Design and Evaluation
of a High-Speed Extended Relational Database Engine, XRDB

Yasuo Yamane*, Mika Narita**, Fumihiko Kozakura**, and Akifumi Makinouchi**

* Fujitsu Laboratories Ltd.
** Fujitsu Limited

Abstract

New database applications such as artificial intelligence, CAD/CAM, and image processing require the database systems with the following capabilities:

1. to store data types different than the conventional ones
2. to have higher performance
3. to be extensible

To satisfy these requirements, we have developed XRDB (extended relational database engine). XRDB is a simple, extensible, high-speed database engine that supports non-first normal form.

In this paper, we discuss the design principles, the techniques used to increase speed, and the system architecture. The performance evaluation of the single-user version using the Wisconsin benchmark test is shown. A Japanese dictionary system based on XRDB is compared with the specialized routine-based dictionary in terms of access time. These results are also discussed.

Keywords: Non-first normal form, relational database, diverse data, high performance, extensibility, hash

1. Introduction

Relational database systems have been commonly used for business applications and have gotten a good reputation for being easy to understand and use. However, new database application areas such as knowledge-based systems, CAD/CAM, and OIS pose new requirements to the database system on which they are built. Some of those are as follows:

1. Diverse data types. Multimedia data such as text, figures, images, and sounds are required for the new applications. Complex structured data that can be represented by frames [Wins77] is necessary to handle so-called complex objects [Hans82], too.

2. High performance. The new data types mentioned above pose performance problems: The size of data to be stored, retrieved, and processed is much larger than the size of the traditional data types. Retrieving a complex object needs many join operations since the object is composed of smaller objects that are linked to each other. Higher performance is thus required.

3. Extensibility. In current database systems, basic arithmetic and logical operations are supported. However, new data types for complex applications require new operations. Different applications require different optimization strategies suitable to their application-specific queries. It is extremely difficult to satisfy all these requirements, so it is important to make database systems easy to customize. See Exodus [Care86] and Postgres [Ston86] for examples of such research.

To satisfy these three requirements, we have developed XRDB (extended relational database engine). XRDB is a software system that executes basic relational operations with very high performance. Since it is an engine, it does not have a query language or optimizer. XRDB serves as the nucleus for a relational database system. The system built on it uses the operations provided by XRDB.

Although XRDB was designed as a general RDB engine, we had in mind two particular systems that would first be built using XRDB. One was an SQL processor, and the other was a multimedia knowledge-base system which is now under development. The system, called JASMINE [Maki87, Ishi88], supports not only traditional data types such as arithmetic and character strings but also new data types such as graphics, images, rules, and programs. The data structures that JASMINE handles are frames. A frame is stored into a tuple. Since a frame may have multiple values, XRDB must support non-first normal form (NF2) [Maki77]. This is the XRDB's main extension to the pure relational data model. The third consideration in designing XRDB was concerned with the environment in which XRDB runs. We thought then and still believe that the future direction of database server architecture is toward multiprocessors. Our initial experiment [Yama87] with a multiprocessor system and the following experiment with the modified system showed it. XRDB is designed as compact as possible so that it can run on each processor of a multiprocessor system. The algorithms implemented in XRDB are also adaptable to the parallel environment.

In Chapter 2, we discuss the design principles of XRDB. In Chapter 3, we discuss the techniques used to
increase speed. In Chapter 4, we explain the system architecture. In Chapter 5, the performance evaluation using Wisconsin benchmark test and a comparison of an XRDB-based Japanese-language dictionary with a specialized-routine-based one are shown.

2. Design principles

2.1 Simplicity

Simplicity is the most important principle to pursue in designing a reliable, easy-to-make, easy-to-maintain software system. This principle allows the system to be compact, too. To make XRDB simple and compact, we had to carefully choose an operation set to support. Essential relational operations such as selection, join, set union, and so on are included in the set. Some facilities that would be useful in some cases, but that would consume more computer resources are excluded. Nil compaction is an example. Nil compaction may reduce the disk space for storing some relations but restoring the compacted tuples requires additional CPU time. Implementing this facility would also increase the complexity as well as the size of XRDB programs. XRDB was implemented in the C language. The source code is 15K steps. This is relatively small. XRDB can thus run in a small memory space. Since it consumes less CPU time, it can run on less powerful CPUs. Using the C language makes the system portable.

2.2 Primitiveness

Primitiveness and simplicity are two sides of a same coin. To make a system simple, it is important to carefully select the primitive operations and primitive data structures that should be supported. While XRDB is primitive, it must be general enough so that it can be the basis of various kinds of relational databases. Here, "the various kinds of relational databases" means relational databases that support different kinds of strategy concerning index maintenance, optimization, and so on. In other words, XRDB's interface is similar to machine languages that are used as the target languages by high-level application-oriented programming languages such as COBOL, Fortran, and C. Below is a list of some examples which illustrate the primitiveness of XRDB.

(1) Executor of the relational primitive operations. A conventional relational database system generally consists of three parts: the query language parser, the optimizer, and the executor. The parser and optimizer are application dependent as high-level programming languages are. They were thus excluded from XRDB. XRDB is an executor of relational primitive operations only.

(2) Only relations. In conventional relational database systems, dictionaries for metadata and indexes are usually not implemented as relations because of access speed. They are implemented as data structures with operations specific to them. This prevents users from treating them as relations, thus requiring extra learning for the users to understand the special way to access them. All data is stored and processed as relations in XRDB. Realizing dictionaries by relations enables users to add application-specific information to system-specific information. Access speed is sacrificed, however. In XRDB this problem was solved by supporting three types of relations: sequential, B-tree, and hash (See Chapter 4 for details). A dictionary and index may be implemented using B-tree or hash type relations that assure high-speed access. Any relation in XRDB permits to embed relations. This is convenient for implementing a dictionary and index, since both require multiple values. Since they can be implemented as relations, XRDB itself does not need to include any operations for them. XRDB users may implement them with their favorite strategy. For example, some indexes may be maintained with delayed-update strategy and others may be maintained with immediate strategy.

```
begin transaction processing;
open relation EMP;
sum : sum + age;
access each tuple of EMP;
while not end do begin
  if age >= 50 then begin
    sum = sum - age;
    count = count + 1;
  end;  
access next tuple of EMP;
end;
average : sum / count;  
output the value of average;
close relation EMP;
end transaction processing;
```

(a) Expert program for application A

```
begin transaction processing;
open relation DEPT;
access first tuple of DEPT;
while not end do begin
  if noemp >= 50 then begin
    insert deptname into T;
  end;
access next tuple of DEPT;
end;
close relation DEPT;
end transaction processing;
```

(b) Expert program for application B

---

Fig. 2.1 Outline of expert programs of applications A and B

- 53 -
(3) Data transparency. XRDB only supports fixed-length and variable-length data as data types of fields. This feature is similar to the storage object, the uninterpreted variable-length record in EXODUS (Caret86), although ours are uninterpreted fields. Data is transparent in the sense that the values of fields are not interpreted by XRDB except for tuple identifiers (TIDs) and inner relations.

2.3 Separation of application-specific and application-independent processing

A close look at application programs accessing the relational database reveals they are generally composed of two types of parts. One type is application-specific and the other is application-independent.

For example, consider an application called A which computes the average salary of 30-year-old employees. Here let's assume that the personnel records are stored in a sequential relation EMP. We can think of an ideal program for the application A called exp(A) meaning the expert program for A. The outline of exp(A) is shown in Fig. 2.1 (a). Consider another application B which retrieves the departments which have at least 50 people working in them. DEPT is the relation that contains information about departments. This relation is also assumed to be sequential. The outline of the application program, exp(B), is shown in Fig. 2.1(b).

The similarity is the structure of the two application programs. It comes from the similarity of the database structures that the two programs access. Both of them search sequential relations. In general, the structure of an application program depends on the structure of the database. Underlines in Fig. 2.1 show different parts of exp(A) and exp(B). They are concerning the conditions that must be satisfied by tuples to be retrieved or the tuple-wise processing to be performed on the retrieved tuples.

A mechanism to execute application-independent parts of application programs is integrated into XRDB. This is possible since the application-independent parts are database-structure dependent and are clearly definable. Application-specific parts are treated differently. Since they are application-specific, they may vary for different applications. It is very difficult or almost impossible to decide which ones are to be included and which should be excluded. The strategy to make XRDB as general as possible is to separate application-specific parts of an application program, implement them as macros or functions, and then embed them into the application-independent parts. See Fig. 2.2 for an illustration. For implementation efficiency, the application-specific parts are implemented as user-defined functions which are called by XRDB.

The user-defined functions are listed below. For example, they can be used for SQL-type queries.
- (a) Predicate function defining a retrieval condition for selection or join.
- (b) Manipulation function for processing each tuple matching a specified predicate in selection or join.

Summation and insertion in Fig. 2.1 are examples.
- (c) Order function for sorting or B-tree relations.
- (d) The range function defining value ranges for a B-tree search. The typical forms are a <= x <= b, or a < x.
- (e) Static hash function used in hash-based relational operations (see 3.1).
- (f) Dynamic hash function used in hash relations (see 4.2).

Separating application-specific and application-independent parts has the following advantages:

1. Increasing processing-speed. By compiling user-defined functions that are called by XRDB, the total processing-speed of an application program can be increased.
2. Modularity and extensibility. Since application-dependent functions are separated from XRDB, users are free to add any functions needed by their application. This also enables users to group functions by application and to manage each group separately. In this way, even if the whole system built on XRDB becomes complex, the system may keep its modularity and extensibility.
2.4 Diverse data storage

The XRDB data types are as follows:
- field: Fixed- or variable-length. The maximum length is about 64K bytes.
- tuple: Its structure is the same for all four types of relations shown below.
- relation: sequential, B-tree, hash, and inner relations

To accommodate multimedia data and NF2, the page size ranges from 4K bytes to 64K bytes. The tuple length must be less than the page size so that any tuple can be contained in a page; this is for efficiency and implementation simplicity. This is the only restriction on field length or the number of fields. It is thus possible to store long data close to 64K bytes in a field. We think comparatively small data are stored in tuples, and long data which cannot be stored in a tuple are stored in relations, or the pages supported by the storage layer (see 4.1). The advantages of storing long data in relations or pages are as follows:

(a) Recoverability. If recovery is not required, it is possible to store them as files, with only the file names in XRDB.
(b) Simplicity of system architecture.
(c) Functionality of relations. Using B-tree or hash relations, it is possible to directly access and alter parts of long data.

Since XRDB does not interpret the values of fields except TIDs and inner relations, any type of data can be defined in user-defined functions. Thus, any type of data can be stored in XRDB. A field can be regarded even as a structured data such as an array, or a record. Actually, XRDB inner relations are implemented as a variable field.

In XRDB, field values can be relations, using inner relations. That is, NF2 can be implemented. Inner relations can have inner relations as its field values, so a recursive structure is possible. Thus, frames or other nested data can be stored in natural way.

TID can be stored as a fixed-length field. TID acts as a pointer, so we can construct a variety of structured data in the database. For example, indexes whose tuples consist of single or multiple keys and TID (Fig. 2.3 (a)), and pre-joined relations (Fig. 2.3(b)) can be constructed.

Thus, XRDB can store many kinds of data. However, the only essential extension to the relational model is supporting NF2. This minimizes additional complexity of the system while keeping the model XRDB supports simple.

3. Techniques for high performance

3.1 Hash-based system

Many relational database systems are sort-based [Blas77, Maki81]. In these systems, sorting is used to implement relational operations such as equi-join, set-difference, duplicate elimination. Hash-based methods have also been discussed [Brat84, Yama85]. In equi-join, sort-based methods require O(m log m + n log n) processing time, where m and n are the number of tuples of the two joined relations. If we use hash join, when either of two joined relations can be loaded into main memory, the processing time is O(m + n). Considering that the cost of main memory is becoming cheaper and that larger amounts of main memory are becoming available, it will be easier to load a smaller relation of two joined relations into main memory and most join operations will be able to be processed quickly. Even if neither relation can be loaded into main memory, as the experiments in [Yama85] showed, hash join generally requires less CPU time and I/O times than the sort-based method. For these reasons, we adopted the hash join method for equi-join. We also decided to apply hashing to the other operations listed above. Thus, XRDB is a hash-based system.

3.2 Simple processing and data structure

In the previous section, we discussed the technique used to reduce the order of time complexity. Here, we discuss the technique used to reduce the coefficient part of the time complexity.

The main principle is to move as little data as possible and to handle tuples in buffers if possible.
Internal sorting, a pointer array is constructed for sorted tuples (Fig. 3.1), and the pointers are moved instead of tuples. To make a hash table, tuples are not moved. Instead, pointers are linked. In database processing, field is a unit of reference. To reference fields in XRDB, a field pointer array is used (Fig. 3.2). Each pointer points to the corresponding field in a tuple in a buffer. The alternative is to copy the field value to another area. This is good for data protection, but our experience in RDB/V1[Maki81] showed it to be slow. When field values are passed to user-defined functions, a field pointer array is used.

In XRDB, the tuple structure is very simple (Fig. 3.3). The first two bytes include its length. The tuple is composed of 2 parts, a fixed part and a variable part. The fixed-length fields are put in the fixed part, and variable-length fields are put in the variable part. Only the offset of each variable length field is put in the fixed part. This way allows any variable-length field position to be computed in constant time. This structure has one drawback. It does not allow nil-value compaction; we consider that time is more important than space.

4. System architecture

4.1 System configuration

XRDB is composed of relational operation, tuple, and storage layers. The interface of each layer is open to users. The system configuration is shown in Fig. 4.1. XRDB runs under Unix.

4.1.1 Relational operation layer

This layer supports 15 functions for relational operations. Fig. 4.2 shows some representative examples. For example, 'select' is an extended operation of selection in relational algebra. It has three parameters 'rb', 'pb', 'mb'. 'rb' is the data block which specifies the source relation. 'pb' specifies the user-defined function representing a filtering predicate. 'mb' specifies the user-defined function applied to every tuple which satisfies the predicate. 'tjoin' is similar to join. The relation specified by 'rb1' has a TID field, and the tuple pointed to by TID is joined. It is typically used for index processing. 'nest' and 'unnest' are used for conversion between the first normal form and NF relations.

4.1.2 Tuple layer

Tuple layer manipulate tuples of sequential, B-tree, hash, and inner relations. The operations of this layer are as follows:

- scan: Scans a relation sequentially and finds a tuple satisfying a given predicate.
- raster: Scans a relation sequentially fixing scanned pages on buffers. This operation is used in internal sorting or making internal hash tables.
- access: Directly accesses a tuple matching the predicate.
- fetch: Directly accesses the tuple specified by a given TID.
- insert: Inserts a tuple or a group of fields.
- delete: Deletes the tuple specified by a given TID.
- update: Updates the tuple specified by a given TID.
- clear: Deletes all tuples.

Unix is a trademark of AT&T Bell Laboratories.
flac: Constructs a field pointer array for the specified fields

This layer conceals the physical structure of tuples or relations. This layer's interface of functions is similar for all four types of relations. For example, the similarity of scan functions for 4 kinds of relations is shown in Fig. 4.3. The main processing of the relational operation layer can be programmed independently of the type of relation as if only one type of relation is supported. This layer is not affected much when a new type of relation is added as long as the interface similarity is maintained. This makes it possible to increase the number of relation types supported and not decrease performance. This is important for system extensibility.

4.1.3 Storage layer

The storage layer manages I/O, buffers, and transactions. A database is composed of two types of subdatabases. One is called a base subdatabase which is permanent and recoverable. The other is called a work subdatabase which is used as work space for keeping temporary relations, and is only effective during a transaction. This is not recoverable. Subdatabases are composed of a fixed number of 4K-byte-length physical pages.

The storage layer supports variable-length pages. A variable-length page consists of several physical 4K-byte pages, which form a virtually continuous page on buffers. It can be 4K bytes, 8K bytes, 16K bytes, 32K bytes, or 64K bytes. We used the buddy system [Knut68] for buffer space allocation, so the page length is the power of 2 times 4K bytes. The reasons the page length is variable are as follows:

(a) In XRDB, a long tuple must be completely contained in a page.
(b) Multimedia data may exceed a 4K-byte page.
(c) An inner relation may exceed a 4K-byte page.

In XRDB, the user can specify the page length for each relation.

4.2 Relations

XRDB supports four kinds of relations as explained below. They have the same tuple structure, although they are structured differently.

(1) Sequential relation. The pages of this relation are linked sequentially. Tuples are stored in the order

select(rb, pb, mb)
Searches (rb) for the tuples which match (pb). Performs (mb) on each tuple matching (pb).

hjoin(rb1, rb2, mb, hb)
Performs equi-join of (rb1) and (rb2). (mb) is performed on each pair of tuples which match on join fields. This operation is performed using hash join and a hash function is specified by hb.

join(rb1, rb2, mb)
Performs general join of (rb1) and (rb2). (mb) is performed on each pair of tuples which match (pb) on join fields.

tjoin(rb1, rb2, tid, mb)
The field specified by tid of (rb1) is a field storing TID. Each tuple in (rb1) is joined with the tuple pointed by its TID, and (mb) is performed on such pair of tuples.

sort(rb1, rb2, ob)
Sorts (rb1) and stores the result into (rb2), in the order specified by ob.

unique(rb1, rb2, hb)
Eliminates duplicates of (rb1) and stores the result in (rb2). This operation is hash-based.

nest(rb1, rb2, gb, hb)
Generates NF² relation as (rb2) from the FNF relation (rb1). This operation is hash-based.

unnest(rb1, rb2, gb)
Generates FNF relation as (rb2) from the NF² relation (rb1).

Fig. 4.2 Representative functions in relational operation layer

rb, rb1, rb2 is the data block which specifies a sequential /B-tree/hash/inner relation (the types of allowed relations are dependent on each operation). (rb) means the relation specified by rb.

pb is the data block which specifies the user-defined function which specifies the predicate. (pb) means the predicate specified by the user-defined predicate function.

mb is the data block which specifies the user-defined function which specifies the processing performed on each tuple which matches the predicate. (mb) means the processing specified by the user-defined manipulation function.

ob is the data block which specifies a user-defined function which specifies the linear order. (ob) means the order specified by the user-defined order function.

hb is the data block which specifies a user-defined hash function.
in which they are inserted. Tuple positions are fixed, so indexes can be constructed.

(2) B-tree relation. This has the B-tree structure. It stores tuples in its leaf pages in the order defined by the user-defined order function.

(3) Hash relation. This is based on a dynamic hashing scheme, which is called linear hashing with partial expansions [Lars80]. The reasons we selected this scheme are as follows.
   a) Space required to store data is proportional to the amount of the data.
   b) Space utilization ratio is adjustable and is high.

The space utilization ratio can be controlled, although the efficiency for deletion and insertion decreases when the space utilization ratio increases. We can change this scheme into a static hashing scheme with a little modification. So we can also use this relation as the relation based on a static hashing scheme. We actually implemented it this way.

(4) Inner relation. This relation is introduced so that XRDB supports the NF* model. In XRDB, an inner relation is placed in a variable field. Tuples of the inner relation are placed sequentially in the variable field in an outer tuple. An inner relation can also have other inner relations as its fields, so it can be NF* recursively.

4.3 User-defined functions

Application-specific parts generally have three parts: preprocessing, main processing, and postprocessing. For example, in the computation of average salary (Fig. 2.1(a)), (A) is preprocessing, (B) is main processing, and (C) is postprocessing.

To support this, the 6 types of user-defined functions have the interface shown in Fig. 4.4. XRDB calls the user-defined function with PRE as the value of part in preprocessing, MAIN in main processing and POST in postprocessing. XRDB also passes other parameters to user-defined functions; for example, field pointer arrays, and the pointer to static area where values are kept.

Here, we show one example to explain how to use XRDB. The processing shown in Fig. 4.5 is almost the same as that in Fig. 2.1(b). In Fig. 4.5, however, the result tuples are sorted on the field deptname and stored in the relation RESULT.

5. Performance evaluation

We evaluated performance of XRDB by using the Wisconsin benchmark and compared a Japanese dictionary system based on XRDB with one based on a specialized file access routine.

5.1 Wisconsin benchmark

For the evaluation, we used a Toshiba AS3260 workstation (SUN3/260 OEM, MC68020 (25 MHz) CPU). The disk used was a Fujitsu M2333 (average positioning time 20 ns and data transfer rate 2.4M bytes/s). The tuple length was 182 bytes. We

sequence of operations:

select(rbl,pb,mb);
sort(rb2,rb3,ob);
user-defined functions:
predicate(part, noemp, ...)
   begin
      if part = MAIN then begin
         if noemp >= 50 then return TRUE;
         else return FALSE;
      end
   end
manipulate(part, noemp, ...)
   begin
      if part = PRE then open relation T
      else if part = MAIN then insert deptname into T
      else close relation T
   end
order(part,deptname1,deptname2)
   begin
      if part = MAIN then begin
         if deptname1 > deptname2 then return LARGER;
         else if deptname1 < deptname2 then return SMALLER;
         else return EQUAL;
      end
   end

Fig. 4.5 XRDB use

rbl is the data block which specifies the relation SECT.
pb is the data block which specifies the user-defined function 'predicate' which specifies the predicate noemp>=50.
mb is the data block which specifies the user-defined function 'manipulate' which specifies the insertion processing.
rb2 is the data block which specifies the relation T.
rb3 is the data block which specifies the relation RESULT.
ob is the data block which specifies the user-defined function 'order' which specifies the linear order of the field deptname.
measured the following commands:

(a) Selection without indexes. Retrieve 1,000 tuples from a relation, A, of 10,000 tuples, without using indexes, and make a new relation, T, containing 1,000 retrieved tuples. We used a sequential base relation as relation A, and a sequential work relation as relation T (the base relation is contained in a base sub-database, and the work relation in a work sub-database).

(b) Selection with indexes. Retrieve 1,000 tuples from a relation, A, of 10,000 tuples, using indexes, and make a new relation, T, containing 1,000 retrieved tuples. We used a sequential base relation as relation A, and a B-tree relation as the relation A index, and a sequential work relation as relation T.

(c) Join. Retrieve 1,000 tuples from a relation, B, which has 10,000 tuples, and make a new relation, B', and join B' with another relation, A, which has 10,000 tuples, and make a new relation, T, which has 1,000 retrieved tuples. We used base sequential relations for A and B, and a work sequential relation for B' and T.

We made measurements of two cases; one case was that the tuples were initially on disk and the disk was used to store the retrieved tuples (secondary memory measurement), and the second was that the tuples were initially in buffers and disks were not used (main memory measurement). For the secondary memory measurement, we used 256 4K-byte buffers, and 1536 buffers for the main memory measurement. About half the retrieved tuples remained on buffers and were not stored into the secondary memory in our secondary memory measurements. The results are listed in Table 5.1. By comparing main and secondary memory measurements, we can see that most of the processing time of XRDB is used for I/O processing. The rates are 91.4% in selection without indexes, 97.6% in selection with indexes, and 88.0% in join.

As listed in Table 5.1, it took 10.63 seconds for XRDB to execute join. It took 26.51 seconds for XRDB to sort 10,000 tuples. This means that it would take more than 28.51 seconds to execute join if using the sort-based method. The data distribution of Wisconsin benchmark data is uniform, so it is the most suitable for the hash join method. We used 256 buffers, so the smaller relation (1,000 tuples) of two relations to be joined could be loaded entirely into buffers. This is why the command could be executed so quickly. Although the conclusion can not be generalized because of the favorable environment for the hash join, we believe that the hash join method is more efficient than the sort-based one.

The performance of commercial-INGRES[Bit68], IDM500 with database accelerator (dac) [Bit68], GAMMA[DeWi87], and DBC/1012[DeWi87] systems are also shown. The length of tuples used in [DeWi87] was 208 bytes. If we take into account the fact that Gamma (8 processors with disks) and DBC/1012 (20 processors with disks) process in parallel, the XRDB performance is very good.

5.2 Comparison with a specialized routine

Database systems generally have inferior performance compared to systems customized for a specific application. To show the real performance of XRDB, we compared it with a routine specialized to access Japanese-language dictionaries.

The dictionary system with a specialized routine had about 115,800 Japanese words (records) in its specially formatted file with a special index and the total size was 20.5M bytes. The average record length was about 80 bytes. We made the same dictionary using a sequential relation in XRDB, and an index with a B-tree relation. It took up 23M bytes. We evaluated them on a Sony NEWS workstation (CPU MC68020 (16.67 MHz)). 80 buffers were used for XRDB.

The application we used for the evaluation was the segmentation of Japanese sentences. Segmentation means breaking down a sentence into words. The three Japanese sentences shown in Fig. 5.1 were segmented. Table 5.2 lists the performance of both systems. The specialized dictionary routine loads the special index into main memory before processing for accelerating dictionary accesses, but the B-tree relation of XRDB was loaded on demand. The measured time is only the

---

<table>
<thead>
<tr>
<th>System</th>
<th>Selection without indexes</th>
<th>Selection with indexes</th>
<th>JoinAselB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(s)</td>
<td>(s)</td>
<td>(s)</td>
</tr>
<tr>
<td>XRDB</td>
<td>4.67</td>
<td>11.81</td>
<td>10.83</td>
</tr>
<tr>
<td>(sec)</td>
<td>0.40</td>
<td>0.28</td>
<td>1.20</td>
</tr>
<tr>
<td>GAMMA</td>
<td>2.11</td>
<td>2.16</td>
<td>5.1</td>
</tr>
<tr>
<td>DBC/1012</td>
<td>15.97</td>
<td>16.82</td>
<td>35.6</td>
</tr>
<tr>
<td>(dac)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBC/1012</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(dac)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 5.1 Wisconsin benchmark results*

*smm: Secondary memory measurement

*mm: Main memory measurement*

---

(1) 今日は風邪です。
(2) 価格は高いが、品質はいいので、先払いに決めた。
(3) これまで日本の広告の中では、誰社の製品やブランドをそのまま使って広告するということはあまり見られなかった。

*Fig. 5.1 Japanese-language sentences used for measurement*
time spent for accessing the words in the dictionary. The time for segmenting the sentence, preprocessing, and postprocessing are not considered. This shows that the XRDB is comparable to a specialized routine.

6. Summary

We have developed XRDB to satisfy diverse requirements that would be expected from different kinds of applications. These requirements are flexible data storage, high performance, and extensibility.

Users can store any type of data they want using the uninterpreted data type supported by XRDB. Multimedia data can be stored using fields or relations and complex structure can be made using inner relations, TID, and relations themselves that are based on diverse access methods (sequential, B-tree, dynamic hash and static hash). Note that these facilities were provided with minimal change to the relational model.

For high performance, we constructed XRDB as a hash-based system. We simplified the algorithms and data structures as much as possible. XRDB proved to have comparable performance of some parallel database machines and a specialized routine for accessing a Japanese-language dictionary. The results show also that about 90% of the processing time is spent for disk accesses, so a main memory database would greatly improve performance, about ten times to be exact.

To get extensibility, we separated application-specific and application-independent parts. All application-specific parts are passed to XRDB as user-defined functions. This separation allows users to define customized data types and operations on them. This also leads to high performance.

XRDB is now a single-user version. The maximum page size is 64K bytes, so the tuple size is limited to about 64K bytes. In the near future, we plan to develop a multi-user version with concurrency control and recovery. The maximum page size will be 2G bytes to store a larger amount of data in a field or tuple.

Acknowledgments

This work was a part of the Large-Scale Project of AIST/MITI, “Interoperable Database System.” We are indebted to Mr. Adachi, who was a research developer of the relational database system RDB/V1, for his helpful advice. We would like to thank Tsuzaki, Izumida, Nakata, Ishikawa, Yoon, and Take for their helpful discussion.

References