PAPER

Accelerating Database Processing at Database-driven Web Sites

Seunglak CHOI†, Jinwon LEE†, Nonmembers, Su MYEON KIM†∗, Student Member, Junhwa SONG†, and Yoon-Joon LEE†, Nonmembers

SUMMARY Most commercial Web sites dynamically generate their contents through a three-tier server architecture composed of a Web server, an application server, and a database server. In such an architecture, the database server easily becomes a bottleneck to the overall performance. In this paper, we propose WDBAccel, a high-performance database server accelerator that significantly improves the throughput of database processing. WDBAccel eliminates costly, complex query processing needed to obtain query results by reusing the results from previous queries for subsequent queries. This differentiates WDBAccel from other database cache systems, which employ traditional query processing. WDBAccel further improves its performance by fully utilizing main memory as the primary storage. This paper presents the design and implementation of the WDBAccel as well as the results of performance evaluation with a prototype.

key words: Database server acceleration, Web site performance, Dynamic content caching, Query result caching

1. Introduction

With the rapid expansion of the Internet, lots of interesting and value-generating services are now provided through WWW. For effective services, it is important for Web sites to sustain a high level of throughput. In many cases, those services are provided through many system components, e.g., Web servers, application servers, and database servers. Thus, the generation of a Web content involves a long sequence of operations through those system components. In such a system, a database server can be a bottleneck to overall Web sites’ performance. A performance problem for Web servers and application servers is now solved by using proven technologies such as the Web server accelerating [23] and the server clustering [7].

We propose a high-performance database server accelerator, called WDBAccel, that significantly improves the performance of database processing at database-driven Web sites. The main approach of WDBAccel is to cache results of frequently-issued queries and reuse these results for incoming queries. The cache can serve the query with significantly less overhead than a database server. In addition, WDBAccel is designed to use main memory as the primary storage, minimizing disk operations. Thus, WDBAccel can significantly improve the performance of database processing.

WDBAccel differs from other existing DB cache systems [1], [17], [20], [25] in that the primary purpose of WDBAccel is to accelerate database processing. In contrast, the primary purpose of other cache systems is to distribute the load of database processing into multiple cache nodes. Those systems replicate a database into multiple nodes and distribute queries among them. Each cache node relies on the underlying DBMS to store replicas and to generate a query result from the replicas. The underlying DBMS stores the replicas as ordinary database tables. Thus, such DBMS-based cache systems require costly query processing to obtain a query result from a lot of tuples in a table.

WDBAccel provides database-driven Web sites with several advantages. First, WDBAccel reduces the cost of ownership. WDBAccel is a specialized lightweight system focusing on tasks related to caching and serving query results, and operates in collaboration with an origin DBMS. In this way, WDBAccel can achieve a high level of performance even on a lower-end H/W system. Second, the high-performance nature of WDBAccel reduces the number of cache nodes which should be managed by an administrator, reducing administration cost. Third, WDBAccel can be easily deployed as a middle-tier solution between Web application servers and database servers (see Figure 1). By supporting the same database connectivity interface for Web applications as existing interfaces, e.g., JDBC and ODBC, WDBAccel does not require any modifications to Web applications.

WDBAccel is a new caching system specialized to accelerate database systems in Web sites. Thus, it is different from traditional Web caches as well as
database systems. In this paper, we explain the design and implementation of WDBAccel. Main characteristics of the system can be summarized as follows.

- WDBAccel employs its own processing scheme called fragment processing to generate the result of a given query from fragments. A fragment is a query result stored in a cache. The fragment processing first searches for previous query results which can be used to generate the result of an incoming query. It then generates the query result from those matching fragments using union and trim filters, which cut off duplicated and unwanted tuples.
- To reduce the cost of generating query results, WDBAccel executes fragment selection algorithms, which choose an appropriate set of fragments that minimizes the generation cost. We propose two algorithms: one aims to minimize the generation cost and the other aims to minimize the cost of the selection algorithm itself.
- To speed up finding fragments matching an incoming query, WDBAccel provides an index for query results. The cache should be able to store an enormous number of query results. Using the index, WDBAccel efficiently finds the matching query results from many stored query results.
- To efficiently utilize limited storage space, WDBAccel employs its own storage system. The main characteristic of the system is that it composes its cache pool in the unit of tuples. This approach removes the storage redundancy of the tuples belonging to two or more query results.
- WDBAccel employs a refined cache replacement policy which differently evaluates storage costs from other Web caching systems. In many systems, the replacement policy considers the size of cached data items. However, it is not appropriate in the tuple-based storage policy of WDBAccel. Thus, WDBAccel collectively considers the size of query results and tuples shared among them.

This paper is organized as follows. Section 2 gives the architecture of WDBAccel and section 3 describes fragment processing. In section 4, we explain technical issues faced in designing WDBAccel, including fragment indexing, cache storage, cache replacement and cache consistency, and solution approaches to these problems. Section 5 describes implementation details of WDBAccel. We evaluate and analyze the performance of WDBAccel from various points of view in section 6. In section 7, we describe related works and compare them to our system. Finally in section 8, we present conclusions.

2. System Architecture

2.1 System Overview

WDBAccel is a Web database server accelerator. Being located between Web application servers (WAS’s) and database servers, it processes queries delivered from the WAS’s on behalf of database servers. The followings are the characteristics of the WDBAccel system.

WDBAccel is a query result caching system. As explained in Introduction, it stores the results of previous queries and then serves incoming queries delivered from WAS. It constructs responses for queries by using previously requested results. Thus, it can serve queries without complex query processing. For instance, if it receives the same query as before, the previous query result is selected and sent without any other query processing. Even in the case that an incoming query does not exactly match any previously stored queries, the system tries to find stored queries from which the result of an incoming query can be derived. This derivation is much simpler than the query processing occurring in the origin database server.

The query result caching is extremely useful when a workload is read-oriented. The workload of many Web applications is read-oriented. In many Web sites, visitors spend the most time finding and reading some information, e.g., product catalogs, news, articles, etc. Update interactions such as ordering products and posting articles are relatively very infrequent. For example, the TPC-W benchmark [26] specifies that the portion of read queries, in an average case, is 80% of the entire workload. Thus, we can expect that the query result caching will show a high level of performance in many Web applications.

To further improve WDBAccel’s performance, we use main memory as the primary storage for storing query results. WDBAccel does not maintain the persistency of query results stored in main memory. However, it is not a serious drawback because WDBAccel stores replicas; all data are stored safely at the origin database server, so sudden loss of cached data will affect only the cache performance. To effectively utilize the limited space of main memory, it is vital to maximize the hit ratio, i.e. the rate of reusing query results cached in main memory. The hit ratio is determined by query matching, cache storage, and cache replacement policies which will be discussed in section 4.

1The TPC-W benchmark is an industrial standard benchmark to evaluate the performance of database-driven Web sites. It models an e-commerce site (specifically, an online bookstore) that is a representative, common Web application.
2.2 System Components

As shown in Figure 2, WDBAccel consists of the following components:

- **Fragment Processor (FP)** is the core component, which generates the result for a given query from stored fragments through two steps: fragment searching and query result deriving. In the step of fragment searching, FP parses a newly incoming query and tries to match it to cached fragments. If a fragment in the cache storage exactly matches the query, the fragment is directly used for a response. On the other hand, if the result of the query can be derived from one or more fragments, the step of query result deriving is required to obtain the query result from the matching fragments. If the cache does not have sufficient fragments to construct the query result, FP forwards the query to the origin database server.

- **Cache Dictionary (CD)** is a collection of meta information about fragments stored in the Cache Pool (see Figure 3). CD is referenced by FP for finding matching queries. Each entry of CD stores meta information on a fragment as shown in Figure 1. Entries are classified into different query groups according to the structure of queries. Then, each group is indexed by selection regions of the queries. This index will be discussed in detail in section 4.1.

- **Cache Pool (CP)** is a storage space and management module for fragments. It stores and retrieves fragments efficiently while minimizing the redundant storage by storing one copy of each duplicate tuple. CP will be discussed in detail in section 4.2.

- **Cache Controller (CC)** updates the Cache Dictionary and the Cache Pool when receiving a new query result. It also executes a cache replacement algorithm when the cache does not have enough space to store new query results.

- **Consistency Maintainer (CM)**. The caching inherently incurs the inconsistency between cached results and an origin database. If a data element in an origin database is updated, the fragments derived from the updated data will be stale. WDBAccel includes CM which ensures the consistency. A consistency mechanism used in WDBAccel will be discussed in detail in section 4.4.

- **Query Redirector (QR)** is in charge of communication with the Application Interface modules on WAS nodes. It routes a query according to the query type. If the query is of a read type, it will be sent to FP. Otherwise, QR forwards queries to the origin database server. So, queries such as insert, delete and update will be processed at the origin database server.

- **Application Interface** is installed on each WAS node and replaces an existing database driver which was used to communicate with an origin database server. This module provides applications with standard database connectivity interfaces, such as JDBC and ODBC, to WDBAccel.

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection region</td>
<td>selection predicates of a fragment</td>
</tr>
<tr>
<td>last access time</td>
<td>time that a fragment was last accessed</td>
</tr>
<tr>
<td>number of access</td>
<td>number of accesses to a fragment</td>
</tr>
<tr>
<td>fragment size</td>
<td>size of a fragment</td>
</tr>
<tr>
<td>data pointer</td>
<td>pointer to data stored in the Cache Pool</td>
</tr>
</tbody>
</table>
2.3 Internal Processing Flow

Figure 2 also depicts the processing sequence of operations for serving each query. The WAS sends a query to QR (1), then QR checks whether the given query is read or write. If the query is a write, it sends the query to CM (A). CM forwards the query to the origin database server (B) and performs the process to maintain cache consistency. If the query is a read, then it forwards the query to FP (2). FP references CD to decide whether the result for the incoming query can be constructed based on cached fragments (3). If fragments for the incoming query are found, FP retrieves the fragments (4). Otherwise, FP sends the query to the database server (a) and receives the result for the query (b). Then, it forwards both the query and the query result to CC (c). CC inserts the query into CD and the query result into CP (d). (When the cache does not have enough space to store new query results, CC executes a cache replacement algorithm.) FP constructs and sends the query result to QR (5). Finally, QR sends the query result to the WAS (6).

3. Fragment Processing

WDBAccel generates query results in a completely different way from DBMS-based cache systems. WDBAccel generates the results of incoming queries by reusing previously cached query results. It does not process high-cost query operators such as top-n, join, and GROUP BY operators, rather it reuses their results received from an origin database server. In contrast, DBMS-based systems utilize the traditional query processing methods of DBMSs. They use conventional DBMSs to copy tables from origin databases or to materialize views on those tables. Data are stored in the form of ordinary database tables, even when data are delivered to a form in the unit of query results [1].

The reusing approach is very effective because the forms of those high-cost query operators are usually fixed and thereby the operators can be easily matched with previous operators. In many Web-based applications, a query template (i.e., the form of queries) is predetermined by an HTML form. When a user clicks the submit button in the form, a WAS generates a query instance from a predetermined query template. Thus, query instances generated from the same template share the same forms of the query operators. The only difference among the query instances lies in their selection predicates, which are determined from search keywords or ranges specified by users.

Even in the case of selection predicates, WDBAccel can show better performance than DBMSs. In WDBAccel, the search space for selection predicates is narrowed to a set of frequently accessed query results. In contrast, in DBMSs, the search space is always an entire table. A query with a LIKE predicate is a good example to show the performance improvement due to the different search space. For a given LIKE query, WDBAccel finds its result in the search space of cached query results. The cost can be O(1) if a hash structure is used. However, DBMSs do not generally provide indices to fully cover all types of LIKE queries. We can think of three types of LIKE queries, i.e., ‘%keyword%’, ‘%keyword’, and ‘keyword%’. DBMSs usually provide indices based on keyword prefixes. Therefore, the ‘keyword%’ type queries can be handled utilizing the index. However, as for the other types, the DBMS-based cache systems scan all tuples in a table for pattern matching, which costs O(N).

In this section, we describe fragment processing, which generates the result for a given query from previously cached query results. As mentioned earlier, fragment processing consists of fragment searching and query result deriving (see Figure 4), which are described in section 3.1 and 3.2 respectively. Fragment searching is further divided into two sub-steps, query matching and fragment selection. Query matching finds one or more fragments that can be used to generate the result of a given query. Fragment selection chooses an appropriate set of fragments from matching ones such that the processing cost of query result deriving is reduced. We describe fragment selection in section 3.3.

Fragment processing works with queries having conjunction predicates. To handle disjunctions, we reduce a boolean expression to its disjunctive normal form (disjunction of conjunctions), for example, \((a \land b) \lor (b \land c)\). Each conjunction is processed as a separate conjunctive query. The results of the conjunctions are cached separately. These results are merged to satisfy the origin query. Following this scheme, a selection region of both query and fragment corresponds to a conjunction of one or more selection predicates.
3.1 Fragment Searching

Query matching is the first step in fragment searching. During query matching, WDBAccel employs derived matching as well as exact matching to maximize the cache hit ratio. When WDBAccel fails to find a fragment that matches a given query exactly, it tries to find one or more fragments from which the query result can be derived. In Example 1, a query $Q$ can be derived from the union of fragments $F_1$ and $F_2$ although $Q$ does not exactly match either $F_1$ or $F_2$.

**Example 1** (An example of derived matching): $Q$ can be derived from the union of $F_1$ and $F_2$ since the selection region of the union contains the selection region of $Q$.

\[
F_1: \text{SELECT } * \text{ FROM ITEM WHERE I_PUB_DATE } \geq '01/01/2003' \text{ AND I_PUB_DATE } \leq '01/20/2003'
\]

\[
F_2: \text{SELECT } * \text{ FROM ITEM WHERE I_PUB_DATE } \geq '01/10/2003' \text{ AND I_PUB_DATE } \leq '01/30/2003'
\]

\[
Q: \text{SELECT } * \text{ FROM ITEM WHERE I_PUB_DATE } \geq '01/05/2003' \text{ AND I_PUB_DATE } \leq '01/25/2003'
\]

There are popular computational dependencies that can be used to find derived matches. A selection region dependency is a computational dependency on selection regions. In example 1, a query $Q$ has a selection region dependency on the union of a fragment $F_1$ and a fragment $F_2$ in that the selection region of the union contains the selection region of $Q$. Other examples of computational dependencies include the dependencies among aggregations and the dependencies among join predicates. For example, a query having ‘GROUP BY YEAR’ can be derived from a fragment having ‘GROUP BY YEAR, MONTH’. These dependencies are commonly utilized in Data Warehouses [11], [21], [24].

WDBAccel focuses on selection region dependencies which maximize the probability of finding the derived matches. By investigating these dependencies among selection regions, WDBAccel is highly likely to find derived matches. This is due to the characteristic of the queries used in many Web-based applications; selection regions from different query instances with the same query template tend to overlap each other and may form a hot range. For example, in the following query template, selection regions on I_PUB_DATE will frequently cluster around the present time. This is because customers in an online bookstore prefer to buy new books.

\[
\text{SELECT I_TITLE, I_ID, A_FNAME, A_LNAME FROM ITEM, AUTHOR WHERE I_A_ID = A_ID AND I_PUB_DATE } \geq @Lower_Date \text{ AND I_PUB_DATE } \leq @Upper_Date \text{ ORDER BY I_TITLE}
\]

A query can be derived from fragments belonging to different templates. For example, a query in a form having $I_{\text{PUB DATE}}$ and $I_{\text{COST}}$ as its selection attributes can be derived from a fragment of another form having $I_{\text{PUB DATE}}$ as its selection attribute. To support such derived matching, WDBAccel tries to find fragments in different templates when it fails to find sufficient fragments for deriving in the template of a given query.

3.2 Query Result Deriving

When one or more fragments are selected by derived matching, the process of query result deriving follows the fragment searching process. In order to derive a query result, two filters, union and trim, are applied to the tuples of the matching fragments. Only tuples that pass both filters constitute a query result. The union filter is used to merge the matching fragments and to eliminate tuple duplicates. In Example 1, the tuples of the matching fragments $F_1$ and $F_2$ are merged and the duplicates ($'01/10/2003' \leq I_{\text{PUB DATE}} \leq '01/20/2003'$) removed.

The trim filter is used to cut off the tuples that are not part of a query result. In Example 1, the tuples ($'01/01/2003' \leq I_{\text{PUB DATE}} < '01/05/2003'$) of the matching fragment $F_1$ and the tuples ($'01/25/2003' < I_{\text{PUB DATE}} \leq '01/30/2003'$) of the matching fragment $F_2$ are cut off by the trim filter.

Both the union and the trim filters use selection attribute (e.g., $I_{\text{PUB DATE}}$) values of tuples. The union filter uses such values of a tuple in order to determine whether the tuple has been already selected by previous fragments. The trim filter also compares selection attribute values of a tuple to the selection predicates of a query in order to determine whether the tuple is a part of the query result. Therefore, selection attribute values should be stored with query results in a cache. Upon a cache miss, WDBAccel rewrites a SQL statement by adding selection attributes to a projection attribute list.

3.3 Reducing the Cost of Query Result Derivation

Derived matching by selection region dependencies can identify many superfluous matching fragments. The selection regions of two or more matching fragments can redundantly cover the same query region. Furthermore, some matching fragments can include a large number of the tuples which are not part of a query result. If all those fragments go through the process of query result deriving, their processing costs will degrade the overall performance of WDBAccel. Therefore, it is important
to select fragments from which a query is derived with minimum cost. For example, assume that both \( F_1 \) and \( F_2 \) contain a query and consist of 10 and 20 tuples respectively. In this case, we should choose \( F_1 \) as it has the minimum cost. We call this problem the minimum cost fragment selection (MCFS) problem.

Let \( \mathcal{F} \) be a set of matching fragments for a query \( Q \). Given a fragment \( F \) in \( \mathcal{F} \), for query result derivation, all tuples in \( F \) go through both union and trim filters. In addition, each tuple in \( F \) is of the same type and hence incurs the same filtering overhead. Thus, for the cost of the fragment \( F \), we simply use the cardinality of the fragment as the cost function, i.e. \( \text{COST}(F) \) is the number of tuples in \( F \). Tuples belonging to different matching fragments are also of the same type. Hence, this metric can be used to compare the costs of different fragments.

We now formally define MCFS as follows:

**Minimum Cost Fragment Selection Problem:**
Given a query \( Q \) and a set of matching fragments \( \mathcal{F} \), find a subset \( \mathcal{F}' = \{F'_1, \ldots, F'_k\} \) of \( \mathcal{F} \) such that \( \bigcup_{i=1}^k \text{SR}(F'_i) \) covers \( \text{SR}(Q) \) and \( \sum_{i=1}^k \text{COST}(F'_i) \) is minimal, where \( \text{SR}(F) \) is the selection region of \( F \).

Unfortunately, the MCFS problem is \( NP \)-complete and therefore there are no polynomial time algorithms unless \( P = NP \).

**Theorem 1:** MCFS is \( NP \)-complete.

**Proof:** We prove that MCFS is \( NP \)-complete by reducing MCFS to a well-known \( NP \)-complete problem, Minimum Cost Set Cover (MCSC) [28]. MCSC consists of a finite set of elements \( U \) and a collection \( S \) of subsets of \( U \). Each subset \( S_i \) has a cost \( C_i \). The objective is to choose a minimum cost subset \( S' \) from \( S \) that covers all elements of \( U \).

Define \( Q \) to be the set of all cells in the partition formed by the perimeters of the fragments of \( \mathcal{F} \), and define each fragment \( F_i \in \mathcal{F} \) to be the set of the cells included in that fragment (see Figure 5). Now, MCFS is easily transformed into MCSC (in polynomial time) by considering \( Q \) as \( U \) and \( F_i \) as a subset of \( U \).

![Fig. 5 A reduction from MCFS to MCSC. \( Q = \{c_1, c_2, c_3\}, F_1 = \{c_1, c_2\}, F_2 = \{c_2, c_3\}. \)](image)

We have shown a reduction from MCFS to MCSC, and therefore MCFS is \( NP \)-hard. Since solutions for the decision problem (i.e. \( \sum_{i=1}^k \text{COST}(F'_i) < w \), where \( w \) is a positive constant) of MCFS are verifiable in polynomial time, it is in \( NP \), and consequently the MCFS decision problem is also \( NP \)-complete.

We present two algorithms, Greedy-MCFS and First Cover Fragment Selection (FCFS), for the MCFS problem. Greedy-MCFS is designed to reduce the cost of the query result deriving. On the other hand, FCFS is designed to minimize the cost of fragment selection itself. Greedy-MCFS has been devised from a known greedy algorithm for MCSC [28]. Similar to the MCSC algorithm, Greedy-MCFS iteratively selects the most cost-effective fragment until the query region is entirely covered. The cost-effectiveness of a fragment \( F_i \) is defined as the average cost incurred by \( F_i \) covering new regions, i.e. \( \frac{\text{size}(\text{SR}(F_i) \setminus \text{SR}(Q) \cup \text{SR}(M))}{\text{COST}(F_i)} \), where \( M \) is the set of fragments already selected at the beginning of an iteration and \( \text{size}(\text{SR}(F)) \) is the size of the selection region of \( F \).

```plaintext
// Greedy-MCFS (Q, F)
Q : input Query,
F : a set of matching fragments for Q
1. M ← ∅ // a minimum cost subset
2. while SR(M) ⊂ SR(Q) do
    Find F_i in F' such that \( \alpha(F_i) = \min_{F \in F'}(\alpha(F)) \),
    where \( \alpha(F) = \frac{\text{size}(\text{SR}(F) \setminus \text{SR}(Q) \cup \text{SR}(M))}{\text{COST}(F)} \),
    i.e. the cost-effectiveness of F.
    M ← M \cup F_i
    F' = F' \{F_i\}
3. Output the chosen fragments M.
```

FCFS is a much cheaper way to select a set of matching fragments that covers a given query. It linearly scans matching fragments until a query region is entirely covered.

```plaintext
// FCFS (Q, F)
Q : input Query,
F : a set of matching fragments for Q
1. M ← ∅ // a set of selected fragments
2. while SR(M) ⊂ SR(Q) do
    Pick a fragment F_i in F',
    M ← M \cup {F_i}
    F' = F' \{F_i\}
    i ← i + 1
3. Output the chosen fragments M.
```
It is intuitive to see that the time complexities of Greedy-MCFS and FCFS are \(O(N^2)\) and \(O(N)\) in the worst case, where \(N\) is the number of matching fragments. However, considering both fragment selection and query result derivation, we conjecture the overall cost will much depend on the number of matching fragments. When the number is small, Greedy-MCFS will perform better as it selects a set of fragments which incurs less overhead in the derivation step. However, as Greedy-MCFS scans all matching fragments at each iteration to choose the most cost-effective fragment, the fragment selection cost increases with the number of fragments. Thus, the benefit in result derivation step will be diminished by the increased cost in fragment selection as the number increases. FCFS is expected to outperform Greedy-MCFS with a high number of matching fragments. We compare the impact of the two algorithms over different cache sizes in Section 6 by experiments.

4. Technical Issues and Solutions

4.1 Fragment Indexing

Since many queries have their own unique selection regions, it is likely that many different fragments may be generated. In that case, it is important to speed up query matching over a large number of cached fragments. Figure 6 shows the structure of the Fragment Index. It reduces the search time by dividing fragments into a number of groups and indexing each group.

The Fragment Index is created in two steps. First, fragments are classified into query groups according to their query template. This step is necessary to reduce the search space before searching by selection regions as described below. Second, fragments in each group are indexed by their selection regions. If selection regions represent ranges, multi-dimensional spatial index structures such as an R-tree [13] and a CR-tree [15] are used. Those structures use hyper-rectangles as their keys and index them. A selection region can be viewed as a hyper-rectangle in a multi-dimensional space, and hence, is effectively indexed by spatial index structures. For example, a selection region 01/01/2003 < I_PUB_DATE < 1/20/2003 and 20 < I_COST < 30 is a rectangle of which the upper-left corner is (01/01/2003, 20) and of which the lower right corner is (1/20/2003, 30) in a two-dimensional space. Each dimension of the space corresponds to the selection attributes I_PUB_DATE and I_COST, respectively. By executing a range search over an index, we can quickly find the fragments matching a query. If selection regions represent points, a hash structure is used. Whenever a query belonging to a new group is arrived, the Cache Controller selects either a hash or a R-tree structure for the group.

4.2 Cache Storage

To increase the utilization of limited cache storage, it is crucial to avoid redundantly storing identical data. Query results can include identical tuples. As an example, assume that the selection regions of two queries are (10, 30) and (20, 40). The tuples located in the overlapping selection region (20, 30) are included in both fragments. We propose two storage policies as follows. The second approach identifies overlaps in selection regions and eliminates redundancies of tuples.

Using a Query Result as the Storage Unit. This storage policy uses a query result as the storage unit. It is simple and easy to implement. However, the redundancies cannot be removed. Throughout this paper, we refer to the Cache Pool adopting this policy as the Cache Pool in the unit of Query results (CPQ).

Using a Tuple as the Storage Unit. This policy stores query results in the unit of tuples. Before storing each tuple in a new query result, it determines if the tuple already exists in the cache. This is done by comparing the values of the selection attributes of each tuple with those of the cached tuples. To speed up the comparison, the policy scans only the tuples of the fragments which overlap the new query result. Note that these fragments have already been retrieved in the query matching process. We refer to the Cache Pool adopting this policy as the Cache Pool in the unit of Tuples (CPT).

CPT does not find all redundancies. A tuple can be included in two or more fragments even when their selection regions do not overlap. For example, a tuple with attribute values (I_ID=10, I_COST=20) is included in both a fragment with a selection region I_ID=10 and a fragment with I_COST=20 although their selection regions do not overlap. Such redundancies can be detected if we retrieve and compare the key values of tuples. However, this approach is more costly since it requires extra storage space for key values and their index structures as well as more management cost for them.
4.3 Cache Replacement

The hit ratio can be improved if the cache stores the query results which are frequently accessed and consume less storage space. Thus, we evaluate the profit of a query result as follows:

$$\text{profit}(f) = \frac{\text{popularity}(f)}{s_{\text{cost}}(f)}$$ (1)

where $s_{\text{cost}}(f)$ is the storage cost of a fragment $f$ and $\text{popularity}(f)$ represents the popularity of a fragment $f$. When the cache space is full, the Cache Controller evicts the query result with the lowest profit value (called the victim). Usually, the last access time or the number of accesses are used for popularity. When CPQ is used for the storage system, $s_{\text{cost}}$ is simply evaluated as the size of the fragment. Under CPT, we consider that some tuples are shared among multiple fragments. We divide the storage cost of a shared tuple among sharing fragments. In this case, $s_{\text{cost}}$ is computed as follows.

$$s_{\text{cost}}(f) = \sum_{t_i \in T(f)} \text{size}(t_i)/n_{\text{frag}}(t_i)$$ (2)

where $T(f)$ is the set of tuples belonging to a fragment $f$, $\text{size}(t_i)$ is the size of a tuple $t_i$, and $n_{\text{frag}}(t_i)$ is the number of the fragments containing a tuple $t_i$.

Such a policy exploiting the storage cost increases the exact hit ratio since it leads to storing more fragments with smaller size. However, it may decrease the derived-hit ratio. It evaluates large fragments as less profitable due to their high storage costs. However, larger fragments have higher probability of containing subsequent queries. We discuss this issue with experimental results in section 6.2.

In CPT, different tuples in a fragment may have different profit potentials. That is, shared tuples are more profitable than others. However, accurate measurement of profits on the level of tuples may be very costly and thus undesirable. We instead estimate the profit of a tuple by the number of fragments containing the tuple. Then, when a fragment is chosen for replacement, we evict only the tuples of which the reference count is zero. The reference count of a tuple becomes zero when all the fragments containing the tuple are evicted. To evaluate the profit of a tuple in this way, we should keep information about all queries requesting a shared tuple. Hence, upon a query which results in derived matching, we add the fragment information for the query as a new item to the Cache Dictionary. Then, we increase the value of $n_{\text{frag}}$ for each tuple of the new fragment.

The hit ratio can be further improved if the cache stores only read-oriented query results; the query results which are frequently updated are not kept long in the cache since they are evicted from the cache upon updates, not being much utilized. One possible approach to incorporate this consideration into the cache replacement is to count and use the update frequency of each fragment as a factor to evaluate profits. However, this approach requires to maintain the update frequency for each fragment, even previously evicted ones. It may flood the Cache Dictionary with a large number of the entries, degrading the performance of WDBAccel. Another approach is to statically identify frequently-updated tables in advance and not to cache the fragments derived from those tables. This approach efficiently identifies frequently-updated fragments by simply observing their base tables without burden to the cache system. As such, WDBAccel employs the latter approach. WDBAccel reports the information on the ratio of write queries to read queries for each table. With this information, an administrator can easily identify frequently-updated tables.

4.4 Cache Consistency Mechanism

The primary goal of our cache consistency mechanism is to ensure commitment consistency. Here we define commitment consistency as follows: once an update is completed at an origin database server, a cache always returns new versions of updated data. For some Web pages, such consistency is critical in order to always serve the most up-to-date version (e.g., the cost of products in an Internet shopping site).

WDBAccel employs an invalidation-based mechanism for ensuring consistency. Upon receiving an update query, a cache immediately performs the following process. The Consistency Maintainer (CM) sends an update query to an origin database server. Then, CM identifies the fragments affected by the update and removes them from the cache. [6] presents the details on identifying affected fragments. This invalidation process is executed also on the affected fragments that are being transmitted from a database server. After all the affected fragments are cleared, new versions of those fragments are delivered from the database server.

In order to guarantee commitment consistency, the following condition must be met: once all affected fragments are invalidated, data items to be updated, say $D_{\text{update}}$, must not be viewed by subsequent queries at an origin database server until completion of the update. It prevents staled data from being delivered to and left in the cache. For this, CM issues an update transaction to the database server before executing invalidation process. This scheme isolates $D_{\text{update}}$ from other queries. $D_{\text{update}}$ are locked and thus the queries for $D_{\text{update}}$ are blocked. After completing the invalidation and the database update, the update transaction is committed and $D_{\text{update}}$ are released in new versions. The queries will retrieve new versions of $D_{\text{update}}$.

In addition, we designed the consistency mecha-
nism to be deployed without modification to the configuration of existing Web sites. To accomplish this, we build CM as a module in a cache system and have the entire process of consistency maintenance performed by CM. After having received an update query from the Query Redirector, CM identifies affected fragments and invalidates those fragments. Note that our consistency mechanism requires update queries to be delivered through Web interfaces. That is, every update query affecting the status of cached results should be through WDBAccel. We believe this requirement is practically effective because many large-scale Web sites use Web-based content management tools.

5. Implementation

We implemented the prototype of WDBAccel which included all components and core functions described above. We also implemented the JDBC driver for WDBAccel. The WDBAccel prototype operated reliably in various experiments. We used GNU C++ and developed WDBAccel on the Linux. The current implementation of WDBAccel uses Oracle 8i database server as an origin database server. To communicate with Oracle 8i, we used Oracle Call Interface (OCI) which is Oracle’s call-level interface.

All data structures related to Cache Dictionary are stored in main memory to speed up cache lookup. The metadata such as the name and the data type of attributes are retrieved from the origin database server and stored in Cache Dictionary. Then, they are shared by all fragments belonging to the same query group. As mentioned in section 4.1, WDBAccel uses spatial index structures such as an R-tree and a CR-tree to index selection ranges of fragments. An R-tree is a representative index structure for range objects. A CR-tree is a variant of an R-tree optimized to main-memory settings. For the current implementation, we used an R-tree due to its convenience of getting open sources. In order to hash strings, we used a modified MD5 [22] hash function.

For the Cache Pool, we implemented both CPQ and CPT. In CPQ, the size and the number of tuples of a query result are stored with each query result. In CPT, the size and the number of the containing fragments of a tuple are stored with each tuple.

6. Experiments

In this section, we discuss the experimental results of WDBAccel from various points of view. First, we compare the performance of WDBAccel to that of the Oracle database server and the TimesTen/Cache. Comparison is done through measuring system throughputs and response times. Second, we evaluate the effect of fragment selection algorithms on performance by measuring their execution times. Third, we evaluate the effectiveness of the derived matching.

6.1 Experimental Setup

6.1.1 Workloads

We evaluated the performances using the TPC-W benchmark suite. As noted earlier, the TPC-W benchmark is a de facto industrial standard benchmark for the performance evaluation of database-driven Web sites. The workload of the TPC-W benchmark includes search queries with book title or author name keywords, queries about best-selling books, and queries about detailed information on a particular book, which are expressed as TOP-n, LIKE queries as well as simple equality selection and join queries. We used the TPC-W benchmark implementation made by a team at the University of Wisconsin [16].

We also used range queries, which are frequently used in many applications, such as travel package searches by date, stock searches by cost, and book searches by publication dates. Since the TPC-W benchmark does not contain range queries, we synthetically generate the search-by-publication-date query and its trace as follows.

```
SELECT I_TITLE, I_ID, A_FNAME, A_LNAME
FROM ITEM, AUTHOR
WHERE I_A_ID = A_ID AND
  I_PUB_DATE >= @Lower_Date AND
  I_PUB_DATE <= @Upper_Date
ORDER BY I_TITLE
```

Each such query includes a range predicate for the attribute ‘publication date’, i.e. \( \leq I\_PUB\_DATE \leq \) \( @Upper\_Date \), and thus, the selection regions of different queries can overlap with each other. We assume that \( @Upper\_Date \) in the predicate follows the Zipf distribution from January 1, 1970 to December 1, 2000. This is a realistic assumption as many users prefer new books. The ranges of the selection predicates can span at most a month and follow a uniform distribution. We refer to the search-by-publication-date query as the range query. In our experiments, the range query is used to evaluate the effectiveness of the various techniques proposed in this paper as well as the overall performance of WDBAccel.

6.1.2 Machine Setup

Figure 7 shows the experimental setup. It is composed of four components: a Remote Browser Emulator (RBE), a WAS, the WDBAccel or TimesTen/Cache, and the origin database server (Figure 7 (a)). When measuring the performance of the Oracle server, the WAS is directly connected to the Oracle server (Figure 7 (b)).
and denoted each case as WDBAccel-1, WDBAccel-10, WDBAccel-100, respectively. Among the tables in the TPC-W database, ITEM and AUTHOR are most frequently accessed. Thus, the cache pool is mostly filled by the query results generated from those two tables. The total sizes of the two tables are 10 MB and 100 MB for 10K and 100K scales of the TPC-W database, respectively. The average fragment size was 884 bytes. The number of fragments stored in the cache ranged approximately from 110 to 110,000 depending on the cache size. We used CPF as the storage policy and LFU as the replacement policy, and disabled derived matching. TimesTen/Cache is designed to load an origin database to its main-memory cache in the unit of tables. We populated the TimesTen/Cache with the ITEM and the AUTHOR tables (which are most frequently accessed among TPC-W tables), and additionally the CUSTOMER table. The size of main-memory allocated to load these tables is 104MB and 195MB at 10K and 100K TPC-W scales, respectively. We populated the Oracle with the entire TPC-W database. The memory size of the buffer pool for Oracle is set to 256MB. This size was attentively selected considering that the throughput of a database server may be affected by the size. This size of 256 MB is large enough to keep most of the accessed data both at the 10K and 100K TPC-W scales and to minimize disk accesses. Also, note that main memory size allocated even for the WDBAccel-100 is much smaller than that for the TimesTen/Cache and for the Oracle. Additionally, for the Oracle and the TiemsTen/Cache, we manually checked processing plans of the individual queries used in the TPC-W measurements, and, to optimize the performance, intentionally selected the best query processing plan if it was not selected by the systems.

Figure 8 (a) shows that, at the 10K scale, the throughput of WDBAccel increases with the cache size since cache misses decrease. In the experiments of WDBAccel, the origin database server was the bottleneck. In that case, the miss ratio is an important factor in throughput. A cache miss incurs query processing for missed query at the origin database server. Thus, a lower miss ratio means less processing load on the origin database server, increasing the throughput of WDBAccel. WDBAccel-100 outperforms both the TimesTen/Cache and the Oracle, while the throughputs of the WDBAccel-1 and WDBAccel-10 are between those of the TimesTen/Cache and the Oracle. This implies that, upon cache hit, the processing cost of WDBAccel is lower than that of TimesTen/Cache and the Oracle. The performance difference becomes more prominent in the 100K scale. In the case, even WDBAccel-1 outperforms the TimesTen/Cache and the Oracle. This implies that the cost of query processing in the Oracle and the TimesTen/Cache is more sensitive to the size of databases than in WDBAccel since the processing in WDBAccel is just simple query

<table>
<thead>
<tr>
<th>Table</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITEM</td>
<td>10</td>
</tr>
<tr>
<td>AUTHOR</td>
<td>100</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>104</td>
</tr>
</tbody>
</table>

---

\(^1\)In TPC-W, the scale of the database is determined by the cardinality of the ITEM table.
Throughput under range query

Figure 8 (b) shows the throughputs when range queries are used as the input trace. In this experiment, we varied the cache pool size of WDBAccel to 1%, 5%, and 10% of the sum of ITEM and AUTHOR table sizes. We used CPT as the storage policy and LFU as the replacement policy, and enabled derived matching with Greedy-MCFS. Those settings are best for range queries, which will be shown in subsequent experiments. The average fragment sizes at 10K and 100K TPC-W scales were 722 and 6263 bytes, respectively. The number of fragments ranged approximately from 10,000 to 30,000. The R-tree was used for indexing range fragments. The maximum number of buckets in a node of the tree was set to 5, which is attentively selected for fitting a node within the order of a block size of a CPU cache. The height of the tree ranged from 6 to 7. The settings for TimesTen/Cache and Oracle are the same as those for the previous experiments. We first warmed up the WDBAccel cache by sending 100,000 range queries. We then measured the throughput by sending another trace of 300,000 range queries. The results exhibit that WDBAccel achieves higher throughput than the TimesTen/Cache and the Oracle.

Response times: WDBAccel vs. TimesTen/Cache

We measured response times for queries to compare query processing costs of WDBAccel and TimesTen/Cache. We classified TPC-W read queries according to their types, i.e., equality selection (EQ), equality selection with join (EQ/J), LIKE selection with join and TOPN (LS/J/TOPN), and equality selection with join and TOPN (EQ/J/TOPN), and computed the average response times for each query type. We also measured the response times for queries involving range selections, i.e., range selection (RS) and range selection with join (RS/J). Measurements were performed only for cache hits.

The results of response times in Figure 9 show that the WDBAccel significantly saves processing costs for the queries with expensive operators. The figure shows that the response times for the queries with LIKE predicates (LS/J/TOPN) show the most significant difference. This type of queries constitutes 20% of the entire TPC-W workload. The difference becomes more significant at the 100K scale. As mentioned in Section 3, the LIKE selection requires a linear search on a table in TimesTen/Cache.

The queries of ES/J/TOPN type, which constitutes 10% of the TPC-W workload, is another important factor of the performance difference. This type of queries executes the ORDER BY operator. The ORDER BY operator executes in $O(n \log n)$ where $n$ is the number of tuples. Thus, it is very costly when the number of tuples is big as shown in Figure 9 (b).
Performance Effect of the Fragment Selection Algorithms

In order to study the performance effect of the fragment selection algorithms, Greedy-MCFS and FCFS, we measured the execution times of the algorithms for 50,000 range queries in a TPC-W database with a 100K scale. To determine the effectiveness of the algorithms, we also measured the execution times of query result deriving with the fragments selected by each fragment selection algorithm.

Figure 10 shows the results with the cache sizes varying from 1% to 5% of the sum of \texttt{ITEM} and \texttt{AUTHOR} table sizes. The results show that Greedy-MCFS outperforms FCFS. This is due to the low cost of query result deriving (Greedy-MCFS (QRD) in the figure) with the fragments selected by Greedy-MCFS. This result implies that when the cost of the query result deriving is sufficiently high (thereby reducing the deriving cost is important), Greedy-MCFS is better. Note that the cost of query result deriving is high when the number of tuples in a fragment is large. The more the number of tuples in a fragment, the more frequently filtering occurs.

The results also show that, compared to FCFS, Greedy-MCFS does not scale well with cache size. This is because the running time of Greedy-MCFS (Greedy-

Impact of the Derived Matching

In order to study the impact of derived matching on cache effectiveness, we ran experiments with the TPC-W database set to a 100K scale and cache sizes ranging from 1% to 25%. We used CPT storage policy for these experiments. We measured hit ratios for three replacement policies, LRU, LFU, and LFU-Size while issuing 50000 range queries. LFU-Size is a variant of the LFU policy which considers the storage cost of query results with the \textit{profit} measure,

$$profit(f) = \frac{\text{freq}(f)}{s_{\text{cost}}(f)}$$

where \text{freq}(f) is the number of accesses to a fragment \(f\) and \(s_{\text{cost}}(f)\) is the storage cost of a fragment \(f\). \(s_{\text{cost}}(f)\) is evaluated as mentioned in section 4.3.

Figure 11 (a) shows that the derived matching (DM) improves the cache hit ratio consistently with all cache sizes. Somewhat surprisingly, with cache sizes up to 10%, LFU-Size exhibits a lower hit ratio than both LRU and LFU. This is because with small cache size and hence, high contention for cache storage, LFU-Size achieves a lower derived-hit ratio than others, as shown in Figure 11 (b). LFU-Size tends to evict large fragments, which have a higher possibility to contain subsequent queries. Consequently, even though the policy achieves a high level of an ‘exact’ hit ratio, the achieved
7. Related Work

Recently, much research has been reported on various cache systems to support dynamic Web content services. As shown in Figure 12, existing caching systems can be classified into three categories, namely, HTML caching, query result caching, and table/view caching according to the stage of the content generation at which data are cached.

7.1 HTML Caching

A commonly-used HTML caching selects cacheable data in the unit of the whole HTML page or HTML components which are parts of a HTML page. It then caches the data in front of Web servers or inside WAS’s. We call the former HTML page caching and the latter HTML component caching. The main advantage of this approach is that the performance gain on a cache hit is better than that of others. This is because the HTML caching saves much or all of the cost of HTML page generation as well as database processing. Another advantage is that much of existing, mature Web-caching-related technologies (e.g., well-established cache replacement and cache consistency mechanisms [29], and lots of HTTP implementations) can be applied to this approach and make its implementation relatively easy.

**HTML page caching.** Upon a cache hit, the HTML page caching achieves the largest performance gain. It is so because by caching final results, i.e. HTML pages, contents can be served without any processing for page generation. However, unlike other approaches, the HTML page caching is not effective in caching dynamic pages. Consider a personalized Web pages. A personalized page is a type of Web page that is tailored to individual users’ characteristics or preferences and is commonly used in many Web sites. Thus, there usually exists a huge number of similar pages which are still different for each user, and hence, it is not meaningful to cache such pages. Even when a page is cached, the hit ratio will be very low since it will be accessed by only its own user. The systems proposed in [14], [4], and [27] come under this category.

**HTML component caching.** This approach caches HTML components which are portions of a HTML page. It can be effective to some extent in caching dynamic contents. As for a personalized page, a lot of pages may have common components. For example, a personalized page for a stock quote or a weather forecast may show the same components, e.g., tables for stock prices or weather forecast, with different settings and personalized data for each client. After dividing a page into common and personalized components, common components are cached. Then, cached components can be reused in generating many pages for different users.

A disadvantage of this approach is that the cache administrator should go through a complex process of marking cacheable and non-cacheable units. The boundary of HTML components is not explicitly demarcated in HTML pages. Thus, the administrator should examine every page and demarcate cacheable units and appropriately modify existing documents and
applications. Considering that there could be huge number of pages in a site, this may require a lot of work. This process should be repeated for each page newly added to the system. The systems proposed in [5], [10] and [9], and Oracle Web Cache [2] come under this category.

7.2 Table Caching

This approach stores data in the unit of tables at between WAS’s and database servers, and process queries from WAS’s on behalf of database servers. Compared with the HTML caching, table caching (and also view and query result caching) provides easier deployment and lower administrative cost. It is mainly because it is easier to specify which data are to be cached. The administrator chooses read-oriented, frequently-accessed data in the unit of tables. Also, the modification of the set of selected tables is rare because a database schema is not frequently modified.

This approach can degrade cache effectiveness because it caches tables in their entirety which can include unpopular regions. Usually, this approach uses a conventional DBMS as the cache storage module. Thus, its performance is limited by complex query processing of the underlying DBMS. Oracle9IAS Database Cache [20], TimesTen/Cache [25], and DBCache [17] come under this category.

7.3 View Caching

This approach [3], [12], [19], [31] has been studied in the context of view materialization in Data Warehouses. The approach is close to our fragment processing in that it reuses previously processed results. However, their primary focus differs from that of fragment processing. This difference comes from a huge number of fragments (which correspond to materialized views). Due to a large number of fragments stored in a cache, one important goal of WDBAccel is to speed up deriving a result of a given query from numerous fragments. For this, we propose the Fragment Index and the fragment selection algorithms. This issue is not considered in Data Warehousing in which the number of views is relatively small. Rather, research on materialized views has focused primarily on view selection and maintenance. WDBAccel also considers those issues, but employs different solutions for them. WDBAccel dynamically selects views relying on incoming queries. In contrast, the materialized view statically selects views for a given workload. The invalidation-based consistency management of WDBAccel is different from the materialized view approach in which views are not invalidated, rather updated. Like table caching, this approach uses a conventional DBMS as the cache storage module. Thus, its performance is determined by the underlying DBMS.

7.4 Query Result Caching

The query result caching stores and reuses query results for subsequent queries. Compared with the table/view caching, this approach provides two advantages: high cache effectiveness and high performance. The query result caching improves cache effectiveness by caching only frequently-accessed regions of databases. Simultaneously, high performance is achieved by reusing previous query results and thereby eliminating complex query processing. Form-based Proxy Caching [18], Weave [30], and our WDBAccel come under this category.

Form-based cache [18] is the first effort for query result caching. It extended the URL-based proxy caching for active proxy query caching with limited query processing capability. The proposed framework could effectively work for a top-n conjunctive queries generated from the HTML forms. However, it only addressed keyword queries, whereas WDBAccel supports a broad range of query types including range queries and employs query matching and optimization techniques for processing those types of queries. Weave [30] caches data in the unit of XML and HTML pages as well as query results. Weave focuses on the declarative specification for Web sites through a logical model and a customizable cache system that employs a mix of different cache policies.

DBProxy [1] is somewhat a hybrid system that combines the query result caching and the table/view caching. DBProxy is a database proxy located in edge-of-network and addresses self-managing features to reduce administration cost. The main purpose of DBProxy is to bring the source of data closer to the clients, and thereby reduce delay in service. It determines data to be cached in the unit of query results as the typical query result caching does. However, like table/view caching, it employs a conventional DBMS for storing and retrieving query results as a table/view caching does. Therefore, its performance could be limited by the complex query processing of the underlying DBMS.

8. Conclusions

We have presented the design and implementation of a high-performance database server accelerator. WDBAccel improves the throughput of database processing, which is a major bottleneck in serving dynamically generated Web pages. To improve the performance, it reuses previous query results for subsequent queries and utilizes main memory as a primary cache storage. WDBAccel performs fragment processing to efficiently generate a query result from cached query results. During the fragment processing, WDBAccel uses an index
structure to speed up finding matching query results. It also executes the fragment selection algorithms to reduce the cost of generating query results. WDBAccel employs the storage policy that reduces storage redundancy. The cache replacement policy takes into account storage costs of query results and overlaps among them. In addition, WDBAccel includes the consistency mechanism to ensure that it does not return old versions of data. The experimental results show that WDBAccel outperforms DBMS-based cache systems by up to 16 times.

Acknowledgments

We would like to thank KyungHyun Cho who helped us while we were evaluating the performance of WDBAccel. We also thank Sungwon Choi and Sungjae Jo for invaluable advice on writing this paper.

References


in Computer Science from the KAIST in 1999. His research interests include service oriented computing, sensor network, and high-performance Internet cache system.

Jinwon Lee is a PhD candidate at the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea. He received the B.S. degree in Electrical Engineering from the KAIST in 2001, and M.S. degree in Computer Science from the KAIST in 2003. His research interests include data stream processing system, peer-to-peer overlay network, and high-performance Internet cache system.

Su Myeon Kim received his B.S. and Ph.D. in Electronic Engineering and Computer Science from the Korea Advanced Institute of Science and Technology in 1997 and 2006. He is currently a research and development staff member at Samsung Advanced Institute of Technology. His research interests include ubiquitous computing, stream processing, service oriented computing, and internet computing.

Junehwa Song is an associate professor, Department of EECS, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea. Before joining KAIST, he worked at IBM T.J. Watson Research Center, Yorktown Heights, NY as a Research Staff Member from 1997 to September 2000. He received his Ph.D. in Computer Science from University of Maryland at College Park in 1997. His research interest lies in Internet Technologies, such as intermediary devices, high performance Web serving, electronic commerce, etc., and distributed multimedia systems.

Yoon-Joon Lee received his B.S. degree in Computer Science from Seoul National University, Seoul, Korea, in 1977 and his M.S. degree in Computer Science from KAIST 1979. And he received his Ph.D. degree in Computer Science from INPGENSIMAG, France, in 1983. In 1984 he joined the faculty of the Department of Computer Science at KAIST, Taejon, Korea, where currently he is a professor. His research interests include database systems, Internet, Web databases and distributed systems.