ISBN 978-89-6211-606-9 98560

비동기식 가시화 트렌젝션 처리를 위한 공유 메모리 데이터 베이스 설계 및 구현

(Shared memory database for asynchronous visualization transaction)

김 민 아 (petimina@kisti.re.kr)

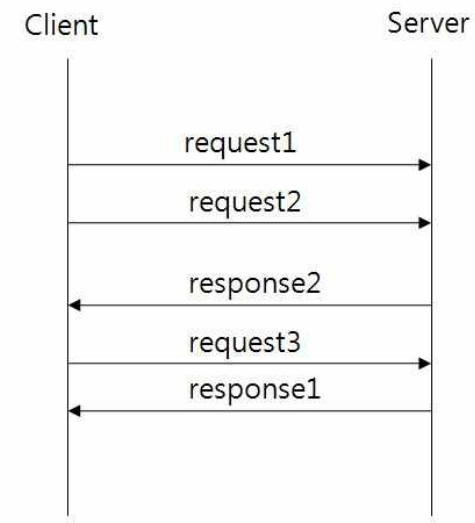
한 국 과 학 기 술 정 보 연 구 원
Korea Institute of Science & Technology Information

목차

그림 차례

1. 개 요

네트워크 프로토콜에는 동기식 프로토콜과 비동기식 프로토콜이 있다. 동기식 프로토콜이란 클라이언트가 서버에 요청을 하고, 그 응답이 올 때까지 기다리는 것이다. 동기식 프로토콜의 대표적인 프로토콜은 HTTP가 있다. 비동기식 프로토콜은 클라이언트가 서버에 요청을 한 후 응답을 기다리지 않고 즉시 다음 요청을 처리하는 것이다. 보통 전체 요청과 응답의 throughput을 높이기 위해 사용한다[1]. 동기식 프로토콜은 transaction의 관리가 매우 간단하다. 하나의 요청을 처리하기 위해, 응답이 올 때까지 기다리므로, 유지해야 할 정보는 세션이 있다면 세션 정보 정도이다. 그러나, 비동기식 프로토콜은 응답이 오기 전에 바로 다음 요청을 처리하므로 실제로 그 먼저 보낸 요청의 응답이 왔을 경우, 어떤 요청에 대한 응답이 왔는지 판단하기 위한 정보를 유지하여야 한다. 그림 1은 비동기식 프로토콜의 예를 보여 준다. request1을 서버에 전송하고, 그에 대한 응답이 오기 전에 request2를 다시 보낸다. 요청에 대한 응답은 request1을 먼저 보냈지만, request2에 대한 응답인 response2가 먼저 올 수도 있다. 클라이언트는 서버로부터 응답이 왔을 때 그 response 가 request1에 대한 응답 response1인지, request2에 대한 것인지 판단하여, 그에 해당하는 처리를 수행하여야 한다.



[그림 1] Asynchronous protocol sequence example

이를 위해 클라이언트는 DB를 사용한다. 즉, request1을 DB에 저장해 두었다가, response1이 오면 request1에 대한 key 로 DB를 찾아, request1에 대한 응답임을 판단한 다음 응답에 대한 처리를 한다. 그러나, 실시간 응답을 필요로 하는

어플리케이션의 경우, 하드 디스크를 사용하는 데이터베이스는 매우 비효율적이다. 이 때문에 많은 상용 메모리 데이터베이스들이 존재한다. 그러나 이들은 단지 비동기식 프로토콜을 위해 사용하기에는 너무 무겁다.

본 문서에서는 이들 상용 메모리 데이터베이스를 대신할 가볍고 매우 간단한 기능만을 가진 shared memory 데이터베이스의 기능 요구 사항을 분석하여 설계하고 구현한다.

2. 상용 메모리 데이터베이스

메모리 데이터베이스는 IMDB(In-Memory Database) 라 불리기도 하고, MMDB (Main Memory Database) 라 불리기도 한다. 하나의 데이터베이스 관리 시스템으로 컴퓨터의 데이터 저장 공간을 디스크가 아닌 메인 메모리에 의존한다는 뜻에서 기존의 데이터베이스 시스템과 다르다.

[표 1] Memory Database Products

Product name	License	Description
Adaptive Server Enterprise (ASE) 15.5	Proprietary	enterprise database from Sybase[4]
Apache Derby	Apache License 2.0	
Altibase	Proprietary	has in-memory and disk table; HYBRID DBMS
BlackRay	GNU General Public Licence (GPLv2) and BSD License	
CSQL	GNU General Public Licence or proprietary	
Datablitz	Proprietary	DBMS
H2	Mozilla Public License or Eclipse Public License	has a memory-only mode
HSQLDB	BSD license	has a memory-only mode
InfoZoom	Proprietary	in-memory BI and data analysis
membase	Apache License	NoSQL, hybrid
MicroStrategy		in-memory BI for MicroStrategy 9
MonetDB	MonetDB License	
MySQL	GNU General Public License or proprietary	has a cluster server which uses a main-memory storage engine
Oracle Berkeley DB	Sleepycat License	can be configured to run in memory only
Panorama		for Windows and Macintosh, both single user and server versions
Polyhedra IMDB	Proprietary	relational, supports High-Availability; acquired in 2001 by ENEA
QlikView		BI-tool developed by QlikTech
RDM Embedded	Proprietary	including hybrid
RDM Server	Proprietary	including hybrid
Redis	BSD	NoSQL
solidDB by IBM		including hybrid, HSB-based HA, Shared memory, embedded, XA, etc.
SQLite	Public domain	hybrid, RAM and disk dbs can be used together
Starcounter		in-memory object relational dbms
TimesTen by Oracle		
VoltDB	GNU General Public License v3	in-memory
TREX		search engine in the SAP NetWeaver integrated technology platform produced by SAP AG
Xcelerix by Frontex		commercial product

MMDB는 하드 디스크에 최적화된 데이터베이스 보다 훨씬 더 빠르고 더 간단하며 더 적은 CPU 명령으로 실행가능하다. 메모리 데이터베이스는 저장 공간의 특성상 , 컴퓨터 시스템의 파워가 켜져 있는 순간에는 데이터를 유지할 수 있지만, 그렇지 않을 경우 데이터를 잃어버리게 된다. 이 때문에, 메모리 데이터베이스들

은 ACID(atomicity, consistency, isolation, durability)의 데이터베이스가 가져야 할 기능 중 durability에 있어서는 취약점을 가진다. 이러한 취약점을 보완하기 위해 대부분의 상용데이터베이스들은 부가적으로 디스크에 데이터들을 백업한다.

현재까지 상품으로 나와 있는 메모리 데이터베이스의 리스트는 표 1과 같다.

그러나, 이러한 시스템들은 매우 무겁고 설치에 시간적, 경제적 비용이 든다. 본 문서에서는 이들 상용 메모리 데이터베이스의 전 기능이 아니라 매우 기본적인 기능을 지원함으로써, 비동기식 프로토콜이나 프로세스 간 데이터 공유가 필요한 응용 프로그램이 간단히 사용할 수 있는 메모리 데이터베이스를 설계하고 구현한다.

3. 비동기식 프로토콜을 처리하는 메모리 데이터베이스 요구사항

가. Simplicity

비동기식 프로토콜을 처리하기 위해 필요한 데이터베이스의 기능은 transaction을 저장하고, 검색하는 매우 간단한 기능이다. 또한, 이를 검색하는 key도 transaction을 구분하는 하나의 key이면 충분하다. 또한, 하나의 요청에 대한 응답은 시간이 지나면 효용성이 없기 때문에 시스템에 문제가 생겨 그 transaction을 처리 하지 못한다 해서, 이를 백업하여 디스크에 가지고 있을 필요도 없다. 그러나, 여러 개의 프로세스가 접근하여 insert, delete, search 하는데 문제는 없어야 한다.

나. 구현과 설치의 용이성

비동기식 프로토콜을 처리하는 메모리 데이터베이스는 구현과 설치가 용이해야 한다. 부가적인 라이브러리나 제약 없이 설치 가능해야 한다.

다. 기본적인 데이터베이스의 기능

테이블을 만들고, 그 테이블에 대한 기본적인 insert, delete, search의 기능을 지원할 수 있어야 한다.

라. 가변형 레코드의 지원

프로토콜 packet 의 길이는 가변적일 수 있다. 어떤 요청이 컴퓨팅의 결과를 포함하고 있을 때는 그 크기가 매우 다양할 수 있다. 따라서, 메모리 데이터베이스

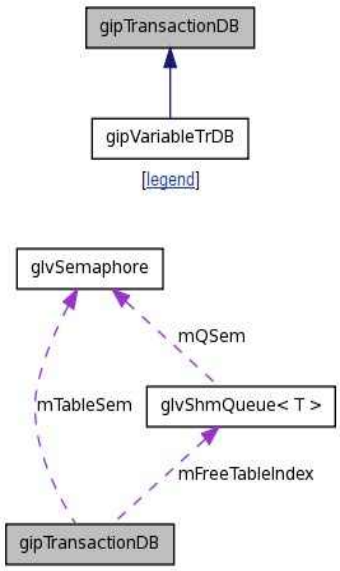
는 가변형 레코드를 지원해야 한다.

4. 메모리 데이터베이스의 설계 및 구현

리눅스는 시스템 함수로 shared memory와 semaphore를 제공한다. 본 문서에서는 간단한 리눅스의 시스템 함수로 shared memory 데이터베이스 시스템을 설계하고, 구현한다. 이를 위해 우리는 데이터베이스 전체를 구현하는 것이 아니라 2절의 요구사항에 맞는 몇 가지의 제약이 있는 시스템을 구현한다. 첫째 검색을 위한 key 는 하나 이상 지원하지 않는다. 둘째, insert, delete, update, search의 네 가지 함수만 지원한다. 셋째, CUI(Command line User Interface)는 지원하지 않는다. 단, 데이터베이스의 상태를 알기 위해 특정 테이블의 상태를 보여주는 CUI는 지원한다. 이러한 메모리 데이터베이스의 설계와 구현을 위해 우리는 리눅스에서 제공하는 IPC인 shared memory와 semaphore 의 시스템 함수를 사용한다. Shared memory 데이터베이스를 위한 클래스는 둘로 나뉜다. 첫번째는 고정 크기의 레코드 테이블을 다루는 gipTransactionDB 클래스이고 두번째는 가변 크기의 레코드를 다루는 gipVariableTrDB 클래스이다. 다음 각 절에서는 이 두 클래스의 설계와 구현에 대해 설명한다.

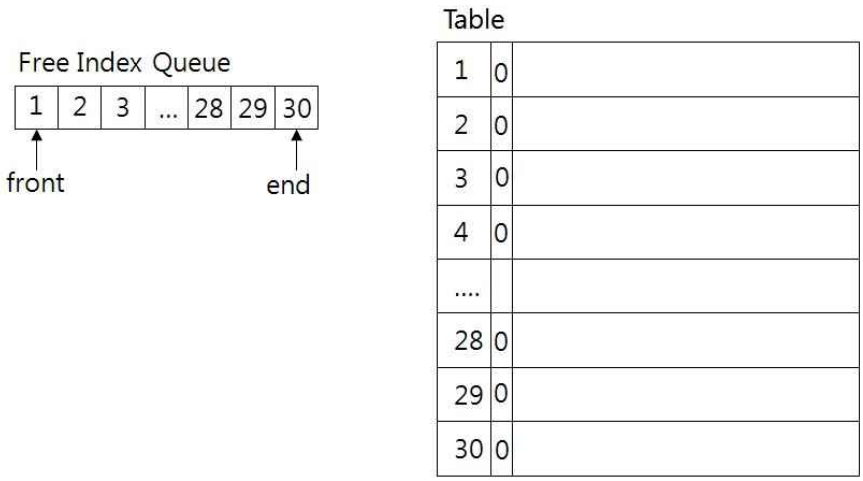
가. 고정 크기의 레코드 테이블을 위한 gipTransactionDB 클래스의 구현

그림 2는 gipTransactionDB 클래스의 inheritance diagram과 collaboration diagram을 보여 준다. Shared memory 데이터베이스는 기본적으로 데이터 접근 시 lock이 필요하다. 따라서, gipTrasactionDB 클래스는 리눅스 semaphore lock을 구현한 glvSemaphore 클래스를 use 한다. 또한, 사용 중이지 않은 클래스의 리스트를 유지하기 위해 역시 shared memory 상의 큐를 구현한 template 타입의 glvShmQueue를 use 한다.



[그림 3] Inheritance diagram and collaboration diagram for gipTransactionDB

1) CreateTable



[그림 4] CreateTable로 생성된 테이블과 테이블 관리를 위한 자료구조

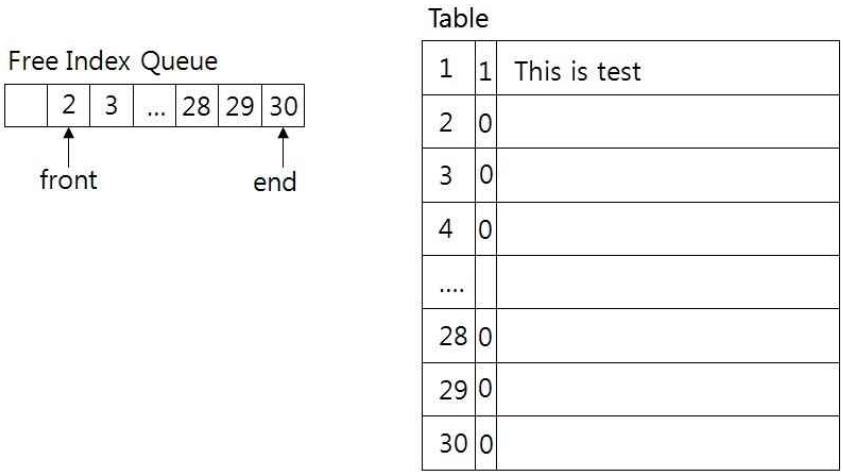
CreateTable은 shared memory 상에 하나의 테이블을 만들고, 테이블을 위한 semaphore를 생성한다. 또한, 테이블의 할당되지 않은 인덱스를 관리하는 free index queue을 생성하고 초기화 한다. 그림 3은 레코드의 수가 30인 하나의 테이블을 생성하였을 때, 초기화된 테이블과 큐의 상태를 보여 준다. Free index queue 역시 shared memory 상에 구현된 자료구조로 여러 프로세스에 의해 상호 배제적으로 접근가능하다. Free index queue는 circular array queue 로 구현되어

있으며 할당되지 않은 인덱스를 관리하므로 초기 free index queue에는 모든 인덱스가 다 들어 있다. 할당된 테이블 인덱스 레코드의 첫 번째 바이트는 1로 set 된다. 따라서 테이블 검색 시 할당된 레코드를 찾기 위해서는 첫 번째 바이트를 보면 된다. 아래의 소스 코드는 CreateTable을 구현한 것이다.

```
int gipTransactionDB::CreateTable(char* tableName, unsigned int trSize, int maxT
rNum, int keyPos, int keyLen)
{
   int      nRet;
   char     shmQSemName[128];

   strcpy(mTableName, tableName);
   mTrSize = trSize+sizeof(int)+1;

   mMaxTrNum = maxTrNum;
   mKeyPos = keyPos;
   mKeyLen = keyLen;

   mTableSem = new glvSemaphore();

   nRet = mTableSem->Create(mTableName);
    .............

   CreateDBIndexTable();

   /* allocate shared memory */
   mShmId = shmget(mDBKey, mTrSize*mMaxTrNum, IPC_CREAT|IPC_EXCL|0
666);
    ...........
   nRet = mFreeTableIndex.Create(mDBKey+1, shmQSemName, mMaxTrNum);
   ..........
   return TR_DB_SUCCESS;
}
```

CreateTable은 테이블을 테이블 이름인 tableName 과 생성 할 때 레코드의 크기인 trSize와 전체 테이블 레코드의 개수인 maxTrNum이 인자로 주어지며, 생성한 테이블에서 다루어질 데이터의 key의 위치인 keyPos와 그 key 의 길이인 keyLen도 인자로 주어진다.

2) Insert

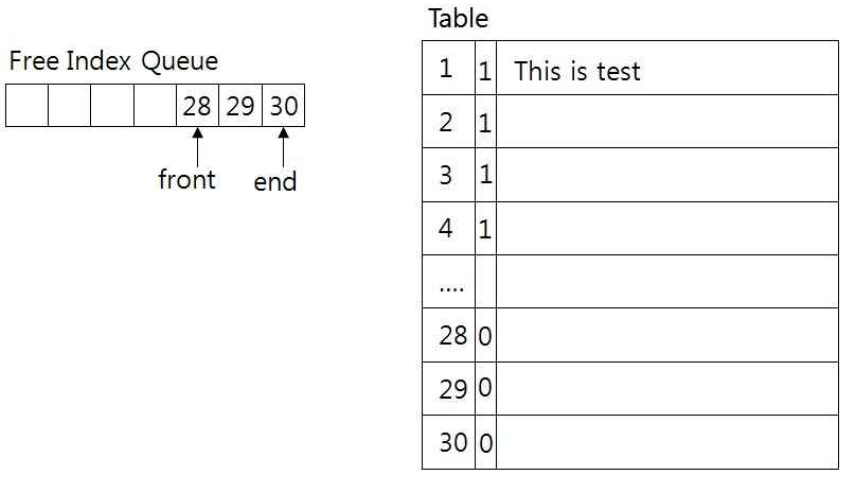


[그림 5] 레코드 삽입

Insert는 key 값을 명시하고 하나의 레코드를 메모리 데이터베이스에 추가한다. Insert는 그림 에서와 같이 free index queue에서 할당되지 않은 인덱스를 가져와 allocated index queue에 넣은 다음 table 의 index에 내용을 write한다. 동일한 key 값이 테이블에 존재할 경우 insert는 실패한다. 그림 4는 하나의 레코드를 테이블에 삽입할 때 동작을 보여 준다. 먼저 free index queue에서 하나의 인덱스를 받아 온다. 그 인덱스를 가지 테이블을 찾아가 첫 번째 바이트를 할당이 되었다는 의미로 1로 set 한다. 아래의 소스 코드는 이러한 동작을 구현한 gipTransactionDB 클래스의 Insert 함수이다. Insert 함수의 인자는 void* 타입의 keyValue와 그 keyValue의 길이, 전체 데이터인 void* 타입의 tr과 전체 데이터의 길이이다. 삽입하는 중에는 다름 프로세스의 접근을 막아야 하므로 mTableSem->Lock()을 걸고 시작한다. 데이터를 삽입하기 전 먼저 그 데이터가 테이블에 존재하는 지를 GetUnlockTransaction(keyValue, keyLen)으로 검색한다. 존재한다면 오류코드를 돌려준다. 존재 하지 않는다면, mFreeTableIndex.GetFrontAndDelete(&index)로 free index queue에서 하나의 index를 가져 온 다음, 그 인덱스에 데이터를 copy 한다. 다음으로 SetAllocated(index)로 index가 가리키는 레코드의 첫 번째 바이트를 1로 set 한다. 테이블삽입이 끝난 후에는 mTableSem->Release()로 해제한다.

```
int gipTransactionDB::Insert(void* keyValue, int keyLen, void* tr, unsigned int trLe
n)
{
    .............
    mTableSem->Lock();

    trans = GetUnlockTransaction(keyValue, keyLen);
    if (trans != NULL)
    {
        mTableSem->Release();
        return -trDBErrCode::DuplicatedKey;
    }
    nRet = mFreeTableIndex.GetFrontAndDelete(&index);
    ...........
    memcpy(mTable[index]+nPos, tr, trLen);
    SetAllocated(index);
    ...........
    mTableSem->Release();
    return index;
}
```

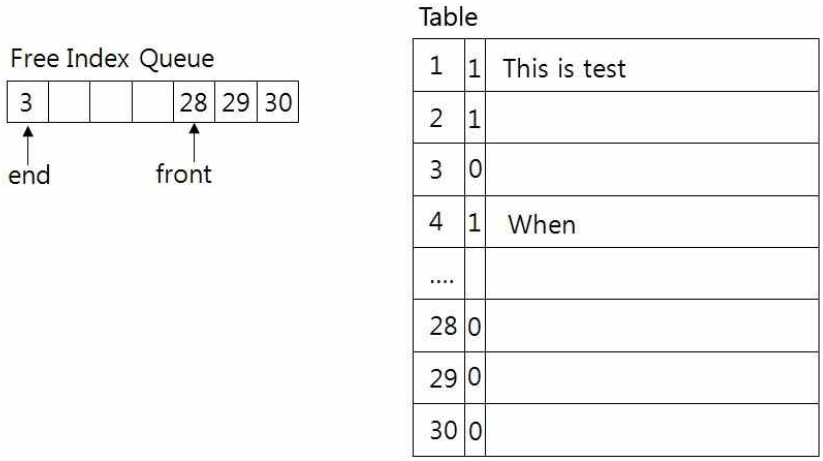
3) Delete



[그림 6] 레코드 삭제 전

Delete는 key값으로 하나의 레코드를 삭제한다. 그림 5는 레코드 삭제 전의 자료구조의 상태를 보여 준다. 28번째 인덱스까지 레코드가 삽입된 상태에서, 세

번째 인덱스의 레코드를 삭제하면 그림 6의 상태가 된다. Allocated index 리스트에서는 3이 삭제되고, free index queue에 3이 들어간다. Free index queue는 shared memory 상에 존재하는 circular array queue이다.



[그림 7] 레코드 삭제 후 (3의 레코드 삭제)

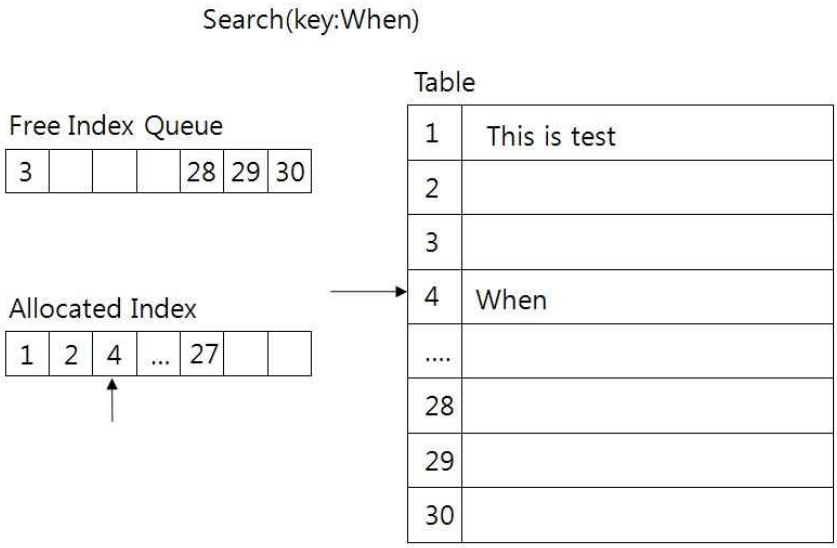
아래 소스 코드는 gipTransactionDB의 Delete를 구현한 것이다. Delete 역시 동작 중 다른 프로세스의 접근을 막기 위해 mTableSem->Lock()을 걸고 시작한다.

```
int gipTransactionDB::Delete(void* keyValue, int keyLen)
{
    ..............
    mTableSem->Lock();
    nPos = nPos + mKeyPos;
    for (int i = 0; i < mMaxTrNum; i++)
    {
        if (IsAllocated(i) != true) continue;
        if (memcmp(keyValue, mTable[i]+nPos, keyLen) == 0)
        {
            SetNotAllocated(i);
            mFreeTableIndex.Insert(&i);
            mTableSem->Release();
            mAllocIndexCnt--;
            return TR_DB_SUCCESS;
        }
    }
    mTableSem->Release();
    return -trDBErrCode::DeleteFail;
}
```

IsAllocated(i) 로 i가 이미 할당된 인덱스에 대해서만 key값을 비교하여 일치하는 key 값이 있으면, mFreeTableIndex.Insert(&i)를 통해 free index queue에 인덱스를 넣는다. 마지막으로 mTableSem->Release()로 락을 해제한다.

4) Search

Search는 하나의 key로 테이블을 검색하는 기능을 제공한다. 그러나, 프로그램의 특성상 레코드 상에서 값의 위치와 값의 길이를 주면, 그 값을 key로 대체하여 검색가능하다. Search는 allocated index list에 있는 모든 인덱스의 내용을 검색하여 key 값과 일치하는 값이 있는 지를 비교하여 값이 있을 경우 그 값을 돌려준다.

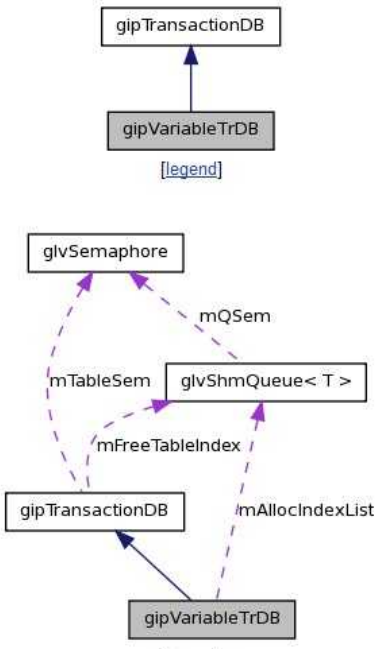


[그림 8] 레코드 검색

아래의 소스는 gipTransactionDB의 Search를 구현한 것이다. GetTransaction은 key 값으로 테이블을 검색하여 해당하는 레코드를 돌려주는 함수이다. 찾지 못했을 경우 NULL 값을 돌려준다. 검색 시 삽입과 삭제로 인하여 인덱스에 문제가 발생할 수 있으므로 역시 mTableSem->Lock()으로 락을 걸어 준다. IsAllocated(i)로 인덱스 i가 할당되었는지 여부를 check 한 후. memcmp를 이용하여 key 값을 비교한다. key 값의 type을 알 수 없으므로 메모리 데이터베이스 내에서의 key 값의 비교는 memcmp를 사용하여 수행한다. 찾을 경우 mTableSem->Release로 락을 해제하고 찾은 레코드를 돌려준다.

```
gipTrans* gipTransactionDB::GetTransaction(void* keyValue, int keyLen)
{
    ........
    mTableSem->Lock();
    for (int i = 0; i < mMaxTrNum; i++)
    {
        if (IsAllocated(i) != true) continue;

        if (memcmp((char*)keyValue, mTable[i]+nPos, keyLen) == 0)
        {
            memcpy(&mTrans.trLen, mTable[i]+1, sizeof(int));
            mTrans.trans = mTable[i]+sizeof(int)+1;
            mTableSem->Release();
            return &mTrans;
        }
    }
    mTableSem->Release();
    return NULL;
}
```

나. 가변 크기 레코드를 위한 gipVariableTrDB 클래스의 구현

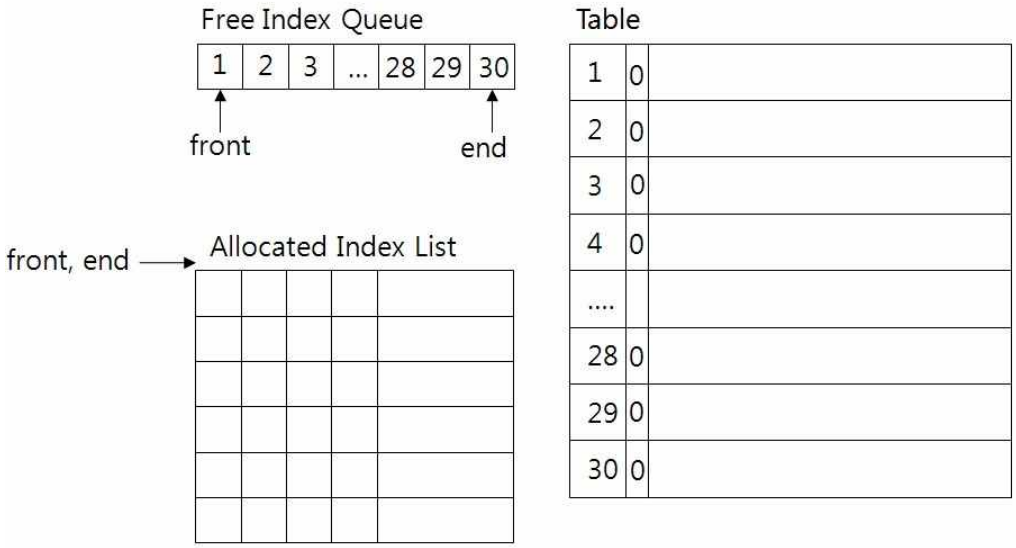


[그림 9] gipVariableTrDB class
inheritance and collaboration

그림 8은 gipVariableTrDB 클래스의 inheritance diagram과 collaboration diagram을 보여 준다. 가변 크기의 레코드를 다루는 gipVariableTrDB는 gipTransaction를 상속 받는다. 가변크기의 레코드는 여러 개의 고정 크기의 레코드로 이루진다는 것을 의미한다. Shared memory 데이터베이스는 기본적으로 데이터 접근시 lock이 필요하다. 따라서, gipTrasactionDB 클래스는 리눅스 semaphore lock을 구현한 glvSemaphore 클래스를 use 한다. 또한, 사용 중인 클래스와 사용중이지 않은 클래스의 리스트를 유지하기 위해 역시 shared memory 상의 큐를 구현한 template 타입의 glvShmQueue를 use 한다.

1) CreateTable

가변 레코드를 다루는 gipVariableTrDB는 gipTransactionDB를 상속한다. 가변크기의 레코드는 몇 개의 고정크기의 레코드를 더하여 하나의 레코드로 다룬다. CreateTable은 고정 크기의 레코드 테이블을 만들고 별도의 할당된 인덱스 리스트를 가지는 mAllocIndexList를 가진다. mAllocIndexList는 하나의 레코드 당 할당된 인덱스의 리스트를 관리한다. 그림 9는 CreateTable로 생성되어 초기화된 자료구조의 상태를 보여 준다.



[그림 10] CreateTable로 생성된 테이블과 테이블 관리를 위한 자료구조

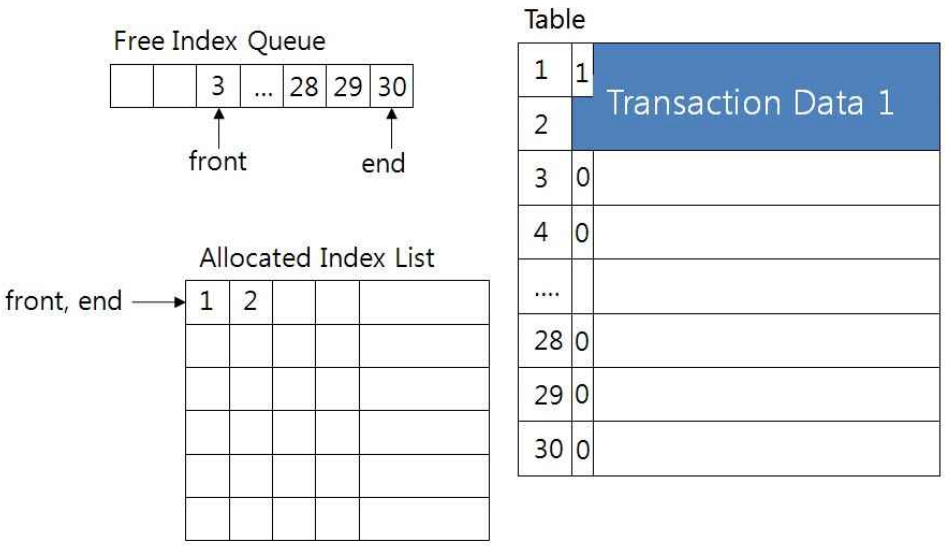
이를 구현한 것이 아래의 소스 코드이다. CreateTable은 gipTransactionDB의 Create 테이블을 호출하여 테이블을 생성하고, free index queue를 초기화한다. 다음으로 gipVariableTrDB 클래스에만 존재하는 자료구조로 Allocated Index를 관리하는 리스트를 mAllocIndexList.Create()를 호출하여 생성한다. 초기화가 끝나면 free index queue는 모든 인덱스를 가지고 있고, allocated index list는 비어 있다.

```
int gipVariableTrDB::CreateTable(char* tableName, unsigned int trSize, int maxTr
Num, int keyPos, int keyLen)
{
    int       nRet;

    nRet = gipTransactionDB::CreateTable(tableName, trSize, maxTrNum, keyPos,
                                         keyLen);

    maxTrNum = mMaxTrNum;

    if (nRet < 0)
    {
        printf("gipTransactionDB::CreateTable fail(%s, %d, %d, %d, %d)\n", tableNa
me, trSize, maxTrNum, keyPos, keyLen);
        return nRet;
    }
    sprintf(mAllocListSemName, "%s__alloc", mTableName);
```

```
    nRet = mAllocIndexList.Create(mDBKey+2, mAllocListSemName, maxTrNum);
    ......
    return TR_DB_SUCCESS;
}
```

2) Insert

그림 10은 가변 레코드를 다루는 gipVariableTrDB 클래스 상에서 하나의 레코드를 삽입하는 과정을 보여준다. Insert 함수는 Transaction Data1이 두 개의 고정 레코드를 필요로 하므로, free index queue에서 두 개의 인덱스를 받아 온다. 그리고, 이 둘을 Allocated index list에 넣는다. 테이블 인덱스의 레코드가 할당된 것을 의미하는 1번 인덱스의 첫 번째 바이트를 1로 set 하고 Transaction Data 1을 write 한다. 이 때 두 번째 레코드의 더 이상 indication 필드가 아니라 데이터 필드로 사용된다.



[그림 11] 가변 레코드 삽입

아래의 소스 코드는 이러한 동작을 구현한 것이다. gipVariableTrDB 클래스의 Insert 함수는 gipTransactionDB의 Insert 함수와 인자는 동일하다. 만일 삽입하고자 하는 레코드의 key 값이 테이블에 존재한다면 오류코드를 돌려주는 것도 동일하다. 그러나, 레코드의 크기에 따라 free index queue에서 가져 오는 인덱스의 개수가 다르기 때문에 삽입하고자 하는 데이터의 크기인 trLen이 몇 개의 고정 크기의 레코드를 필요로 하는 지를 먼저 계산한다. 다음으로 mFreeIndexFrontAndDelete(indexList.list, indexList.num)을 호출함으로써, free index queue 로부터 필요한 수만큼의 인덱스를 받아 온다. 그리고, 그 리스트의 첫 번째 인덱스

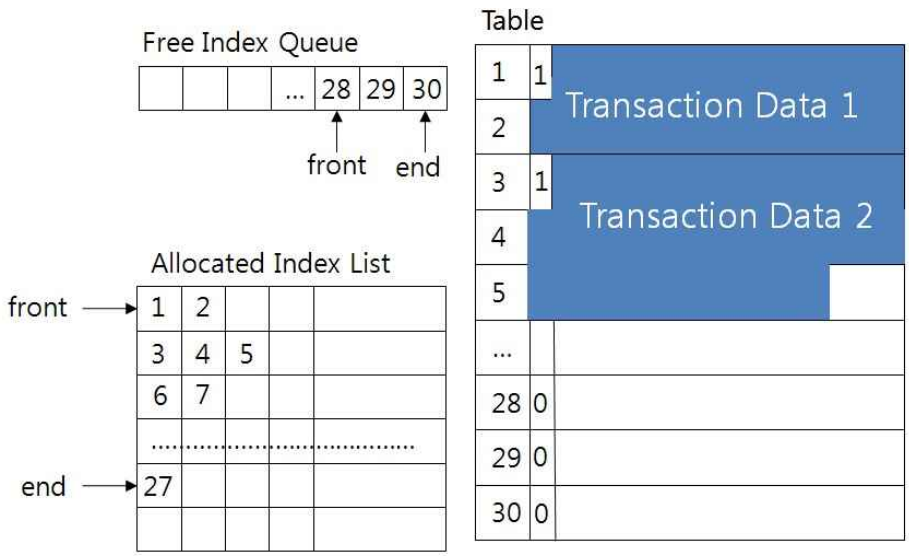
인 indexList.list[0]의 인덱스로 찾아가 SetAllicated(index) 함수를 호출하여 첫 번째 바이트를 1로 set한다. 마지막으로 데이터를 copy하고, mAllocIndexList.insert(&indexList)를 호출하여 allocated index list 에 할당 받은 인덱스를 추가한다.

```
int gipVariableTrDB::Insert(void* keyValue, int keyLen, void* tr, unsigned int trLen)
{
   ..........
   struct gipTrAllocIndexList indexList;

   mTableSem->Lock();

   trans = GetUnlockTransaction(keyValue, keyLen);

   if(trans != NULL)
   {
      printf("%s Duplicated key\n", mTableName);
      mTableSem->Release();
      return -trDBErrCode::DuplicatedKey;
   }
   /* calcuate the number of indexes */
   totalLen = trLen+sizeof(trLen)+1;
   trNum = totalLen/mTrSize;
   nRemainder = totalLen % mTrSize;

   if (nRemainder != 0)
   {
      indexList.num = trNum+1;
      nRet = mFreeTableIndex.GetFrontAndDelete(indexList.list, indexList.num);
   }
   else
   {
      indexList.num = trNum;
      nRet = mFreeTableIndex.GetFrontAndDelete(indexList.list, indexList.num);
   }
   ..............................
   index = indexList.list[0];
   SetAllocated(index);
  ........................................
```

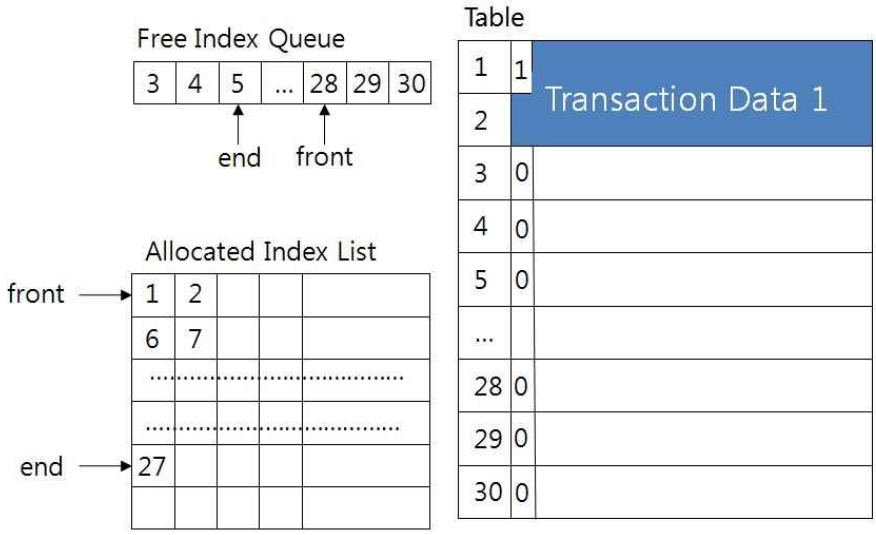
```
    memcpy(mTable[index]+nPos, tr, trLen);
    mAllocIndexList.Insert(&indexList);

    mTableSem->Release();
    return TR_DB_SUCCESS;
}
```

3) Delete

gipVariableTrDB 클래스의 delet 함수는 key 값으로 하나의 레코드를 삭제하는 기능을 제공한다. 그림 11은 인덱스 27까지 할당된 가변 레코드 테이블을 보여준다. Transaction Data2를 테이블에서 삭제한다고 할 때, delete 함수는 allocated index list에서 할당된 리스트들을 받아와 지우고자 하는 key 값과 일치하는 data를 찾는다. Transaction Data2를 찾으면, 그 데이터가 차지한 레코드에 해당하는 인덱스에 해당하는 테이블의 첫 번째 바이트를 모두 0으로 만든다. 다음으로 이들 인덱스들을 free index queue에 삽입하고 allocated index list 로부터 지운다. 이러한 과정을 거친 후 자료구조는 그림 12와 같아진다.



[그림 12] 레코드 삭제 전



[그림 13] 레코드 삭제 후

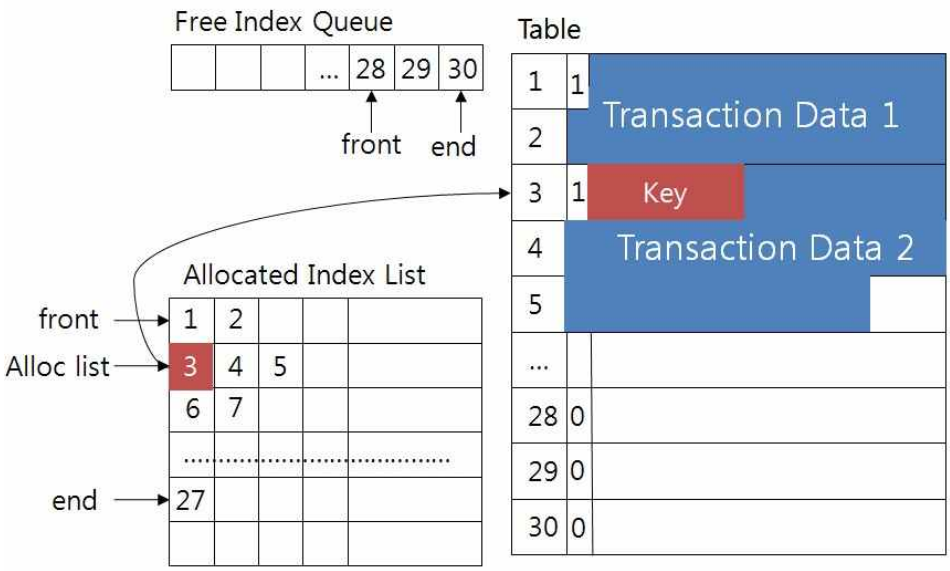
아래의 소스 코드는 gipVariableTrDB에서 이러한 동작을 구현한 것이다. mAlloc IndexList.GetNextFront() 함수로 allocat 된 인덱스의 리스트를 받아 온다. 이들 리스트의 첫 번째 인덱스의 첫 번째 바이트가 1로 set 되어 있으면, 할당된 레코드라 보고 key 값을 비교한다. key 값이 일치할 경우, 그 인덱스의 첫 번째 바이트를 0으로 되돌려 주고, FreeIndexInTable(&index) 함수를 호출하여 free index queue에 이 레코드가 차지한 인덱스들을 삽입한다. mAllocIndexList.Delete(&indexList)함수를 호출하여 allocated index list에서 이들 인덱스들을 지운다.

```
int gipVariableTrDB::Delete(void* keyValue, int keyLen)
{
    ..........................
    struct gipTrAllocIndexList indexList;

    mTableSem->Lock();

    nAllocNum = mAllocIndexList.GetItemNum();
    if (nAllocNum == 0)
    {
        mTableSem->Release();
        return -trDBErrCode::DeleteFail;
    }
    ..................
    int i = 0;
```

```
    while (i < nAllocNum)
    {
        nRet = mAllocIndexList.GetNextFront(&indexList);

        if (nRet < 0) break;

        index = indexList.list[0];

        if (IsAllocated(index) != true) continue;
        if (memcmp(keyValue, mTable[index]+nPos, keyLen) == 0)
        {
            SetNotAllocated(index);
            FreeIndexInTable(&indexList);
            mAllocIndexList.Delete(&indexList);
            mTableSem->Release();
            return TR_DB_SUCCESS;
        }
        i++;
    }
    mTableSem->Release();
    return -trDBErrCode::DeleteFail;
}
```

4) Search

가변 레코드를 다루는 gipVariableTrDB 클래스의 search는 key 값으로 데이터를 검색하는 기능을 제공한다. 그림 13은 gipVariableTrDB 클래스의 검색 동작을 보여 준다. 먼저 Allocated Index List를 처음부터 검색하여, 해당 인덱스가 가리키는 테이블에서 key 값을 비교한다. key 값이 일치할 경우, 찾은 데이터를 돌려준다.



[그림 14] 레코드 검색

아래의 소스 코드는 gipVariableTrDB 클래스의 GetTransaction 함수에서 이러한 동작을 구현한 것이다. Delete 함수와 마찬가지로, 먼저, mAllocIndexList.GetNextFront() 함수로 allocat 된 인덱스의 리스트를 받아 온다. 이들 리스트의 첫 번째 인덱스의 첫 번째 바이트가 1로 set 되어 있으면, 할당된 레코드라 보고 key 값을 비교한다. 일치하면 이 데이터를 돌려주고, 끝까지 못 찾을 경우, NULL을 값을 돌려준다.

```
gipTrans* gipVariableTrDB::GetTransaction(void* keyValue, int keyLen)
{
    ..................
    struct gipTrAllocIndexList indexList;

    mTableSem->Lock();

    nAllocNum = mAllocIndexList.GetItemNum();

    if (nAllocNum == 0)
    {
        mTableSem->Release();
        return NULL;
    }
    ....................
    while (i < nAllocNum)
    {
```

```
        nRet = mAllocIndexList.GetNextFront(&indexList);

        if (nRet < 0) break;

        index = indexList.list[0];

        if (IsAllocated(index) != true) continue;

        if (memcmp((char*)keyValue, mTable[index]+nPos, keyLen) == 0)
        {
            memcpy(&mTrans.trLen, mTable[index]+1, sizeof(int));
            mTrans.trans = mTable[index]+sizeof(int)+1;
            mTableSem->Release();
            return &mTrans;
        }
        i++;
    }

    mTableSem->Release();

    return NULL;
}
```

5. 결론

이상에서 살펴 본 바와 같이 비동기식 프로토콜을 처리하기 위한 메모리 데이터베이스는 리눅스에서 제공하는 매우 간단한 semaphore 와 shared memory를 활용하여 구현 가능하다. 이 때문에 설치 시, 라이브러리 하나만 링크하면, 간단한 메모리 데이터베이스의 기능을 사용할 수 있다. 또한 show [table name] 과 같은 간단한 CUI로 실행 중에 테이블의 상태를 확인할 수 도 있다. 이러한 메모리 데이터베이스는 비동기식 프로토콜의 transaction 처리를 위해서 뿐만 아니라 프로세스 간 공유가 필요한 데이터 중 insert 와 delete, update등의 매우 간단한 기능을 필요로 하는 모든 응용에 활용 가능하다.

6. 참고문헌

[1] Toshiyuki Kimura, "Asynchronous Communication Models for JAX-RPC

2.0", NTT Data Corperation, 2003

[2] Min Ah Kim, "GLOVE(GLObal Virtual reality Environment for scientific simulation): VR환경에서의 대용량 데이터 가시화 시스템", 정보과학회, 2010.

[4] "TeleCommunication Systems Signs up as a Reseller of TimesTen; Mobile Operators and Carriers Gain Real-Time Platform for Location-Based Services". Business Wire. 2002-06-24. http://findarticles.com/p/articles/mi_m0EIN/is_2002_June_24/ai_87694370.

[5] PR Newswire (2003-04-28). "Solid Announces General Availability of BoostEngine 4.0, the First On-Disk/In-Memory Hybrid Database Manager". Press release. http://www.thefreelibrary.com/Solid+Announces+General+Availability+of+BoostEngine+4.0,+the+First...-a0100738850.

[6] http://www.intelligententerprise.com/channels/business_intelligence/showArticle.jhtml?articleID=210700171

[7] http://www.sybase.com/products/databasemanagement/adaptiveserverenterprise

GLOVE Reference Manual

0.01

Generated by Doxygen 1.4.7

Mon Nov 15 14:21:50 2010

Contents

Chapter 1

GLOVE Hierarchical Index

1.1 GLOVE Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

GLOVE Class Index

2.1 GLOVE Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

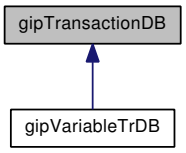
GLOVE Class Documentation

3.1 gipTransactionDB Class Reference

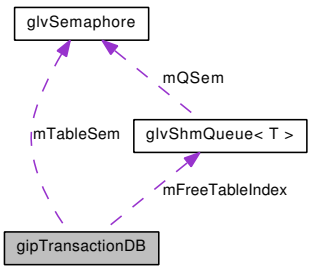
gipTransactionDB (p. ??) is a class for a table with static size records on shared memory.

`#include <gipTransactionDB.h>`

Inheritance diagram for gipTransactionDB:



Collaboration diagram for gipTransactionDB:



Public Member Functions

- **gipTransactionDB (key_t shmKey)**
- **~gipTransactionDB ()**

Destructor.

- **int CreateTable (char *tableName, unsigned int transactionSize, int maxTrNum, int keyPos, int keyLen)**
- **int Insert (void *keyValue, int keyLen, void *tr, unsigned int trLen)**
- **int Insert (void *keyValue, void *tr, unsigned int trLen)**
- **gipTrans * GetTransaction (void *keyValue, int keyLen)**
- **gipTrans * GetTransactionByIndex (int index)**
- **gipTrans * GetUnlockTransaction (void *keyValue, int keyLen)**
- **gipTrans * GetTransaction (void *keyValue)**
- **int Delete (void *keyValue, int keyLen)**
- **int Delete (void *keyValue)**
- **int DeleteByIndex (int index)**

3.1.1 Detailed Description

gipTransactionDB (p. **??**) is a class for a table with static size records on shared memory.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 gipTransactionDB::gipTransactionDB (key_t *shmKey*)

Constructor

Parameters:

shmKey means shared memory key for a table

3.1.3 Member Function Documentation

3.1.3.1 int gipTransactionDB::CreateTable (char * *tableName*, unsigned int *transactionSize*, int *maxTrNum*, int *keyPos*, int *keyLen*)

Create a memory db table

Parameters:

tableName,transactionSize,maxTrNum,keyPos,keyLen

Returns:

success or fail

Reimplemented in **gipVariableTrDB p.** (classgipVariableTrDB_74321afb7b677e97f441d77ca63aef1 **??**)

3.1.3.2 int gipTransactionDB::Delete (void * *keyValue*)

delete a transaction record for an key value when the key position and the key length are already known.

Parameters:

keyValue

Returns:

success or fail

Reimplemented in **gipVariableTrDB p.** (classgipVariableTrDB_c0bf6a5de5c62dd25256b4db41e52b73 ??)

3.1.3.3 int gipTransactionDB::Delete (void * *keyValue*, int *keyLen*)

delete a transaction record for an key value when the key position is already known.

Parameters:

keyValue,keyLen

Returns:

success or fail

Reimplemented in **gipVariableTrDB p.** (classgipVariableTrDB_27ec4eca4a3ef3a95ac62658e0f9f90d ??)

3.1.3.4 int gipTransactionDB::DeleteByIndex (int *index*)

delete a transaction record by the table index

Parameters:

index

Returns:

success or fail

Reimplemented in **gipVariableTrDB p.** (classgipVariableTrDB_044407e1ba67553634d7e5e0971d6f5d ??)

3.1.3.5 gipTrans* gipTransactionDB::GetTransaction (void * *keyValue*)

get a transaction record for an key value when key position and key length are already known.

Returns:

a transaction record data

Reimplemented in **gipVariableTrDB p.** (classgipVariableTrDB$_{fb3ccb94c64eaddf62ff85fc6e2c9746}$??)

3.1.3.6 gipTrans* gipTransactionDB::GetTransaction (void * *keyValue*, int *keyLen*)

get a transaction record for a key

Parameters:

keyValue*,*keyLen

Returns:

a transaction record data

Reimplemented in **gipVariableTrDB p.** (classgipVariableTrDB$_{91d29b5b6d1c219c2c969ff279c15f90}$??)

3.1.3.7 gipTrans* gipTransactionDB::GetTransactionByIndex (int *index*)

get a transaction record for an index

Parameters:

index

Returns:

a transaction record data

3.1.3.8 gipTrans* gipTransactionDB::GetUnlockTransaction (void * *keyValue*, int *keyLen*)

get a transaction record for an index without locking

Parameters:

keyValue*,*keyLen

Returns:

a transaction record data

Reimplemented in **gipVariableTrDB p.** (classgipVariableTrDB$_{e6289a28bf0c6083053b638542f4f690}$??)

3.1.3.9 int gipTransactionDB::Insert (void * *keyValue*, void * *tr*, unsigned int *trLen*)

insert a record for a transaction when key position and key length are already known.

Parameters:

keyValue,tr,trLen

Returns:

success or fail

Reimplemented in **gipVariableTrDB p.** (classgipVariableTrDB$_0$e76208e6238c0ad732872164afab6a0 ??)

3.1.3.10 int gipTransactionDB::Insert (void * *keyValue*, int *keyLen*, void * *tr*, unsigned int *trLen*)

insert a record for a transaction

Parameters:

keyValue,keyLen,tr,trLen

Returns:

success or fail

Reimplemented in **gipVariableTrDB p.** (classgipVariableTrDB$_4$cf0d64a6ef2da47832beaa6a32f5de7 ??)

The documentation for this class was generated from the following file:

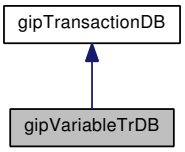
- /home/petimina/glove/trunk/include/trDB/gipTransactionDB.h

3.2 gipVariableTrDB Class Reference

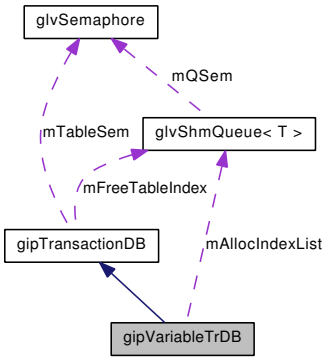
gipVariableTrDB (p. ??) is a class for a table with variable size records on shared memory.

`#include <gipVariableTrDB.h>`

Inheritance diagram for gipVariableTrDB:



Collaboration diagram for gipVariableTrDB:



Public Member Functions

- **gipVariableTrDB (key_t shmKey)**
- **~gipVariableTrDB ()**

  Destructor.

- **int CreateTable (char *tableName, unsigned int transactionSize, int maxTrNum, int keyPos, int keyLen)**
- **int Insert (void *keyValue, int keyLen, void *tr, unsigned int trLen)**
- **int Insert (void *keyValue, void *tr, unsigned int trLen)**
- **gipTrans * GetTransaction (void *keyValue, int keyLen)**
- **gipTrans * GetTransaction (void *keyValue, int keyLen, int keyIndex)**
- **gipTrans * GetUnlockTransaction (void *keyValue, int keyLen)**
- **gipTrans * GetTransaction (void *keyValue)**
- **int Delete (void *keyValue, int keyLen)**

- **int Delete (void *keyValue, int keyLen, int keyIndex)**
- **int Delete (void *keyValue)**
- **int DeleteByIndex (int index)**
- **int FreeIndexInTable (struct gipTrAllocIndexList *indexList)**
- **void Print ()**

  print table contents

- **void Clear ()**

  delete all data in the memory database table

3.2.1 Detailed Description

gipVariableTrDB (p. ??) is a class for a table with variable size records on shared memory.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 gipVariableTrDB::gipVariableTrDB (key_t *shmKey*) [inline]

Constructor

Parameters:

shmKey

3.2.3 Member Function Documentation

3.2.3.1 int gipVariableTrDB::CreateTable (char * *tableName*, unsigned int *transactionSize*, int *maxTrNum*, int *keyPos*, int *keyLen*)

Create a memory db table with **gipTransactionDB** (p. ??) CreateTable

Parameters:

tableName,transactionSize,maxTrNum,keyPos,keyLen

Returns:

success or fail

Reimplemented from **gipTransactionDB** **p.** (classgipTransactionDB$_d$5ba0440b0a9301562657b714a4785c0 ??)

3.2.3.2 int gipVariableTrDB::Delete (void * *keyValue*)

delete a transaction record for an key value when the key position and the key length are already known.

Parameters:

keyValue

Returns:

success or fail

Reimplemented from **gipTransactionDB** p. (classgipTransactionDB$_a$77741dc4674b1cc0db4328253952e44 ??)

3.2.3.3 int gipVariableTrDB::Delete (void * *keyValue*, int *keyLen*, int *keyIndex*)

delete a transaction record for an key value when the key position and the key length are already known.

Parameters:

keyValue,keyLen,keyIndex

Returns:

success or fail

3.2.3.4 int gipVariableTrDB::Delete (void * *keyValue*, int *keyLen*)

delete a transaction record for an key value when the key position is already known.

Parameters:

keyValue,keyLen

Returns:

success or fail

Reimplemented from **gipTransactionDB** p. (classgipTransactionDB$_0$$b0a3f86b04ac6b2c48635e8c75ed$943 ??)

3.2.3.5 int gipVariableTrDB::DeleteByIndex (int *index*)

delete a transaction record by the table index

Parameters:

index

Returns:

success or fail

Reimplemented from **gipTransactionDB** **p.** (classgipTransactionDB$_{dc6c971ebdb9098c912564e62cb76d9c}$??)

3.2.3.6 int gipVariableTrDB::FreeIndexInTable (struct gipTrAllocIndexList * *indexList*)

free a index in the allocated index list.

Parameters:

indexList

Returns:

success or fail

3.2.3.7 gipTrans* gipVariableTrDB::GetTransaction (void * *keyValue*)

get a transaction record for an key value when key position and key length are already known.

Parameters:

keyValue

Returns:

a transaction record data

Reimplemented from **gipTransactionDB** **p.** (classgipTransactionDB$_{cd7099cd781586051bd2681942c7a2b5}$??)

3.2.3.8 gipTrans* gipVariableTrDB::GetTransaction (void * *keyValue*, int *keyLen*, int *keyIndex*)

get a transaction record for an index

Parameters:

keyValue,keyLen,keyIndex

Returns:

a transaction record data

3.2.3.9 gipTrans* gipVariableTrDB::GetTransaction (void * *keyValue*, int *keyLen*)

get a transaction record for a key

Parameters:

keyValue,keyLen

Returns:

a transaction record data

Reimplemented from **gipTransactionDB** p. (classgipTransactionDB$_4$4ff41ba884f5e0ce440ac188a5d3ca1 ??)

3.2.3.10 gipTrans* gipVariableTrDB::GetUnlockTransaction (void * *keyValue*, int *keyLen*)

get a transaction record for an index without locking

Parameters:

keyValue,keyLen

Returns:

a transaction record data

Reimplemented from **gipTransactionDB** p. (classgipTransactionDB$_6$a38756717493af075b83dab405ce346 ??)

3.2.3.11 int gipVariableTrDB::Insert (void * *keyValue*, void * *tr*, unsigned int *trLen*)

insert a record for a transaction when key position and key length are already known.

Parameters:

keyValue,tr,trLen

Returns:

success or fail

Reimplemented from **gipTransactionDB** p. (classgipTransactionDB$_b$767568b86fac25af75113c4e86c6$aab ??)

3.2.3.12 int gipVariableTrDB::Insert (void * *keyValue*, int *keyLen*, void * *tr*, unsigned int *trLen*)

insert a record for a transaction

Parameters:

keyValue,keyLen,tr,trLen

Returns:

success or fail

Reimplemented from **gipTransactionDB** **p.** (classgipTransactionDB$_d$4e9e25c07301793159c7e8313f33ebb ??)

The documentation for this class was generated from the following file:

- /home/petimina/glove/trunk/include/trDB/gipVariableTrDB.h

3.3 glvSemaphore Class Reference

glvSemaphore (p. **??**) class is a class for supporting linux semaphore

`#include <glvSemaphore.h>`

Public Member Functions

- **glvSemaphore ()**
  Constructor.
- **∼glvSemaphore ()**
  Destructor.
- **int Create (char *semName)**
- **int Lock ()**
  lock the semaphore
- **int Release ()**
  release the semaphore
- **int Close ()**
  close the semaphore

3.3.1 Detailed Description

glvSemaphore (p. **??**) class is a class for supporting linux semaphore

3.3.2 Member Function Documentation

3.3.2.1 int glvSemaphore::Create (char * *semName*)

Create a named semaphore

Parameters:

semName

Returns:

success or fail

The documentation for this class was generated from the following file:

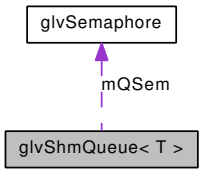
- /home/petimina/glove/trunk/include/util/glvSemaphore.h

3.4 glvShmQueue< T > Class Template Reference

glvShmQueue (p. ??) class is a template class for a queue on shared memory that processes can share.

`#include <glvShmQueue.h>`

Collaboration diagram for glvShmQueue< T >:



Public Member Functions

- **glvShmQueue ()**
  Constructor.
- **~glvShmQueue ()**
  Destructor.
- **int Create (key_t shmKey, char *semName, int qSize)**
- **int Insert (T item)**
- **void Print ()**
  Print all items in the queue.
- **int InsertBackFreeList (int index)**
- **int GetFrontFreeList ()**
- **int Insert (T *itemPtr)**
- **int Insert (T *itemPtr, int num)**
- **int Delete (T *item)**
- **void Delete ()**
  delete a item from the front of the queue
- **int GetFront (T *item)**
- **int GetFrontAndDelete (T *item, int num)**

3.4.1 Detailed Description

template<typename T> class glvShmQueue< T >

glvShmQueue (p. ??) class is a template class for a queue on shared memory that processes can share.

3.4.2 Member Function Documentation

3.4.2.1 template<typename T> int glvShmQueue< T >::Create (key_t *shmKey*, char * *semName*, int *qSize*) [inline]

Create a shared memory queue

Parameters:

shmKey,semName,qSize

Returns:

success or fail

3.4.2.2 template<typename T> int glvShmQueue< T >::Delete (T * *item*) [inline]

delete a item from the queue

Parameters:

item : a item pointer

Returns:

success or fail

3.4.2.3 template<typename T> int glvShmQueue< T >::GetFront (T * *item*) [inline]

Get a front item from the queue without deletion.

Parameters:

item pointer to save a getting item

3.4.2.4 template<typename T> int glvShmQueue< T >::GetFrontAndDelete (T * *item*, int *num*) [inline]

Get front items from the queue withdeletion.

Parameters:

item,num item pointer to save getting items and the number of getting item

3.4.2.5 template<typename T> int glvShmQueue< T >::GetFrontFreeList () [inline]

Get the index of an item from the front of the queue

Returns:

the index

3.4.2.6 template<typename T> int glvShmQueue< T >::Insert (T * *itemPtr*, int *num*) [inline]

Insert a list of items into the queue

Parameters:

itemPtr,num a list of items and the number of items

Returns:

success or fail

3.4.2.7 template<typename T> int glvShmQueue< T >::Insert (T * *itemPtr*) [inline]

Insert a list of items into the queue

Parameters:

itemPtr a list of items

Returns:

success or fail

3.4.2.8 template<typename T> int glvShmQueue< T >::Insert (T *item*) [inline]

Insert a item into the queue

Parameters:

item an item

Returns:

success or fail

3.4.2.9 template<typename T> int glvShmQueue< T >::InsertBackFreeList (int *index*) [inline]

Insert the index of an item into the back of the queue

Parameters:

index

Returns:

success or fail

The documentation for this class was generated from the following file:

- /home/petimina/glove/trunk/include/util/glvShmQueue.h

3.5 trDBErrCode Class Reference

trDBErrCode (p. **??**) is a class for dealing with error codes for shared memory DB.

`#include <trDBError.h>`

3.5.1 Detailed Description

trDBErrCode (p. **??**) is a class for dealing with error codes for shared memory DB.

The documentation for this class was generated from the following file:

- /home/petimina/glove/trunk/include/trDB/trDBError.h