EZDB: A Framework for Easy Evaluations of Commercial Applications

Kangwoo Lee, Jigar Thakkar and Michel Dubois

CENG 97-23

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4475

EZDB: A Framework for Easy Evaluations of Commercial Applications

Abstract

Currently computer architectures are evaluated almost exclusively with scientific applications. Commercial applications are rarely explored because it is difficult to obtain the source codes of commercial DBMS (Data Base Management System). Even when the source code is available, such as for POSTGRES95 [10], understanding the source code enough to perform detailed and meaningful performance evaluations is a daunting task.

We have developed our own DBMS called *EZDB*. EZDB is a parallelized DBMS loosely inspired from POSTGRES95 and running on top of a software architecture simulator. It is capable of executing parallel programs written in SQL. Contrary to POSTGRES, EZDB is not intended as a prototype for a production-quality database systems, but rather its purpose is to easily run and evaluate the performance of commercial applications on architectures. EZDB is modular and can be incrementally improved to run more complex applications. Currently, enough of the database system has been written to execute the Transaction Processing Council (TPC) benchmarks.

To illustrate the use of EZDB we show data collected for the simplest TPC benchmark, i.e. TPC-B, on a cache-coherent shared-memory system model. The simulation results show that the data structures exhibit unique sharing characteristics and that their locality properties and working sets are very different from those in scientific applications.

1 Introduction

Understanding the access patterns of shared data is key to obtaining good performance on shared-memory systems. In this respect, scientific applications are fairly well understood, since the development of the SPLASH [9] benchmarks at Stanford. However, since parallel systems are mostly used for commercial applications, researchers in architecture are eager to explore the behavior of parallel commercial workloads. Unfortunately, DBMS programs are usually proprietary. Recently, a public-domain DBMS (Data Base Management System) called *POSTGRES95* [10] was released. However, the large number of complex data structures, the complicated locking schemes and the very large size of the code make it difficult to exploit for performance evaluations. Although reports describe the POSTGRES95, the information in them are for general DBMS users, not for computer architects. As a result, exploiting POSTGRES95 as a research tool in architecture is a daunting task.

Due to these reasons, relatively few results have been published for commercial applications. Suggs and Reynolds [11] presented the instruction fetch statistics of a uniprocessor TPC-B [7] workloads measured on the Motorola 88110. Torrellas *et al* [13] and Bhandarkar [1] evaluated commercial workload for the SGI multiprocessor and a DEC Alpha AXP system. In [8], the performance of a shared-nothing architecture is investigated using synthetic workload and actual traces. The shared-nothing architectures are preferred to shared-memory systems in [5]. The role of IO subsystems in shared-memory systems was investigated in [12]. In [14], the authors used POSTGRES95 to show the memory performance of some queries in TPC-D for various cache and cache line sizes on a shared-memory model. They categorized the data structures into private data, database data, metadata and index data.

To be able to obtain more results more easily, we have developed our own DBMS called *EZDB*. EZDB is a parallelized DBMS loosely inspired from POSTGRES95, which is capable of executing parallel programs written in SQL. Like POSTGRES95, EZDB is an in-memory relational database and makes no use of operating system facilities. EZDB runs on top of a software simulator and thus all events happening during the execution on a simulated architecture can be traced. As in the SPLASH benchmarks, the parallel runtime system is provided by the ANL macros [2]. Currently, enough of the database system has been written to execute the Transaction Processing Council (TPC) benchmarks. EZDB is not intended as a complete, usable database system, but rather its purpose is to provide researchers in architecture with detailed performance information, including the memory behavior of each individual shared data structure. In this paper, the first goal is to describe the structure and the design of EZDB. The second goal is to illustrate its application on TPC-B and show various statistics on the memory behavior of TPC-B [7]. The cache size, cache block size and number of processors are changed to investigate their impacts on the number of misses for each individual data structure separately.

This paper is organized as follows. EZDB and its parallelization are described in Sections 2 and 3, respectively. The TPC-B benchmark is briefly described in Section 4. Section 5 presents the simulation methodology. Simulation results are presented in Section 6 along with detailed analysis. We propose a methodology to expand traces obtained with EZDB in Section 7 and pointing to our future plans for EZDB in Section 8.

2 Design of EZDB

Generally, in a database system, the *database* is the collection of data records. The database *operations* specify how to define, construct and manipulate the records in the database. The *database management system* (DBMS) physically executes the database operations on the database records while enforcing various types of database constraints. There are two types of users: general users and database administrator (DBA). The DBA has a privilege to directly access the database definition contained in the *metadata* (see Section 2.4.1). In EZDB, we only simulate the general user queries provided in a script file. The architecture of a general DBMS is shown in Figure 1. The black boxes indicate the components implemented in EZDB; the components in the white boxes are not implemented as yet.

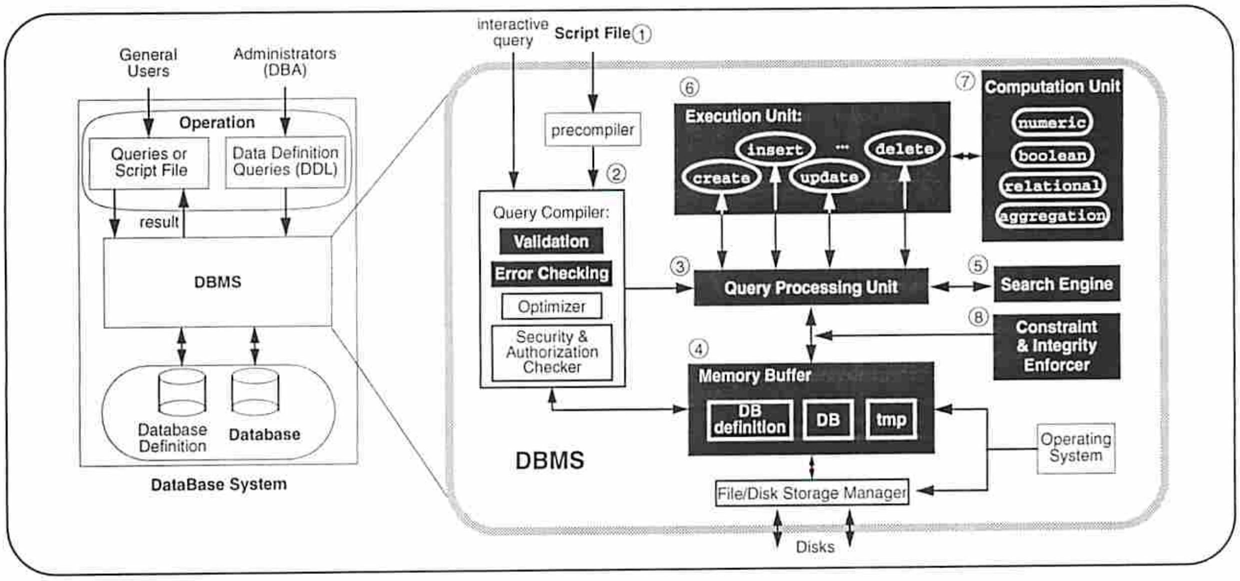


Figure 1. Architecture of a DBMS.

As shown in Figure 1 the user writes a script file made of consecutive queries to the database and compiled by the query compiler. The compiled program is then executed by the query processing unit, the execution unit, the computation unit and the search engine. The data structures include the metadata, the database data, and the index data. In the following we describe the parts of EZDB that we have implemented. We use the terms *query statements*, *query instructions* and *query operations* to mean the program statements appearing in the script file, the SQL-like commands derived from the query statements, and the activities in a query instruction, respectively. Usually, the query instruction contains a set of *query parameters* to specify the table names, the set of conditions, the set of numeric, boolean, relational and/or aggregation computations, and the temporary storages for query results.

2.1 Script File

The script file is basically a C program in which a sequence of query statements are embedded and it is compiled by *gcc*. In line 2 of Figure 2 (a), a *query statement* is passed by a built-in function *PQexec*()[1] to the DBMS and will be compiled by the query compiler. Note that when a variable (such as `i` in Figure 2 (a)) is included in the query statement, the DBMS has no means to know its value since it is defined outside the query. The way around this problem is shown in Figure 2 (b). The query statement in which the variable has been replaced by a numerical value is stored into a temporary character string (`char *command`) which is then passed to the DBMS through *PQexec*() (lines 3, 4).

```
1.        for(i=0;i<N;i++)
2.            PQexec("INSERT INTO table_A VALUES (:i,0);");
```

(a)

```
1.        char *command;
2.        for(i=0;i<N;i++) {
3.            sprintf(command,"INSERT INTO table_A VALUES (%d,0);", i);
4.            PQexec(command);
5.        }
```

(b)

Figure 2. Example of a script file.

2.2 Query Compiler

In general, three compilers are needed in a DBMS. One is for DDL (Data Description Language) code used for DBA queries. No DDL compiler is included in EZDB. Another compiler is for the programming language used in the user script file. Our script file is written in C and compiled with *gcc* (Section 2.1). Additionally, we have developed a query compiler to translate a query statement in a script file into a *query instruction* with its *query parameters*. This compiler accesses the information stored in the metadata for validation purpose. The output of the query compiler is then passed to the query processing unit which actually executes the query.

In the current version of EZDB, the query compiler does parsing, as well as syntax and error checking. Error correction, warnings or help features are not supported. This omission does not impact the performance results since the results are only meaningful when the script file is correct. Other features such as query optimization, and security and authorization checking proce-

1. This is a common way to interface a script in C with the query compiler. In Illustra, the script file passes the query statement by invoking a function called *mi_exec()* which is equivalent to *PQexec*() is in POSTGRES95 and in EZDB.

dures, which are not currently implemented, will be added.

2.3 Query Processing Unit

At the core of EZDB, the query processing unit orchestrates the query operations to execute a query instruction. It controls not only the order of query operations but also the accesses to metadata, database data and index data. Typically, given a query instruction and set of query parameters, the query processing unit may

- first access the metadata to obtain the information regarding the database tables, records and attributes,
- invoke a program module in the execution unit which corresponds to the query instruction such as create, insert, and delete,
- invoke the search engine to quickly identify the locations of a set of records that satisfy the conditions in query parameters,
- perform a sequential search over the database records in a table,
- directly access the database data to retrieve or to modify them, and
- finally, add or remove some nodes in index data structures according to the query instruction types.

2.4 Memory Buffer

Usually, the metadata and database are stored on hard disks. By contrast, in EZDB, they are stored in memory buffer in main memory. The memory buffer is also used by private or temporary variables needed in EZDB program modules. Note that the operating system would normally be in charge of file IO and memory management (allocation and release). However, since EZDB keeps all data in memory and our simulation environment can only monitor the activities of user programs, no operating system activity is included in our simulation.

2.4.1 Metadata

The *metadata* is itself a small database which stores the data describing each database. It includes a description of the conceptual database schema, the internal schema, any external schemas and the mappings between the schemas at different levels [6]. The metadata are mainly accessed during query processing. However, the query compiler uses them in validation procedure and the constraint and integrity enforcer accesses the metadata to maintain the database in a correct state.

(a) TableHead:

no_of_tables
3

(b) TableTable:

table_num	table_name	num. recs	num. fields	num. primary keys	primary key fields
0	table-1	10	3	1	pkey-1
1	table-2	50	4	1	pkey-2
2	table-3	10,000	4	1	pkey-3

(c) RecordHead:

table_number	no_of_fields	data type 1	field 1	data type 2	field 2	data type 3	field 3	.....
0	3	int	pkey-1-1	float	xxx	char	...	-
1	4	int	pkey-1-2	int	yy	float	...	-
2	4	int	pkey-1-3	int	zzz	float	...	-

Figure 3. Metadata in EZDB.

In EZDB, a set of data structures are defined as metadata and are shown in Figure 3. The **TableHead** stores the number of tables existing in the database. Every time a new table is created, this number is incremented by one. The **TableTable** contains the list of table names, the number of records in each table and the number of fields in each record of each table. It also keeps the number of fields that form a primary key and the list of these fields. The **RecordHead** contains the name of the fields and data types in tables. It also stores the number of records in each table.

2.4.2 Database

The database is a collection of tables containing the records. Each record is made of a set of attributes. The records in tables are stored in arrays and will be retrieved or modified at users' request. The table in Figure 4 which is called a **RecordTable** is a database table containing its records. It includes a *delete bit*. Whenever a record is deleted, this bit is set to 1. Physical removal of a record takes place only when we commit or exit.

RecordTable:

delete bit	field 1 (bid)	field 2	field 3	field 4
0	1	105000	LA_CA90007	-
0	2	200100	LA_CA90008	-
1	3	30545	LA_CA91101	-
0	4	56550	LA_CA90011	-

Figure 4. Database data in EZDB

2.4.3 Temporary Data

As shown in Figure 1, the DBMS must be able to store the query results and return them to users. For example, a set of attribute values in selected records should be provided after the completion of the *select* operation. In EZDB, these values are stored in a temporary storage at the end of the *select* module and are mapped in the script file onto the corresponding user variables.

By contrast, in case of the *cursor* operation, the result is a small database table, called cursor-table, with a specific name usually given in the script file. As for other database tables, another data structure, called the cursor-metadata, is created to store the information regarding the cursor-table. The cursor-metadata is accessed before a query can access the records in a cursor-table. The information in cursor-metadata includes

- name of the cursor table,
- cursor table number,
- query id,
- number of records in the cursor-table,
- variable (attribute) names,
- cursor delete bit,
- pointer to the current record, and
- name of the field used in *order by* operations.

In EZDB, a certain amount of shared address space is reserved for these temporary data so that, if subsequent program statements or queries are executed by different processors, the correct values can be supplied.

2.5 Index Search Engine

The basic task in a database system is the search operation. Index data structures are used to speed up the retrieval of records in response to certain search conditions. We use B-trees to implement dynamically changing multilevel indexes. The number of B-trees is equal to the number of tables and the number of nodes in a B-Tree is equal to the number of records in a table.

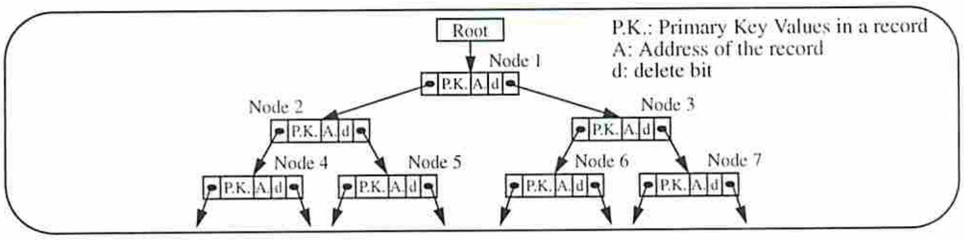


Figure 5. A B-Tree.

In B-trees, a node contains the value(s) of primary key(s), and has two children (Figure 5). The node also stores the address of the record whose key values are stored in it. The left child of a node points to a node with a lower primary key value(s) and similarly the right child points to a node with a higher primary key value(s) as compared to the current node. When a record is inserted, a new node is created in the appropriate B-tree at the appropriate position while, when a record is deleted, the delete bit in an appropriate node is set to 1. Like the records in a `Record-Table`, the deletion of a node is physically done when we close the database.

2.6 Execution Unit

The execution unit executes query instructions. Whenever necessary, it invokes the computation unit for numeric, boolean, relational and/or aggregation operations. The information stored in the metadata and in the database are provided by the query processing unit. The results of a query instruction is stored in temporary storage and passed to the query processing unit. In EZDB, this unit can execute all standard SQL-like instructions including basic instructions such as insert, delete, select, update as well as complex instructions such as cursor, fetch, group by, and order by.

2.7 Constraint and Integrity Enforcer

EZDB supports various types of constraints and integrities that are required to maintain the database in a correct state. In this section, we introduce the definitions of a set of representative restrictions and how they are satisfied in EZDB. Clearly, there are more instances of constraints and system integrity requirements. Even though we do not enumerate all of them, we implement our database operations so that they satisfy the existing system integrities.

First of all, the transactions are *atomic*; all individual transactions must assure that no partially completed operation leaves any effects on the data. For this requirement, each transaction is serviced completely by a processor without being preempted by others. The *consistency* property means that any execution of a transaction must take the database from one consistent state to another. For this, we first run the validation procedure at the query compilation stage and check the domain constraints and data integrities. The *isolation* property requires that concurrent transactions yield the same results as if every transaction was executed serially. To execute concurrent transactions, EZDB is parallelized as described in Section 3. To ensure the durability constraint, the tested system must preserve the effects of committed transactions and ensure data consistency after recovery from any hardware failures.

Another set of representative requirements that must be guaranteed by a DBMS are the domain constraints for the values taken by a given attribute, the key constraints which enforce that no two records have exactly the same values in all of their attributes, the entity integrity constraints which require that all primary keys must take non-null values and the referential integrity constraints which stipulate that tuples in a relation referring to another relation must refer to an existing tuple in that relation [6]. All these requirements are properly satisfied in EZDB. For example, the *insert* operation checks for any record with the exact same values as the one inserted. The *delete* operation checks whether the deletion of a record breaks the referential integrity.

2.8 Summary

Table 1 summarizes the features of general commercial DBMS as well as the ones implemented in EZDB.

Items		General DBMS	EZDB
Level of Users	Users	DBA & General Users	General users
	Query Language	DDL & General Query Language	Our own SQL
	Special Command	Privileged Commands for DBA	Not supported
Query Types		Interactive & Script File	Script File
Compilers	DDL Compiler	DDL Compiler	No
	Precompiler	Host machine-dependent	gcc
	Query Compiler	Validation	Implemented
		Error Checking	Implemented
		Query Optimizer	Not implemented
		Security & Authorization	Not implemented
Query Processing	Query Execution	create, insert, delete, select, update, roll back, group, order, *etc.*	Implemented
	Computation	numeric, boolean, relational, aggregation	Implemented
Data Structures	Data Definition	Metadata	Implemented
	Index Data	B-tree, B^*-tree or B^+-tree	B-tree
	Database Data	Relational or Object-oriented	Relational
Constraints & Integrity		Enforced	Enforced
Storages	Disk Storage	Hard Disk	Memory Buffer
	Disk Manager	Operating System	No
	Memory Storage	Memory	Memory Buffer
	Memory Manager	Operating System	No

Table 1. Database system components (shaded components are implemented in EZDB).

3 Parallelization

To simulate a concurrent transaction processing system, a parallel version of EZDB has been developed. Of course, the script files must be parallelized as well. The runtime system supporting this concurrency is the ANL macros. To run parallel programs on shared-memory systems, three mechanisms must be provided; creation of processes, declaration of shared variables and their address spaces, synchronization of accesses to shared data. To see how the parallelization is done, let's first consider the script file for the TPC-B in Figure 6 (a) and its parallel version in Figure 6 (b). `Bid`, `Tid`, `Aid` and `Delta` are randomly generated input parameters.

In conformance with the ANL macro model, a (master) process runs first to initialize the execution. Then it creates a set of slave processes and all processes enter the parallel section. Each process executes a common code image to perform concurrent transaction processing (body of the `for`-loop). The couple of private variables, `start` and `end`, specify the number of transactions to be executed by each process. During the initialization, the metadata, index data and database data are declared as shared data so that all update operations on them can be correctly reflected when other processes access them, later. The ANL macro used for this purpose is the `G_MALLOC` which is not shown in this example. All other variables are considered as private.

```
BEGIN TRANSACTION
     Update Teller where Teller_ID = Tid
          Set Teller_Balance = Teller_Balance + Delta
     Update Branch where Branch_ID = Bid
          Set Branch_Balance = Branch_Balance + Delta
     Update Account where Account_ID = Aid
          Set Account_Balance = Account_Balance + Delta
     Insert to History
          Tid, Bid, Aid, Delta and Time_stamp
COMMIT TARANSACTION
```

(a) A transaction in TPC-B

```
main() {
      MAIN_INITENV()
      initialize_by_master();
      for (i=0;i<P;i++)    /* P : number of processes */
         CREATE (Slave)
      BARRIER
      Parallel_Section();
      BARRIER
      MAIN_END
}
Slave() {
      BARRIER
      Parallel_Section();
      BARRIER
}
Parallel_Section() {
      WHO_AM_I( &Pid )
      start = Num_Tr / P * Pid;
      end = start + Num_Tr / P;
      for (i=start;i<end;i++) {
        Get_Input( &Bid, &Tid, &Aid, &Delta);
        /* Update Teller */
        sprintf(comm,"update Account set Abal=Abal+%d where Aid=%d\n",Delta, Aid);
        PQexec(comm);
             :  :  :  :
      }
}
```

(b) Parallelized script file

Figure 6. A transaction example and its parallelized script file.

To maintain the system in a correct state, accesses to shared data must be synchronized. In EZDB, binary locks are associated to every shared data structure and only one processor can access a data structure at a time. Other policies are possible such as associating a lock with each memory page to reduce the overhead due to busy-waiting.

In the `for`-loop of the parallelized script only one query (`Update Account`) is shown among the four in a transaction. The DBMS receives multiple query statements via *PQexec*() from P processes at a given time. There may be up to P concurrent queries in progress. They are compiled and executed as described in previous section, but the execution of the P queries can proceed concurrently on multiple processors. EZDB does not exploit intra-query parallelism. For concurrent query processing, ANL macros `LOCK` and `UNLOCK` are inserted before and after program statements accessing shared data in the program modules of EZDB.

4 TPC-B

The TPC benchmarks are standard benchmarks widely used in industry to compare the performance of database systems. The simplest among them is TPC-B, in which a series of simple queries are executed on behalf of independent banking transactions shown in Figure 6 (a).

4.1 Database Specification

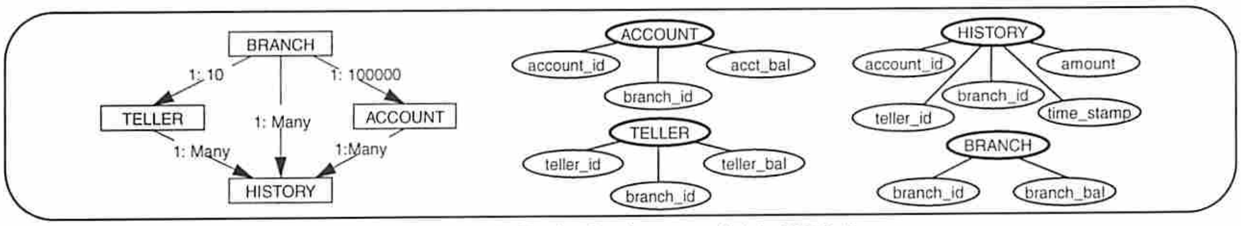


Figure 7. Basic database model in TPC-B.

There are four tables defined in TPC-B: `Teller`, `Branch`, `Account` and `History`. Their logical relationships and schema are shown in Figure 7. A `Teller` record is at least 100 bytes and contains fields `Tid`, `Bid` and `T_Balance`. A `Branch` record is at least 100 bytes and contains fields `Bid` and `B_Balance`. An `Account` record is at least 100 bytes and contains fields `Aid`, `Bid` and `A_Balance`. A `History` record is at least 50 bytes and contains fields `Aid`, `Tid`, `Bid`, `Delta` and `Time_stamp`. One of the restrictions in TPC-B is that the ratio between the numbers of records of types `Account`, `Teller` and `Branch` must be 100000:10:1.

4.2 TPC-B Transaction

In this section, we illustrate how a basic operation is executed on the database data using the metadata and index data. Given `Bid` and `Delta`, the `B_Balance` field of a record in `Branch` is incremented by `Delta`. From the `TableHead`, the number of tables in the database system is found and it is used to define the search space in the `TableTable`. The `table_name` field of each record in the `TableTable` is compared with "`Branch`". The table number, the number of records in `Branch`, the number of fields in a `Branch` record and the primary key, `Bid`, are obtained. The data types of the `Bid` and `Delta` are checked by consulting `RecordHead`. Since the `Bid` is the primary key, index search is performed using the B-Tree for `Branch`. From the B-Tree, the memory location of the `Branch` record whose `Bid` is given is found. The sum of `B_Balance` and `Delta` is computed in the numeric function unit and written into the

`B_Balance` field.

5 Simulation Methodology

Figure 8 illustrates the simulation method. The simulation environment is the CacheMire test-bench [3] which executes parallelized applications to generate instructions and memory references. CacheMire executes the code of EZBD running the script file and generates memory references. Instruction fetches, private and shared data accesses and synchronization operations (test-and-set) are monitored on the fly as the execution progresses. The target architecture model is that of a cache-coherent, single-bus, shared-memory multiprocessors with up to 32 processors. The caches are 4-way set associative with the LRU (least-recently-used) replacement policy.Data coherence is enforced by the Illinois protocol. In our simulation, all the database data are brought initially into the main memory along with the metadata and B-Trees so that we may track the accesses to shared data structures. Therefore, IO activities are not simulated. In addition, the instruction fetches and data accesses of operating system processes are not included.

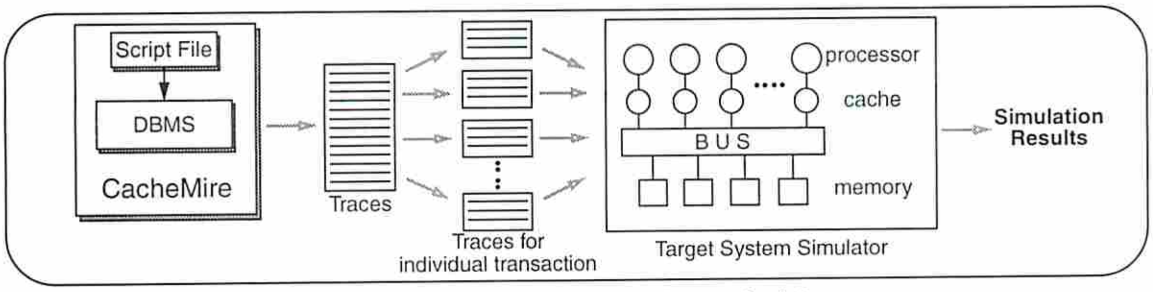


Figure 8. Overview of simulation methodology

In our methodology, a trace is first derived and then the trace is applied to various system configurations. We collect a global trace on a simulator of a 4 processor system which services 200 TPC-B transactions with 4 branches, 40 tellers and 400,000 accounts. The trace is then broken down into 200 small files, one for each transaction. To do this, the parallelized DBMS and script files are augmented with directives at the beginning and at the end of each transaction. These directive insert markers in the global trace.

When the transaction traces are applied to a particular configuration, we first warm up the caches by running the 200 transactions. After this initial run, all metadata structures, B-Trees, and small tables are in cache. The 200 trace files are then applied again and measurements are taken. We simulate configurations of up to 32 processors. Cache size varies from 32 KBytes to 1

MBytes. We have looked at cache block size between 8 and 1024 bytes. The main memory in our simulation is 1 Gbytes with 4 KByte pages.

6 Memory Behavior of TPC-B

6.1 Sizes of Data Structures

The records in all database tables are 196 bytes long and each node in the B-Trees holds in 20 bytes. All shared data are stored in less than 100 MBytes of main memory. Table 2 shows the sizes of all shared data structures. In our simulations, the numbers of entries in `History` does not exceed 1,000. The largest data structures is `Account` with 400,000 data entries.

Data Structures	Metadata			Database Data				Index Data			
	TableHead	TableHead	TableHead	TableHead	TableHead	TableHead	TableHead	TableHead	TableHead	TableHead	TableHead
Size (bytes x items)	4 x 1	26 x 4	220 x 4	196 x 4	196 x 40	196 x 400K	196 x 1K	20 x 4	20 x 40	20x 400K	20 x 1K

Table 2. Shared data structures and their sizes.

6.2 Memory Access Counts

Table 3 shows the global measurements obtained for the average transaction in the traces we collected on EZDB. There are 34,258 instruction fetches per transaction. This number correlates well with previous evaluations of TPC-B. For example, Oracle on a Data General AViiON AV/8500 server generated about 35,000 user instruction fetches per TPC-B transaction [11].

Compile Time			Transaction Processing					
Instruction fetch	Private Data		Instruction fetch	Private Data		Shared Data		
	Read	Write		Read	Write	Read	Write	Lock
195571	18039	5966	34258	2063	291	261	41	51

Table 3. Global access counts in the average TPC-B transaction.

Data	MetaData			Synch.	Record Table				Search Tree (B-Tree)			
	Table-Head	Table-Table	Record Head		Branch	Teller	Account	History	Branch	Teller	Account	History
Read	5795	4589	5594	-	432	542	834	545	1306	3136	10675	9603
Write	-	200	-	-	322	317	317	6459	-	-	-	1194
Lock	-	-	-	10176	-	-	-	-	-	-	-	-

Table 4. Access counts for individual data structures in 200 TPC-B transactions.

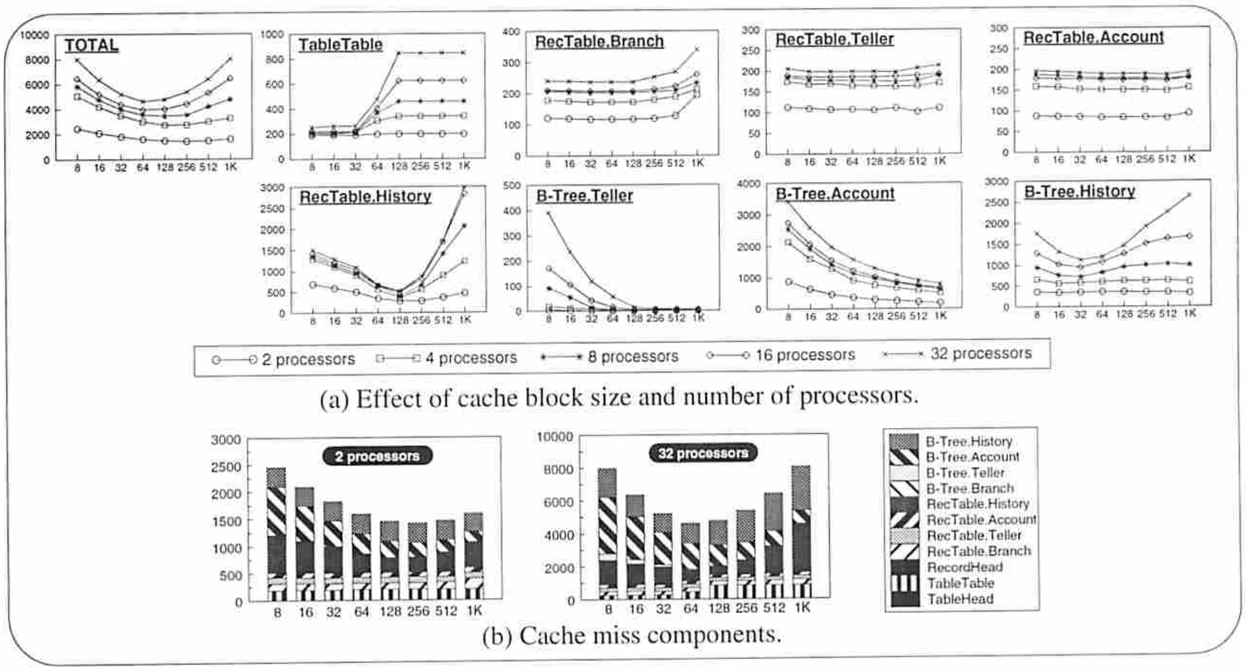
Table 4 shows the total number of data accesses to each metadata, database data and index data for the 200 transactions. Note that by far the most accessed shared data (besides locks) are the B-Trees, `History` and the metadata. Whereas `Account` is by far the largest data structure there are relatively very few accesses to it.

The accesses to metadata structures are mostly reads. In particular, `TableHead` and

RecordHead are read-only in TPC-B. In TableTable one entry is updated once whenever a History record is inserted resulting in a total of 200 writes. One single field (id) of a record in each of Branch, Teller and Account is updated. By contrast, in History, all fields are written by insert operations and the number of writes dominates. Since B-Trees are updated only when a record is inserted or deleted in a table the B-trees in TPC-B are read-only except for the B-tree of History which is read-write.

6.3 Data Sharing (Infinite Caches)

Each graph in Figure 9 shows the number of misses for one data structure in a system with infinite caches as a function of the block size. The five curves in each graph correspond to systems with different number of processors. In the following we comment on the effects of the block size and number of processors. Accesses to TableHead and RecordHead experience no miss (Figure 9 (b)) because they are read-only and the caches are warm at the beginning of the simulation. Thus, we ignore these data structures along with the Branch B-Tree and private data.



(a) Effect of cache block size and number of processors.

(b) Cache miss components.

Figure 9. Effects of block size and number of processors in infinite cache.

6.3.1 Block Size

Looking at the curves for the total miss rates in Figure 9, we observe some gains due to spatial locality for smaller block sizes, but, for larger block sizes, the miss rate increases, a sign that false sharing dominates. False sharing is due to accesses to TableTable and to accesses to History and its B-Tree. Let's look now at individual data structures.

The only interesting metadata is TableTable. Given a query and a database table name, the records in TableTable are searched to find the number of records and the number of fields in a record of the table. During a delete or insert operation the number of records in the table is updated in TableTable. In a TPC-B transaction, there are four queries and TableTable is searched four times per transaction. History is the only table where one record is inserted in a transaction and this causes cache misses on accesses to TableTable in consecutive queries. Because the size of TableTable is 104 bytes the number of false sharing misses grows dramatically for block sizes of 64 and 128 bytes, but remains constant for larger blocks.

Among the tables in the database, Branch and Teller are the smallest ones. One integer variable, B_id or T_id, of one record in Branch or Teller, respectively, is modified once in a transaction. Since the size of each record in these tables is 196 bytes, a block size smaller than 196 bytes does not affect the number of misses and all misses are true sharing misses. Larger blocks will cause false sharing misses. We observe more (false) sharing misses in Branch than in Teller for larger blocks, because there are only four records in Branch. Overall, the number of misses on accesses to Branch and Teller is small. Because there are 400,000 Account records and only one record is accessed (modified) in a transaction, the probability that an access to Account record hits in a cache is almost zero. Thus, most of times, one (cold) miss is counted in a transaction independently of the block size. In the case of History, a new record is created and inserted into the table in each transaction and is never accessed again. On an insertion, a 196 byte record is filled, causing cold misses. Larger blocks reduce the number of cold misses. However, if the block is larger than 256 bytes, false sharing takes over.

During a transaction, multiple nodes in B-Trees are accessed to locate the record in a query. The B-Trees of Branch, Teller and Account are read-only. Each query in TPC-B updates the primary key field of a record of History, and thus its B-Tree is updated. To understand the memory behavior of B-Trees, look at the perfectly balanced B-Tree in Figure 5. Although the number of nodes in the upper portion of the tree (near the root) is small, they are accessed more frequently than the nodes in the lower levels. Furthermore, the nodes in higher levels reside closely to each other in physical memory. Therefore, in a search operation, larger blocks bring more useful nodes in the upper portion of the tree into a cache. For this reason the B-trees of

Branch, Teller and Account have fewer misses for larger blocks. In particular, since the size of the B-Tree of Teller is only 800 Bytes, it fits into a 1024 byte cache block. Accesses to the B-Tree for History, cause false sharing misses. In consequence, the number of misses ends up increasing for large block sizes.

6.3.2 Number of Processors

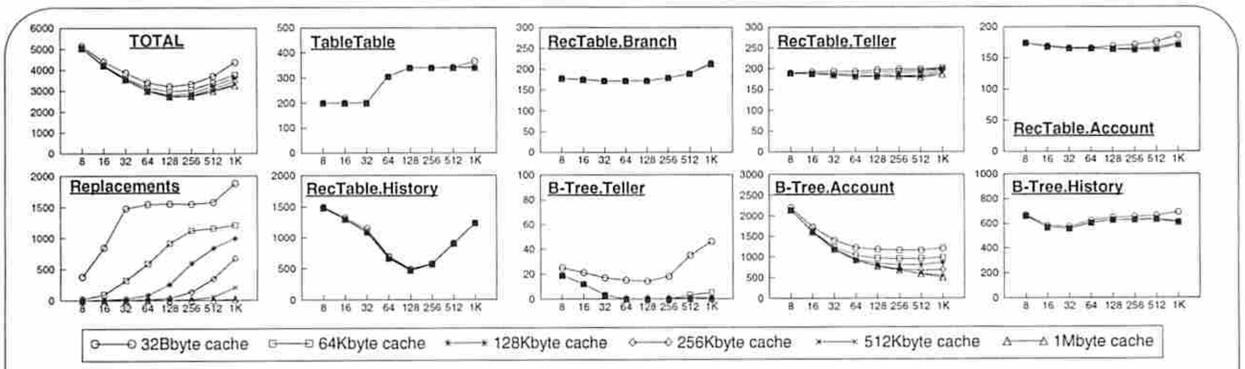
In scientific applications, shared data structures are partitioned and allocated among processors at the initial stage of a program. Careful data partition and allocation can minimize data communication. By contrast, in TPC-B, any processor can access any portion of the shared data. Therefore, most write-shared blocks are migratory, i.e. the ownership of these cache blocks is transferred among processors very often. Hence the number of misses on every data structure in Figure 9 is drastically affected by the number of processors.

The effect of the number of processors is less pronounced in Branch, Teller and Account than in TableTable, History and B-Trees. The reason is that a single integer variable is modified and communicated in the first three data structures. On the other hand, even though one single integer variable is modified in TableTable and the B-Tree of History, the variables are communicated multiple times during a transaction. Finally, in History, a whole 196 byte record is modified causing more misses.

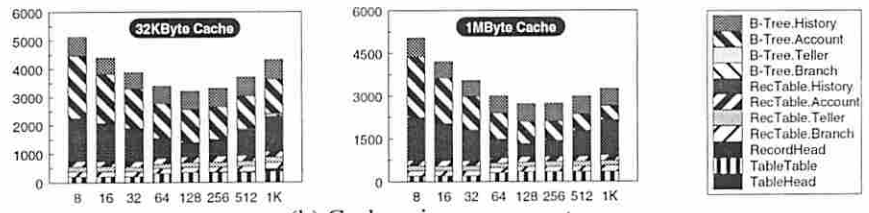
6.4 Finite Cache Effects

We now consider finite cache sizes between 32 KBytes and 1 MBytes. The number of processors is four and the block size is variable. The graphs in Figure 10 show that the block size minimizing the total miss rate of TPC-B is around 128bytes, for all cache sizes. Moreover, the total miss rate is not very sensitive to the cache size, implying that a small data cache of 32Kbytes is sufficient for TPC-B.

To explain the cache size effects observed in Figure 10 on each individual data structure, we group data structures into three classes. The first class includes the metadata plus two database tables, i.e. TableTable, Branch and Teller. These data structures are all small and accessed frequently. Therefore, they are rarely victimized and the cache size has little effect on their miss rate. The second class of data includes Account and History and have little or no temporal



(a) Effect of cache size and of block size.



(b) Cache miss components.

Figure 10. Effects of block size and cache size in finite cache.

locality. Hence their miss rate is not affected by the cache size, even though they are the cause of most replacements. The only data structures whose miss rate is sensitive to the cache size are the B-Trees, especially the B-Tree of `Account`. The temporal locality on accesses to a large B-Tree varies across the tree. At the top the locality is high and the hit rate is high independently of the cache size. The locality decreases for lower levels of the tree. The spatial locality is good and, for a given cache size, we observe a constant decrease of the miss rate with the block size. However, for a given block size, the chances that a block is displaced in between two consecutive accesses is higher when the number of blocks decreases.

6.5 Summary

Table 5 summarizes the memory behavior of all shared data structures in TPC-B. The *spatial locality* is defined as the amount of data in contiguous address range accessed at a time. The *temporal locality* is defined as the average number of the sequences of accesses to a particular memory location per transaction.

The working sets of `TableTable`, `Branch` and `Teller` tables are so small and they presented little sensitivity to the changes of the cache size (Figure 10). Since the `Account` and `History` tables no locality, they also presented negligible differences as the cache size varies. Finally, a few nodes in the upper portion of B-Trees form a working set and their sensitivities to cache size is significant as shown in Figure 10.

		Size (bytes x items)	Accesses		Locality		Sensitivity to		
			Type	Number (%)	Spatial	Temporal	Cache Size (32K to 1M)	Num.Procs (2 to 32)	Block Size (8 to 1K)
Metadata	TableHead	4 x 1	RO	11.52	1 field	4	-	-	-
	TableTable	26 x 4	R/W	9.23	4 records	4	low	high	high
	RecordHead	220 x 4	RO	10.78	4 records	1	-	-	-
Database Data	Branch	196 x 4	R/W	1.45	1 field	1	low	low	low
	Teller	196 x 40	R/W	1.65	1 field	1	low	low	low
	Account	196 x 400K	R/W	2.21	-	1/4K	low	low	low
	History	196 x 1K	R/W	13.50	-	-	low	very high	very high
Index Data	Branch	20 x 4	RO	2.51	few nodes	1/4 to 1	-	-	-
	Teller	20 x 40	RO	6.04	few nodes	1/40 to 1	high	high	high
	Account	20 x 400K	RO	20.58	few nodes	1/400K to 1	high	high	high
	History	20 x 1K	R/W	20.81	few nodes	0 to 1	high	very high	very high

Table 5. Sharing characteristics of shared data.

7 Trace Expansion

Usually, software simulations of database applications consume huge amounts of memory space to store the data structures of the DBMS and benchmarks as well as the codes and arrays of the simulator itself. This is why we could only run a small-scale system in this paper. This is a common shortcoming of such studies. For example, Trancoso et al [14] had to reduce their database by factor of 100. To evaluate architectures for realistic database sizes, we can expand the trace of a reduced database into a trace for a much larger database by taking advantage of the facts that the code is independent of the actual table sizes and that it can be easily annotated.

Recall that we annotated the DBMS and script file with specific directives to identify the beginning and end of a transaction. The directive is actually a program statement which insert a marker in the trace. This is possible only when we understand the DBMS program codes well enough to locate the places where we should put the directives.

Let's assume that there are 10 tables each requiring 1 GBytes in the target database. Furthermore, assume that the size of the virtual memory space on our host workstation is 1.5 GBytes. We only need to simulate accesses to one large table at a time. We first run simulations with scaled-down tables and collect traces as shown in Figure 11. These traces are then fed into a simulator for the realistic database. This simulator runs in two modes: trace-driven and execution-driven simulation. We supply the trace lines to this simulator until we detect a maker (such as the beginning of update of table `i` as in Figure 11). At this point the simulator switches to execution driven simulation to search and update large table `i`. When the update is completed, the simulator

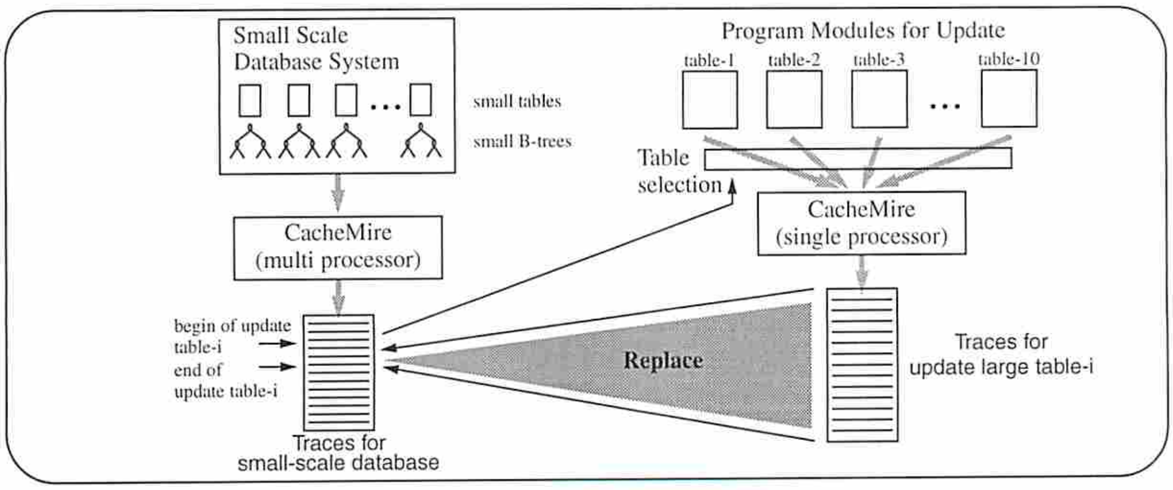


Figure 11. Trace expansion for large-scale database from small-scale trace.

switches back to trace-driven simulation using the traces for the small data base.

There are many possible variations to replace other portion of traces whenever large tables are accessed in a transaction. For example, the DBMS and script file can be annotated to specify the beginnings and ends of index or sequential searches, as well as of insertion, deletion and update of a record in a table.

8 Conclusions

Obtaining reliable measurements for commercial application behavior on existing DBMS is, for many reasons, a very difficult task, which explains why practically all evaluations of computer architectures done today are based on scientific applications. This situation is deplorable since most machines today are designed to run commercial applications.

To address this problem and make progress in the understanding of commercial applications, we have developed our own parallelized DBMS, which we call EZDB. EZDB provides an easy-to-use platform to run and evaluate database programs on multiprocessor architectures.

We have applied the tool to the TPC-B benchmark and have presented the unique characteristics of each individual shared data structure including the localities, working sets and cache misses over various cache sizes, block sizes and numbers of processors. Overall, the cache size does not play a critical role for the database and metadata. On the other hand, some parts of the B-Trees show good utilization of caches. Some spatial locality also exists in the data base because of the large record size. In consequence, the cache block size under TPC-B is suggested to be larger than that used for scientific applications.

We are applying EZDB to other benchmarks, namely TPC-C and -D. EZDB is modular and can be incrementally improved to run more complex applications. In addition, although the current version of our DBMS is working properly, we are improving it. In later versions, we plan to add compiler modules to improve query optimizations as well as code optimization. The index data structures can also be improved by replacing B-Tree with B^+-Trees or B^*-Trees. Besides improving the features offered by EZDB we are also working on improving its efficiency. Much can be done for example, to improve its utilization of memory. This is critical to be able to simulate bigger databases on a given workstation with limited memory. However, since there is a limit to such optimizations, we are also experimenting with our trace expansion approach.

9 References

[1] Cvetanovic, Z and Bhandarkar, D., "Characterization of Alpha AXP Performance Using TP and SPEC Workloads," *Proc. of the 21st Annual International Symposium on Computer Architecture*, pp. 60-70, Apr. 1994.

[2] Boyle, J., et al. "Portable Programs for Parallel Processors". *Holt, Rinehart, and Winston Inc.* 1987.

[3] Brorsson, M., Dahlgren, F., Nilsson, H., and Stenstrom, P., "The CacheMire Test Bench--A Flexible and Effective Approach for Simulation of Multiprocessors," *Proc. of 26th Ann. IEEE International Simulation Symposium*, pp. 41-49 Apr. 1993.

[4] Cormen, T. H., Leiserson, C. E. and Rivest, R. L., "Introduction to Algorithms," *McGrow Hill Book Co.*, 1989.

[5] DeWitt, D. and Gray, J., "Parallel Database Systems: The Future of High Performance Database Systems," *Communications of the ACM*, 35(6):85-98, Jun. 1992.

[6] Elmasri, R. and Navathe, S. B., "Fundamentals of Database Systems," 2nd Edition, *The Benjamin/Cummings Pubishing Co. Inc.*, 1994.

[7] Gray, Jim, "The Benchmark Handbook for Database and Transaction Processing Systems," Second Edition, *Morgan Kaufmann*, 1993.

[8] Marek, Robert, and Rahm, Erhard, "Performance Evaluation of Parallel Transaction Processing in Shared Nothing Database Systems," *Proc. of PARLE*, May 1992, pp. 295-310.

[9] Singh, J. P., Weber, W-D, and Gupta, A., "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, 20(1): 5-44 March 1992.

[10] Stonebraker, M. and Kemnitz, G., "The POSTGRES Next generation Database Management System,", *Communications of the ACM*, June, 1986.

[11] Suggs, D. and Reynolds, R., "Constructing Multiprocessor Workload Characterizations," *Proc. 33rd ACM Annual Southeast Conference*, Clemson, March 1995.

[12] Thakkar, Shreekant and Sweiger, M., "Performance of an OLTP Application on Symmetry Multiprocessor System," *Proc. of the 1990 Int'l. Computer Architecture Symposium*, pp. 228-238.

[13] Torrelas, J., Gupta, A. and Henessey, J., "Characterizing the Caching and Synchronization-Performance of a Multiprocessor Operating System," *Proc. of the 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 162-174, Oct. 1992.

[14] Trancoso, P., Larriba-Pey, J., Zhang, Z. and Torrelas, J., "The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors," *Proc. of 23rd Ann. Int'l Symposium on Computer Architecture*, 1996.