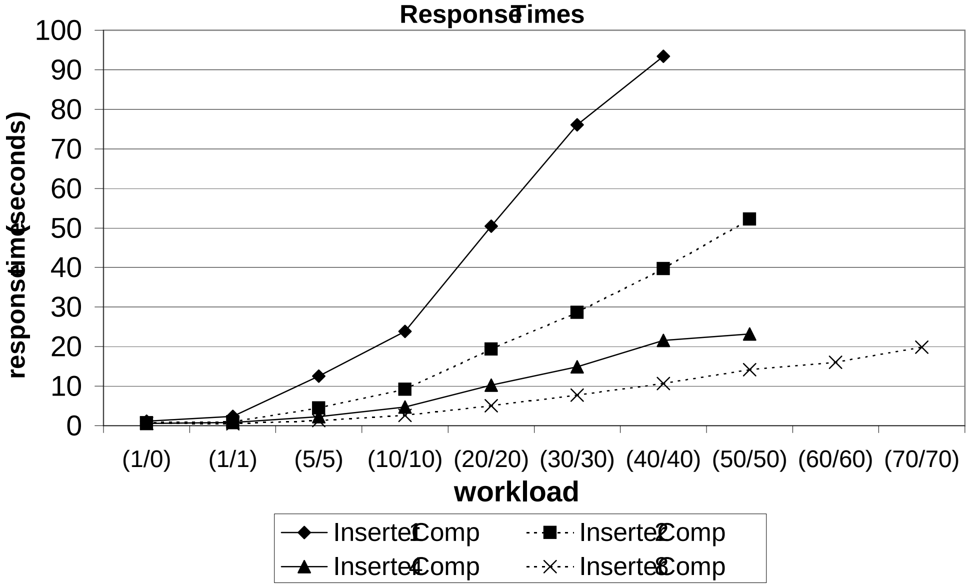

Figure 5: Placement HASHCONS response times insertion

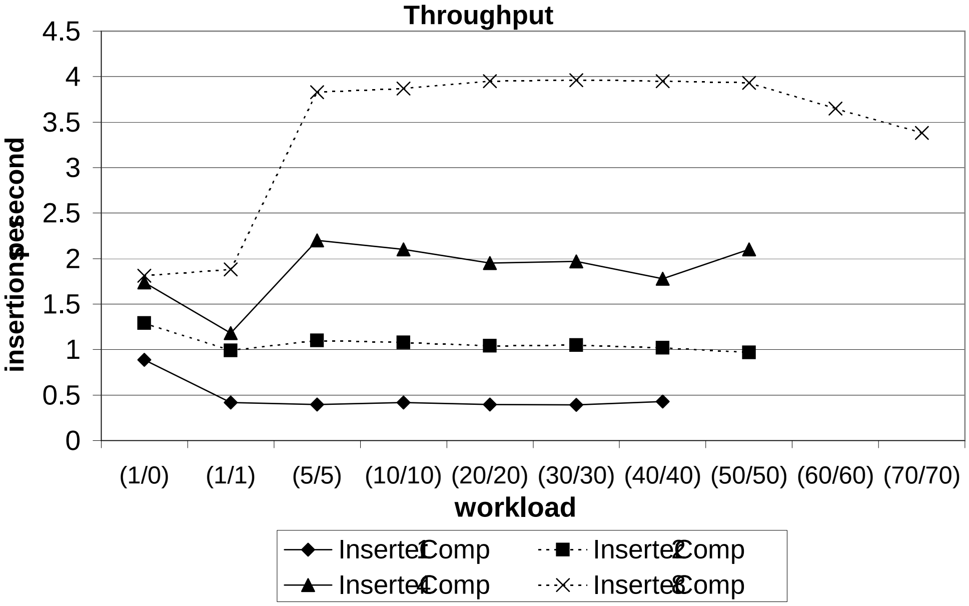

Figure 6: Placement HASHCONS throughput insertion

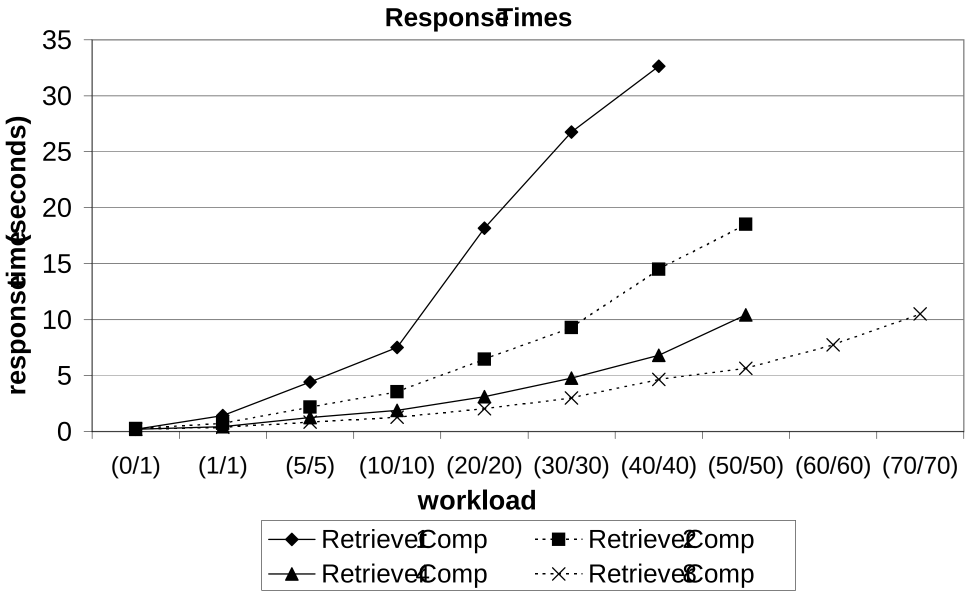

Figure 7: Placement HASHCONS response times retrieval

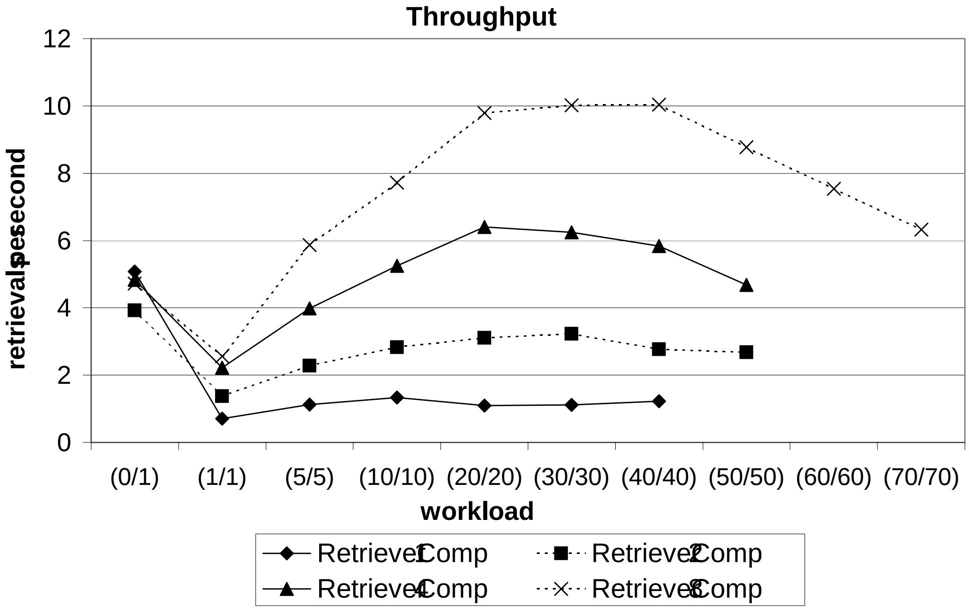

Figure 8: Placement HASHCONS throughput retrieval

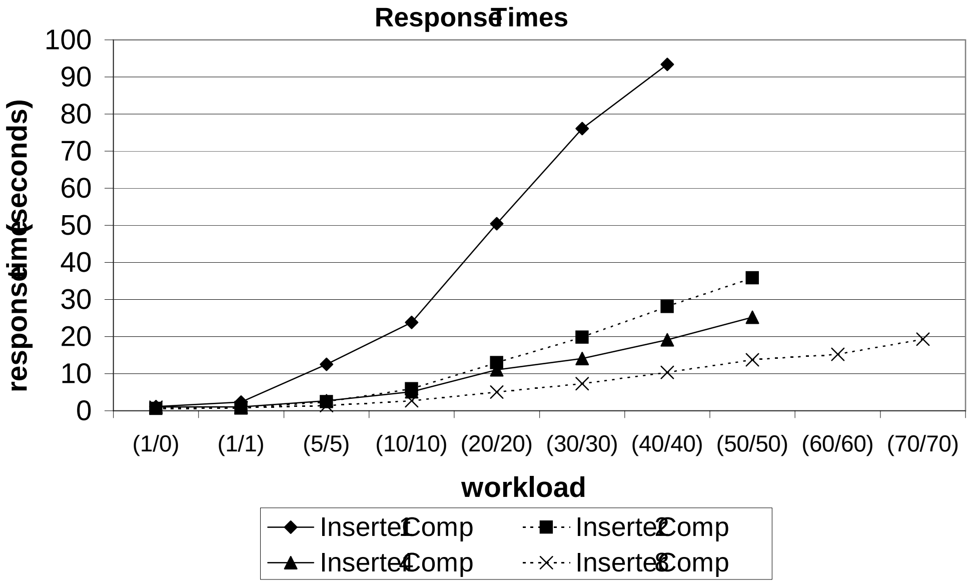

Figure 9: Placement HASHLOC response times insertion

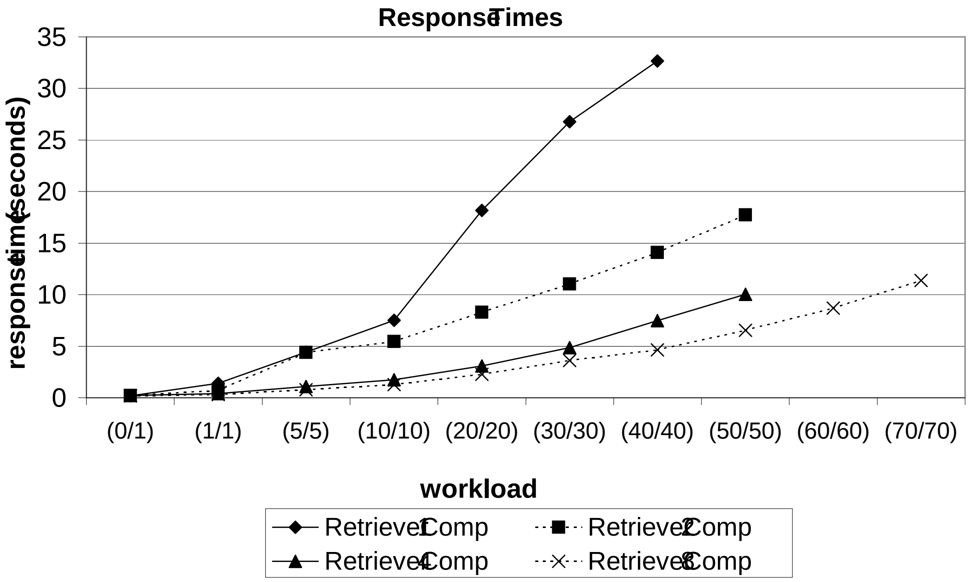

Figure 10: Placement HASHLOC response times retrieval

	4 Components		8 Components	
workload	HASHCONS	HASHLOC	HASHCONS	HASHLOC
$(1, 1)$	2.86	2.21	4.63	2.79
$(20, 20)$	4.93	4.57	10.01	9.95
$(40, 40)$	4.34	4.89	8.77	9.02

Table 4: Insertion speedup

	4 Components		8 Components	
workload	HASHCONS	HASHLOC	HASHCONS	HASHLOC
$(1, 1)$	3.15	3.41	3.63	4.40
$(20, 20)$	5.81	5.88	8.90	7.98
$(40, 40)$	4.79	4.37	7.02	7.01

Table 5: Retrieval speedup

the speedup both for insertion and retrieval is more than linear. E.g., for 8 components, the insertion speedup is 10 and for retrieval it is close to 9 for the HASHCONS placement alternative.

Figure 11 represents the increase in maximum throughput when increasing the number of components. Figure 11 shows that insertion throughput increases nearly linearly: doubling the number of components doubles the throughput. For retrieval, the situation is a bit more differentiated: with 1, 2 and 4 components there is a linear increase. But from 4 to 8 components the gain in throughput is less than linear. This is not surprising because with a retrieval request each component has to process the query.

> With insertions, doubling the number of components yields a speedup of 2 or more for medium to high workloads. At least, it also doubles the throughput. With retrieval, speedup and increase of throughput are less than linear.

5.3 Coordinator Overhead

Figure 12 and Figure 13 depict the CPU consumption of the most CPU-intensive processes for different workloads with 8 components. The curve for ”GTXMgmt” represents the CPU consumption of the centralized coordinator process that does the conflict test and performs logging for global transactions. Figure 12 also shows the CPU requirements of the ORACLE database system. Clearly, ORACLE consumes some orders of magnitude more processing power than any of the other processes. Figure 13 only contains processes from our middleware extensions. Hence, the CPU consumption of the ORACLE processes has been omitted. The graph shows that the coordinator process (GTXMgmt) has little CPU requirements. That is, within the 900 seconds of a (70,70) measurement run for a cluster with 8 components, GTXMgmt consumes only 27 CPU seconds. ORACLE instead needs about 148 CPU seconds at one component.

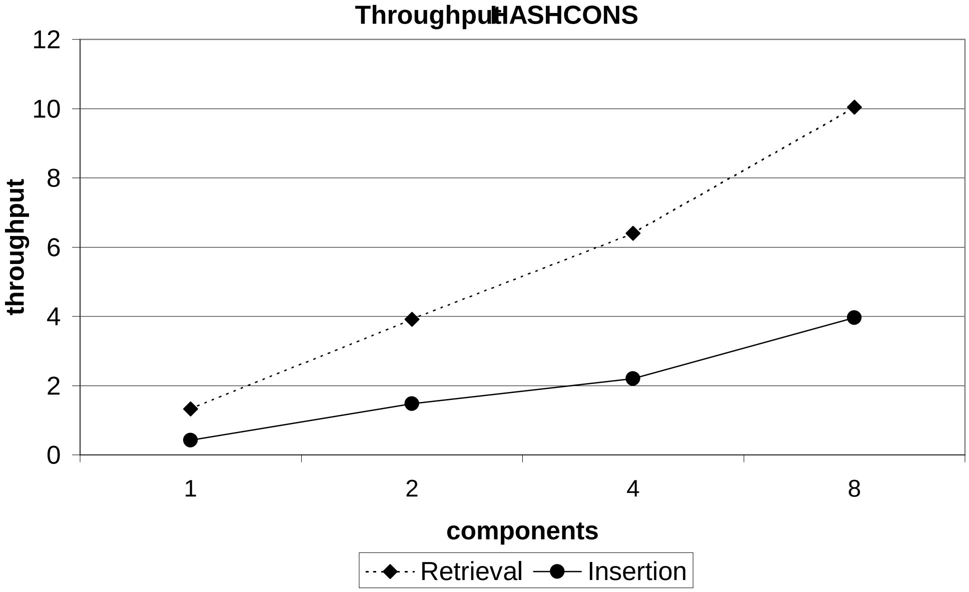

Figure 11: Increase in throughput – HASHCONS

workload	*retrieval time*	*retrieval cc fraction*	*insertion time*	*insertion cc fraction*
	seconds	seconds $\cdot 10^{-3}$	seconds	seconds $\cdot 10^{-3}$
(1,1)	0.42	0.0	0.54	0.0
(20,20)	3.00	0.183	4.75	0.105
(40,40)	6.15	0.342	9.13	0.316
(60,60)	10.1	0.427	13.56	0.475
(1,100)	4.94	0.0061	4.79	0.372
(1,150)	7.38	0.009	6.46	1.223

Table 6: Single request transactions: response time fractions spent in the concurrency control

We now discuss the impact of second layer concurrency control onto request response times. Table 6 shows the average response times in seconds for retrieval and insertion requests for the different workloads of the 8 component setup. The table also shows the fractions of the response times for second layer transaction management in the coordinator. In contrast, Table 7 gives the values for transactions with either a single retrieval request or two insertion requests.

> The response times for processing the document requests are in the order of seconds or more. The fraction spent for concurrency control and global logging is below one millisecond in most cases.

That means that the performance impact of centralized computations is not significant in our particular setting.

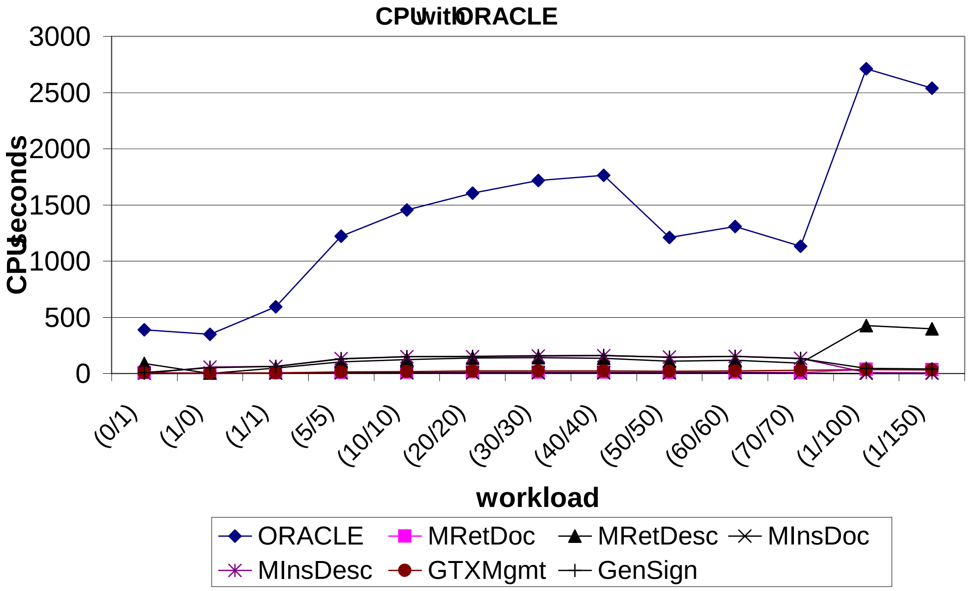

Figure 12: CPU consumption of the different processes of our architecture with 8 components with ORACLE

workload	*retrieval time*	*retrieval cc fraction*	*insertion time*	*insertion cc fraction*
	seconds	seconds $\cdot 10^{-3}$	seconds	seconds $\cdot 10^{-3}$
(1,1)	0.42	0.004725	0.52	0.0062
(20,20)	2.95	0.25	4.51	0.1
(40,40)	6.41	0.37	8.93	0.15
(60,60)	9.88	1.057	13.3	0.77
(1,100)	5.04	0.225	4.59	0.337
(1,150)	7.97	0.14	8.02	0.29

Table 7: Multi-request transactions: response time fractions spent in concurrency control

5.4 Evaluation of Global Transactions

Figure 14 shows the outcome of our experiments for global transactions on a cluster with 8 nodes. Figure 14 shows response times for different workloads.

> Response times are only minimally affected by introducing long global transactions in our setting.

This is a nice finding, as it allows the user to decide freely upon having transactions or not. Defining long global transactions does not degrade the performance of his application. Hence, any decision about transaction properties only has to consider the needs of the application, and not the impact on performance.

Summary of our Findings. For any HASH alternative, increasing the number of components leads to a significant speedup of both insertion and retrieval, as Subsection 5.2 shows. It is particularly high in

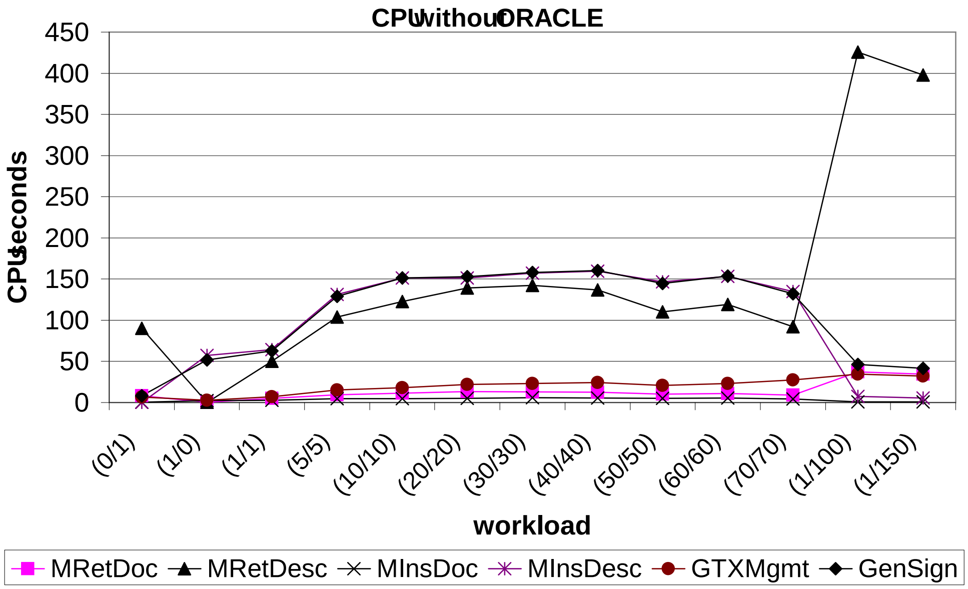

Figure 13: CPU consumption of the different processes of our architecture with 8 components without ORACLE

medium workload situations. The system has interactive response times with an appropriate number of components for medium workloads (Subsection 5.1).

Delegating the application-specific operations to other nodes gives rise to good scalability as the experiments from Subsection 5.3 show. The overhead introduced by semantic concurrency control and logging in the coordinator is minimal even for hundred concurrent retrieval requests or for mixed workloads.

Another finding from our experiments that we do not describe here for lack of space is that data and workload distributes evenly in the cluster. This yields a good load balancing for our system.

6 Conclusions

Performance of document search and indexing engines remains an important research issue. Our contribution is the design and implementation of an engine that supports concurrent access of retrieval and insertion. Insertion also comprises the update of the index data. Hence, index data is always up-to-date in our setting. The engine is implemented on top of a database cluster on PCs. Mapping information retrieval functionality to relational database systems supports both SQL queries on the relational attributes as well as retrieval on the textual attributes of the data.

A cluster of databases is promising with respect to scale-out. With a coordination-based architecture, there is a coordinator cluster node that manages the distributed processing at the other cluster nodes – the components.

With our design, the implementation of the coordinator is as slim as possible. Only necessarily centralized processing is actually done at the coordinator. We split the document service processing such as

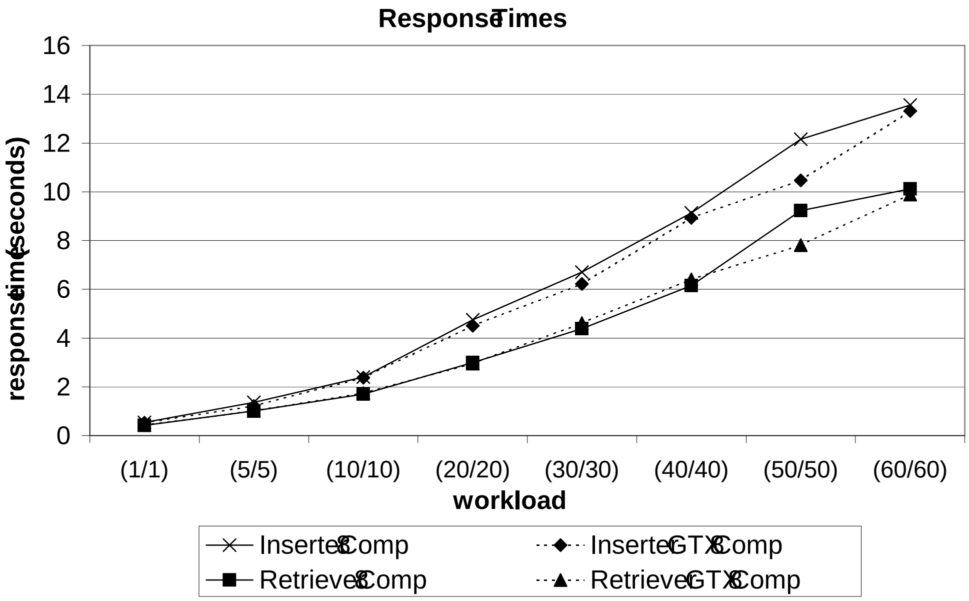

Figure 14: Long and short transactions with placement HASHCONS response times

term extraction and stemming into independent functionality. We have implemented that functionality with independent processes, which we replicate arbitrarily among cluster nodes. This distributes the workload and allows to process a request in parallel on several nodes. Furthermore, our coordinator applies multi-level-transactions. Hence, it increases parallelism of the request processing without sacrificing correctness. We avoid a two-phase commit protocol over the component databases and provide an efficient proprietary implementation of logging for document services.

In our experiments, we have shown that already with 8 components our system yields response times of less than 5 seconds for workloads such as 30 insertion client streams running concurrently with 30 retrieval client streams. The coordination overhead for document service transactions is in the order of milliseconds, whereas response times for document services are in the order of seconds. Furthermore, speedup of insertion service processing is linear when we increase the number of components. The speedup of retrieval services is less than linear.

Generally, we conclude that a cluster of databases is well suited as an infrastructure for managing large amounts of derived, redundant or replicated data. This is an issue for management of materialized views such as data cubes, any index data, and inverted lists for document retrieval. From the findings in our setting, we conclude that our approach is analogously applicable to the other cases with similar good results.

Acknowledgement We want to thank Jim Gray for many discussions about clusters of databases.

References

[1] B. Badrinath and K. Ramamritham. Performance evaluation of semantics-based multilevel concurrency control protocols. In *Proc. of the SIGMOD Conf. 1990*, pages 163–172, 1990.

[2] D. Barbará, S. Mehrotra, and P. Vallabhaneni. The gold text indexing engine. In S. Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 172–179. IEEE Computer Society, 1996.

[3] C. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. Copeland, and W. Wilson. Db2 parallel edition. *IBM Systems Journal*, 34(2):292–321, 1995.

[4] BEA Systems. *TUXEDO Guides and References (V 6.5)*.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In *Proc. of the 20th Conf. on Very Large Databases*, pages 192–202, 1994.

[7] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in bubba. In *Proc. of the ACM SIGMOD Conference*, pages 99–108, 1988.

[8] P. Dadam, P. Pistor, and H. Schek. A predicate oriented locking approach for integrated information systems. In *IFIP Congress 1983*, pages 763–768, 1983.

[9] S. DeFazio. *Overview of the Full-Text Document Retrieval Benchmark (In: The Benchmark Handbook – Jim Gray (ed.))*, pages 435–487. Morgan Kaufmann, 1991.

[10] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–61, Mar. 1990.

[11] 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.

[12] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[13] T. Grabs, K. Böhm, and H.-J. Schek. A document engine on a db cluster. In *High Performance Transaction Systems Workshop, Asilomar, California*, 1999.

[14] D. A. Grossman, O. Frieder, D. O. Holmes, and D. C. Roberts. Integrating structured data and text: A relational approach. *Journal of the American Society for Information Science (JASIS)*, 48(2):122–132, Feb. 1997.

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

[16] M. Kamath and K. Ramamritham. Efficient transaction support for dynamic information retrieval systems. In *Proc. of ACM SIGIR*, pages 147–155, 1996.

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

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

[19] D. Knaus and P. Schäuble. The system architecture and the transaction concept of the SPIDER information retrieval system. *IEEE Bulletin of the Technical Committe on Data Engineering*, 19(1):43–52, 1996.

[20] Microsoft Corp. Building high-performance databases using microsoft sql server 2000 federated database servers. Technical report, Microsoft Corp., http://www.microsoft.com/, 2000.

[21] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[22] W. Schaad, H.-J. Schek, and G. Weikum. Implementation and performance of multi-level transaction management in multidatabase environment. In *Proc. of RIDE-DOM'95 Taipei, Taiwan*, pages 108–115, 1995.

[23] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 289–300, 1994.

[24] R. Vingralek, H. Hasse-Ye, Y. Breitbart, and H.-J. Schek. Unifying concurrency control and recovery of transactions with semantically rich operations. *Theoretical Computer Science*, pages 363–396, 1998.

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

[26] D. J. D. Witt and J. Gray. Parallel database systems: The future of high performance database processing. *Comm. of the ACM*, 35(6):85–98, 1992.