GOLDILOCKS LITE 3.1 Manual (ko)



07228 서울특별시 영등포구 영신로 220 KnK디지털타워 1109호

전화: 02-322-6288 팩스: 02-322-6788

Webpage: www.sunjesoft.com

Support: technet@sunjesoft.com

GOLDILOCKS LITE 3.1 Manual (ko)

SUNJESOFT Inc

차례

1. Getting Started 1
   1.1 개요 2
   1.2 Quick Start 12
   1.3 구문 20
   1.4 DICTIONARY 90
   1.5 dbmMetaManager 104
   1.6 복구 가이드 113
   1.7 Utility 117
   1.8 Sizing 132
   1.9 Monitoring 135
2. API Reference 141
   2.1 API 공통사항 142
   2.2 C/C++ APIs 142
   2.3 JAVA 258
   2.4 Python 연동 283
   2.5 GO Lang 286
   2.6 Error Message 291

1.

Getting Started

1.1 개요

GOLDILOCKS LITE는 트랜잭션 기능을 지원하는 shared memory 기반의 데이터 관리 library이다.

주요 특징은 다음과 같다.

- Shared memory (POSIX) direct attached 방식으로 동작하는 C/C++ library
- Shared memory 자동 확장 및 최대 크기 제한 기능 제공
- 장애 복구와 안정성을 위한 disk logging 기능 제공
- C 언어 기반의 직관적인 API (Application Interface) 제공
- 운영 관리용 SQL syntax 제공
- Array, B tree, Splay, Queue 등 다양한 접근 방식 지원
- 트랜잭션 기능 제공 (Atomic, concurrency, durability 등 지원)

다음은 사용자 프로그램에서 데이터를 관리하는 방식을 보여주는 구조체 예제이다.

```
struct user_data
{
    int             mEmpNo;
    char            mEmpName[20];
    int             mDeptNo;
    struct timeval  mBirth
}
```

- 다음 예제와 같이 구조체와 동일한 형태의 테이블을 생성한다.

```
dbmMetaManager(DEMO)> create table user_data
(
    empNo       int,
    empName     char(20),
    deptNo      int,
    birth       date
)
success
dbmMetaManager(DEMO)>
```

- 다음 예제와 같이 인덱스를 생성한다.

```
dbmMetaManager(DEMO)> create unique index idx_user_data on user_data( empNo);
success
```

- 다음의 예제와 같이 데이터를 삽입/조회할 수 있다.

```
dbmMetaManager(DEMO)> desc user_data;
-------------------------------------------------------------------------------------
Instance=(DEMO) Table=(USER_DATA) Type=(TABLE) RowSize=(40) LockMode(1)
-------------------------------------------------------------------------------------
EMPNO                           int                 4          0
EMPNAME                         char                20         4
DEPTNO                          int                 4          24
BIRTH                           date                8          32
-------------------------------------------------------------------------------------
IDX_USER_DATA                   unique     (EMPNO asc)
-------------------------------------------------------------------------------------
success
dbmMetaManager(DEMO)> insert into user_data values (1, 'alice', 100, sysdate);
success
dbmMetaManager(DEMO)> commit;
success
dbmMetaManager(DEMO)> select * from user_data;
-----------------------------------------------------------------------------
EMPNO   : 1
EMPNAME : alice
DEPTNO  : 100
BIRTH   : 2025/07/30 08:08:15.484030
-----------------------------------------------------------------------------
1 row selected
```

- 제공되는 API를 활용해 다음과 같이 개발할 수 있다.

```
dbmHandle * Handle = NULL;
struct user_data xData;
// 세션 할당
dbmInitHandle( &Handle, NULL );
// 데이터 저장 API 예
dbmInsertRow( Handle, "node", &xData, sizeof(struct user_data));
// 데이터 조회 API 예
dbmSelectRow( Handle, "node", &xData );
...
```

노트

- dbmMetaManager는 사용자가 GOLDILOCKS LITE를 관리할 수 있도록 제공되는 interactive tool 이다. (자세한 내용은 **dbmMetaManager** 를 참조한다.)
- API에 대한 자세한 내용은 **C/C++ APIs**를 참조한다.

Object

GOLDILOCKS LITE에서 object는 shared memory 상에 생성되는 모든 유형의 segment를 의미한다.

DICTIONARY

Dictionary는 object 정보를 관리하기 위해 instance 생성 시점에 자동으로 생성되며, object 관련 테이블과 상태 정보를 확인할 수 있도록 제공되는 내부 view들을 의미한다. (자세한 내용은 **DICTIONARY** 를 참조한다.)

INSTANCE

Instance는 GOLDILOCKS LITE를 사용하는 세션 정보와 트랜잭션을 처리하기 위한 공간으로 사용된다. 일반 RDBMS의 undo segment나 SGA/PGA와 유사한 의미를 갖는다.

노트

GOLDILOCKS LITE에서 세션이란 **dbmInitHandle/ dbmConnect API**를 통해 instance에 공간을 할당받아 실행되는 프로그램을 의미한다.

Instance는 다음과 같은 용도로 사용된다.

항목	설명
Session area	Instance에 접근하는 세션의 정보를 저장하는 공간이다. (SessionID, PID, status 등)
Transaction area	Session에서 발생한 트랜잭션의 정보를 저장하는 공간이다. (트랜잭션 로깅 등)
Rollback image area	트랜잭션에 의해 변경되기 전의 record image를 저장하는 공간이다.

노트

- INSTANCE에는 두 가지 유형이 있다.
  - Dictionary instance: **initdb**에 의해 생성되는 최상위 instance 이다.

    - User instance: 사용자가 직접 생성하는 instance 이다.
- 하나의 instance에는 최대 1,023 개의 세션이 동시에 접속할 수 있다.

TABLE

- TABLE은 사용자 데이터를 저장하는 공간이다.
- Instance 내에서 테이블 이름은 고유해야 하며, 테이블 생성 개수에는 제한이 없다.

다음과 같은 유형의 테이블들을 지원한다. 각 테이블 유형은 생성 구문 (**create table**)을 통해 정의한다.

테이블 유형	설명
Normal table	B tree indexing segment 지원하는 테이블
Direct table	하나의 column value를 key로 사용하는 array 형태 테이블
Splay table	Splay indexing 을 사용하는 테이블
Queue table	FIFO (First-In, First-Out) 방식 테이블
Store table	Char data type만으로 구성된 key/ value 형식의 테이블
Sequence	채번을 위한 object

Column 타입으로 정의할 수 있는 데이터 유형은 다음과 같다.

Column type	설명
int	sizeof(int)
short	sizeof(short)
float	sizeof(float), 별도의 정밀도를 제공하지 않는다.
long	sizeof(long long)
char (size)	입력된 size 크기의 fixed 공간이다.
double	sizeof(double), 별도의 정밀도를 제공하지 않는다.
date	sizeof(unsigned long long)

노트

GOLDILOCKS LITE에는 실제 column 개념이 존재하지 않는다. Column 정의는 사용자 입력 데이터의 특정 위치에서 index key 값을 추출하거나, 관리 편의를 위해 column 형태로 출력하기 위한 용도로만 사용된다.

주의

Column의 offset과 size는 C struct의 default padding/packing 방식을 기반으로 설정된다. 따라서 #pragma pack 구문을 사용하여 default 설정과 다르게 지정할 경우, application이 정상적으로 동작하지 않는

다.

INDEX

Index는 테이블 내의 데이터를 효율적으로 검색하기 위한 object 이다.
(자세한 내용은 **create index** 를 참조한다.)

- Unique constraint 지원
- (float, double) 데이터 타입은 정밀도 오차가 발생할 수 있으므로 index key column으로 사용하는 것은 권장하지 않는다.
- 하나 이상의 composite key를 구성할 수 있다.
- Index key column의 정렬 방식을 지정할 수 있다. (ASC, DESC)
- Secondary index는 normal table에만 생성할 수 있다.

노트

- DIRECT, SPLAY, QUEUE, STORE 테이블에는 한 개의 INDEX만 생성할 수 있다.
- DIRECT table에는 한 개의 정수형 데이터 타입 column만 index로 지정할 수 있다.
- STORE, QUEUE table은 생성 시 자동으로 B-tree 기반 index가 생성되며, 사용자가 임의로 index를 변경할 수 없다.

DIRECT TABLE

Direct table은 테이블 내 숫자형 데이터 타입 column 중 하나를 index key로 지정하여, 해당 column의 데이터 값을 테이블 내 저장 위치로 사용한다. 예를 들어, 이 값이 1 인 경우 table에 데이터를 저장할 수 있는 공간 중 1 번 위치에 해당 데이터를 저장한다. (이는 array indexing 방식과 유사하며, 자세한 내용은 **create table** 을 참조한다.)

노트

Direct table은 create unique index 구문을 통해 반드시 하나의 column만 key column으로 지정해야 한다. Key column에는 long, short, int 등 정수형 data type만 지정할 수 있다.

SPLAY TABLE

Splay table은 splay indexing으로 정렬되는 테이블이다. 이전에 조회하거나 조작한 데이터와 인접한 데이터를 빠르게 탐색해야 하는 경우에 적합하다. (자세한 내용은 **create table** 을 참조한다.)

> 주의
> Splay table의 성능은 데이터 조작 방식 및 접근 패턴에 따라 달라질 수 있다.

STORE TABLE

Store table은 key에 대응하는 value를 저장하는 테이블이다.
테이블의 column은 별도로 정의하지 않고 key와 value의 저장 크기만 지정한다.
SET 기능을 이용하여 데이터를 저장하고, GET 기능을 이용하여 조회한다.
Key의 최대 길이는 64 byte이며, value의 최대 크기는 512 K 이다. (해당 크기는 fixed size로 동작한다. 자세한 내용은 **create store** 를 참조한다.)

> 주의
> Store table은 자동으로 B-tree index를 생성하며, 사용자가 임의로 index를 변경할 수 없다.

QUEUE

Queue는 First-In/First-Out (FIFO) 방식으로 사용할 수 있는 테이블이다.
사용자가 입력한 데이터를 순서대로 저장하고 조회하는 구조의 테이블이다.
(자세한 내용은 **create queue**를 참조한다.)

Queue는 처리 순서를 제어하기 위한 priority 기능을 제공하며, priority 값이 낮을수록 우선 순위가 높다.

Dequeue 동작 결과는 수행 시점에 따라 차이가 있을 수 있다.
예를 들어, 데이터가 (1, 2, 3)과 같은 순서로 저장되어 있을 경우, dequeue 수행 시점에 따라 다음 표와 같은 결과가 나올 수 있다.

Time	Deq Process #1	#2	#3
Time-1	"1"	"2"	
Time-2		rollback	
Time-3			"2"

각 데이터는 enqueue 시점에 고유한 MsgID가 부여되어 저장된다.
만약 dequeue를 수행한 세션이 rollback 되더라도 해당 MsgID는 변경되지 않는다. 이로 인해 rollback이 선행된 경우, 이후 dequeue를 수행하는 세션이 동일한 데이터를 다시 가져올 수 있다.

노트

데이터의 처리 순서가 반드시 보장되어야 하는 경우에는, enqueue와 dequeue 세션을 1:1 구조로 구성해야 한다.

SEQUENCE

Sequence는 고유 번호를 획득하기 위해 사용하는 object이다. (자세한 내용은 **create sequence** 를 참조한다.)
Sequence 객체를 생성한 후에는 반드시 nextval/ dbmGetNextVal을 호출해야 한다.

동시성 및 복구

본 절에서는 별도의 관리 프로세스가 없는 GOLDILOCKS LITE에서 세션 간 동시성을 제어하거나 복구하는 방법에 대해 설명한다.

Row Level Lock

테이블에 저장되는 데이터가 동시에 변경되는 것을 방지하기 위해 row level의 lock을 제공한다.
사용자가 호출한 변경 연산은 내부 처리 과정에서 자동으로 필요한 lock을 획득하며 commit/rollback을 통해 해제한다.

Read Committed

세션은 갱신 작업을 수행하기 전에 갱신할 image를 별도의 공간 (instance undo space)에 저장한다. 이 때 접근하는 조회 세션은 갱신 트랜잭션을 기다리지 않고 이전에 commit 된 image를 읽을 수 있도록 동시성을 제공한다.

노트

해당 동작은 DBM_MVCC_ENABLE property (**환경 변수 및 프로퍼티**) 에 의해 제어된다.
사용자가 데이터를 갱신하는 동안 조회 시도가 대기하도록 하려면, 해당 속성을 FALSE로 설정한다.
(기본값은 TRUE 이다.)

Auto Dead Lock Detection

갱신 연산 간에 상호 대기가 발생할 수 있다.
예를 들어, T1 테이블에 (A, B) 레코드가 존재할 때 세션 #1이 A를 갱신한 후 B를 갱신하려고 하고, 동시에 세션 #2

가 B를 먼저 갱신한 후 A를 갱신하려고 하면, 세션 #1과 세션 #2는 교착 상태에 빠진다.
GOLDILOCKS LITE 라이브러리는 이러한 상황을 자동으로 감지하며, 세션 ID가 높은 세션에 오류를 발생시켜 교착 상태를 해소한다.

> 주의
> Application이 교착 상태 오류에 빠진 경우, 이전 트랜잭션이 자동으로 rollback되지 않는다.
> 따라서 사용자가 명시적으로 트랜잭션을 commit 또는 rollback 해야 교착 상태가 해소된다.

Delayed Recovery Concept

Delayed recovery는 GOLDILOCKS LITE에서 비정상적으로 종료된 트랜잭션을 application이 감지하고 복구할 수 있도록 지원하는 기능이다.
GOLDILOCKS LITE에는 세션이나 트랜잭션을 관리하는 별도의 프로세스가 없으므로, lock을 점유하려는 세션이 직전에 접근했던 세션의 비정상 종료 여부를 직접 감지하고 필요한 복구 작업을 수행한다. 이러한 방식의 복구를 Delayed Recovery 라고 한다.

GOLDILOCKS LITE에서 트랜잭션은 다음과 같은 방식으로 처리된다.
- 사용자 application은 instance segment에 자신의 정보와 트랜잭션 정보를 기록한다.
- Lock을 점유할 경우, 점유한 자원 (레코드)에 해당 세션의 트랜잭션 정보를 기록한다.

Delayed recovery는 다음과 같은 방식으로 감지된다.
- Lock 대상에 기록된 정보를 기준으로 해당 트랜잭션이 정상적으로 종료되었는지 확인한다.
- 현재 lock을 점유하고 있는 세션의 유효성을 판단한다.

Delayed recovery가 필요하다고 판단한 세션은 다음 작업을 수행한다.
- 비정상 종료된 세션에 대한 lock을 획득한다. (동시 복구 방지)
- 비정상 종료된 세션이 기록한 transaction log를 기반으로 데이터 복구 및 자원 해제를 진행한다.
- 유효하지 않은 세션이 사용한 instance 영역을 해제한다.
- 복구 완료 후 자신의 트랜잭션 수행한다.

> 노트
> 복구가 불가능한 상황, 즉 critical section 구간에 진입한 상태에서 kill -9 등의 비정상 종료가 발생한 경우에는 메모리 상의 데이터가 손상되었을 가능성이 있다. 이러한 경우 시스템은 사용자에게 오류 메시지를 반환하며, 대상 테이블을 직접 복구할 수 있는 방법을 제공한다.
> 자세한 내용은 **alter system refine [TableList]**를 참조한다.

주의

- 최신 3.3 버전의 B-tree index는 node lock 구조를 사용한다. 이전 트랜잭션이 특정 노드의 lock을 점유한 상태에서 종료되더라도, 해당 노드와 무관한 트랜잭션은 인덱스 작업을 정상적으로 수행한다. 이로 인해 지연 감지 (delayed detection)가 발생할 수 있으며, 이러한 경우에는 인덱스를 재구축해야 한다.
- Delayed recovery 과정에서 object에 DDL이 수행된 경우에는 다음과 같이 처리된다.
  - Table object가 재생성된 경우에는 트랜잭션 로그를 복구하지 않고 drop 한다.
  - Table object가 유효한 경우에는 직전 트랜잭션의 상태에 따라 commit 또는 rollback 을 수행한다.

Disk Logging

GOLDILOCKS LITE는 기본적으로 instance 영역에서 in-memory logging만 수행한다.
보다 높은 안정성이 필요한 경우에는 디스크 로깅 기능을 사용할 수 있으며, 동작 가능한 방식은 다음 두 가지로 구분된다.

• Non cache mode: 각 세션이 자신의 전용 로그 파일에 직접 기록하는 방식이다.
• Log cache mode: 여러 세션이 공유 메모리 영역 (log cache)에 로그를 순차적으로 적재하면, 별도 프로세스 (dbmLogFlusher)가 이를 디스크 로그 파일로 플러시하는 방식이다.

노트

- 디스크 로깅을 사용하면 OS fatal 이나 memory H/W 장애와 같은 상황에 대비할 수 있다.
  (자세한 내용은 **복구 가이드**를 참조한다.)
- 단, 디스크 로깅은 in-memory logging에 비해 성능 저하가 발생할 수 있으므로 이를 고려하여 사용해야 한다.

Replication

GOLDILOCKS LITE는 network replication 방식을 통해 데이터를 동기화할 수 있다.
이중화 환경에서는 active 노드에서 트랜잭션이 커밋되는 시점에 해당 변경 내용을 standby 노드로 전송한다.
복제 성능 향상을 위해 서로 간섭하지 않는 트랜잭션은 병렬로 처리될 수 있다.
모든 복제 작업은 트랜잭션 단위로 수행되며, standby 노드에서도 커밋된 트랜잭션 단위로 적용된다.

제공되는 network replication 방식의 특징은 다음과 같다.

- Master-slave 기반의 단방향 1:1 구조를 사용한다.
- Commit 시점에 application이 트랜잭션 로그를 전달하는 방식으로 동작한다.

- SYNC/모드만을 지원하며 application이 commit을 수행할 때 slave에 로그 반영이 완료된 후 commit 이 완료된다.
- Slave 측에서는 해당 로그를 수신하고 반영하기 위해 dbmReplica process를 구동해야 한다.
- 테이블 단위의 이중화를 지원한다. (자세한 내용은 **create replication** 을 참조한다.)
- 이중화 대상 테이블에는 unique index가 반드시 존재해야 한다.
- 네트워크 장애가 발생하면 발생 시점의 Tx는 실패하며 이중화는 더 이상 진행되지 않는다.
- Replication conflict가 발생하면 System Commit Number (SCN)를 기준으로, 더 높은 SCN 값을 가진 데이터를 우선하여 정합성을 유지한다.
- GODILOCKS LITE의 replication은 transaction 단위의 정합성만 보장한다.

Replication 환경에서는 동기화 과정에서 데이터 간 불일치를 해소할 수 없는 상황이 발생할 수 있으며, 이를 replication conflict 상태로 정의한다. GOLDILOCKS LITE는 이러한 conflict를 다음과 같은 방식으로 처리한다.

Replication conflict 유형	Slave 처리 방식
Insert conflict • Duplicated record	Slave의 데이터가 정상인 경우, conflict 오류만 기록한다.
Update conflict • Not found	• 해당 데이터가 존재하면 SCN이 더 높은 데이터를 기준으로 정합성을 맞춘다. • 존재하지 않으면 현재 로그를 기반으로 데이터를 삽입한다.
Delete conflict • Not found	Slave에서 데이터를 찾을 수 없는 경우 conflict 오류만 기록한다.

주의

- Replication 환경에서 active/ standby의 object 객체는 동일한 createSCN 을 가져야 한다.
- Replication 환경에서는 conflict 발생 가능성을 고려하여 slave의 데이터를 직접 조회하거나 수정하지 않는 것을 권장한다.
- 네트워크 장애가 발생할 경우 이중화는 중단된다. 장애 발생 시점을 포함하여 진행 중인 트랜잭션은 local (primary)에만 반영되며 장애 시점의 커밋중인 트랜잭션은 네트워크 관련 오류가 리턴될 수 있으나 커밋에는 성공한다.

노트

- 이중화 대상 테이블은 추가 및 삭제가 가능하다. 그러나 이러한 변경 사항은 application 에서 자동으로 감지할 수 없으므로, 이중화 관련 DDL을 수행한 경우에는 관련 application 을 재기동해야 한다.

1.2 Quick Start

GOLDILOCKS LITE의 설치 및 기본 사용법에 대해 설명한다.

설치 전 작업

/dev/shm 공간 설정

Posix 방식의 shared memory를 사용하므로 /dev/shm에 충분한 공간을 확보해야 한다.

커널 파라미터 설정

일부 Linux 커널 버전에서는 IPC 자원이 자동으로 삭제될 수 있으므로, 이를 방지하기 위해 "RemoveIPC" 설정을 적용해야 한다.

```
# cp -i /etc/systemd/logind.conf /etc/systemd/logind.conf_prev
# cat /etc/systemd/logind.conf
[Login]
...
RemoveIPC=no
...
```

환경 변수 및 프로퍼티

GOLDILOCKS LITE를 사용하려면 사용자가 $DBM_HOME/conf/dbm.cfg 또는 사용자 환경 변수를 설정해야 한다. (각 속성의 의미는 아래 표에 설명되어 있으며, 환경 변수가 설정 파일보다 우선 적용된다.)

환경 변수	설명
DBM_HOME	GOLDILOCKS LITE가 설치된 경로이다.
DBM_INSTANCE	사용할 Default Instance Name을 지정한다.
PATH	실행 파일을 사용할 수 있도록 사용자 환경 변수 PATH에 $DBM_HOME/bin 을 추가한다.
LD_LIBRARY_PATH	실행 파일이 참조하는 shared library를 탐색할 수 있도록 사용자 환경 변수 LD_LIBRARY_PATH에 $DBM_HOME/lib 를 추가한다.
DBM_SHM_PREFIX	/dev/shm에 shared memory segment를 생성할 때 파일명의 접두어로 사용된다.
DBM_SHM_DIR	kernel 버전에 따라 /dev/shm 하위 디렉토리 생성이 허용될 경우에 사용한다. (default(0): 사용하지 않음, (1): DBM_SHM_PREFIX 이름으로 디렉토리 생성)
	프로세스를 바인딩할 NUMA NODE ID를 설정한다. 지정하지 않는 경우 OS 정책에 따른다

환경 변수	설명
DBM_NUMA_BIND	예: DBM_NUMA_BIND=1 (단일 노드 지정) DBM_NUMA_BIND=0,1 (여러 노드 지정) DBM_NUMA_BIND=0-2 (범위 지정) DBM_NUMA_BIND=0,1-2 (노드와 범위 혼합 지정)
DBM_NUMA_POLICY	프로세스를 바인딩 할 메모리 할당 정책을 설정한다. • 설정하지 않을 경우: NUMA 노드가 1 개인 경우 BIND 로, 그 외에는 INTERLEAVE 로 설정된다. • BIND: DBM_NUMA_BIND에 지정된 ID로 바인딩 되며, 메모리 부족 시 할당에 실패한다. • INTERLEAVE: DBM_NUMA_BIND에 지정된 노드로 메모리를 분산하여 할당한다. • LOCAL: 현재 실행 중인 CPU에 속한 노드에 메모리를 할당한다. • PREFERRED: DBM_NUMA_BIND에 지정된 첫 번째 노드에 우선 할당하고, 메모리가 부족할 경우 INTERLEAVE 방식이 적용된다.

프로퍼티 이름	설명	옵션	Default 값	create instance 후 변경 가능 여부
DBM_DISK_LOG_ENABLE	DISK mode 사용 여부를 설정한다.	[0 | FALSE] = disable [1 | TRUE] = enable	0 /FALSE	X
DBM_DISK_LOG_DIR	DISK mode를 사용할 경우, 트랜잭션 로그 파일이 저장되는 경로이다.	-	${DBM_HOME}/wal	X
DBM_DISK_LOG_FILE_SIZE	트랜잭션 로그 파일의 크기를 지정한다. 지정된 크기를 초과하면 자동으로 다음 로그 파일이 생성되어, 로그가 계속 기록된다.	예: 1024M(1024megabyte) 1G (1gigabyte)	100M	X
DBM_DISK_DATA_FILE_DIR	DISK mode에서 CheckPoint 수행 시, 데이터 파일이 저장되는 경로이다.	-	${DBM_HOME}/dbf	X
DBM_DIRECT_IO_ENABLE	DIRECT I/O 사용 여부를 설정한다.	[0 | FALSE] = disable [1 | TRUE]= enable	[0 | FALSE]	X
DBM_DIRECT_IO_SIZE	DIRECT I/O를 사용하기 위한 disk sector size를 설정한다.	예: 512(byte)	512	X
DBM_LISTEN_PORT	dbmListener를 통한 원격 접속 시 사용할 port를 지정한다.	예: 27584	27584	O
DBM_LISTEN_CONN_TIMEOUT	dbmListener와 연결을 시도할 때 적용되는 timeout 시간이다.	예: 10000(ms)	10000	O
DBM_LISTEN_RECV_TIMEOUT	dbmListener에 요청 전송 후, 응답을 기다리는 시간을 설정한다.	예: 10000(ms)	10000	O

프로퍼티 이름	설명	옵션	Default 값	create instance 후 변경 가능 여부
DBM_LOG_CACHE_MODE	Log cache 모드를 설정한다.	• 0: 사용 안함 • 1: NVDIMM 사용 • 2: Shared memory 사용	0	X
DBM_LOG_CACHE_SIZE	Log cache 크기를 지정한다.	예: 1G	1G	X
DBM_LOG_CACHE_FLUSH_INTERVAL	dbmLogFlusher의 동작 주기이다.	예: 3000(ms)	3000	O
DBM_DISK_COMMIT_WAIT	Commit 시, redo 파일이 완전히 기록될 때까지 대기할지 여부를 설정한다. (데이터 안전성은 높아지지만 성능이 저하될 수 있다.)	[0 | FALSE] = disable [1 | TRUE]= enable	[0 | FALSE]	O
DBM_ARCHIVE_ENABLE	CheckPoint 이후의 트랜잭션 로그 파일을 archive로 저장할지 여부를 설정한다.	[0 | FALSE] = disable [1 | TRUE] = enable	[0 | FALSE]	X
DBM_ARCHIVE_PATH	Archive 로그 파일이 저장되는 디렉토리 경로이다.	-	${DBM_HOME}/arch	X
DBM_REPL_ENABLE	이중화 기능 사용 여부를 설정한다.	[0 | FALSE] = disable [1 | TRUE] = enable	[0 | FALSE]	X
DBM_REPL_TARGET_PRIMARY_IP	dbmReplica가 동작하는 node의 IP를 설정한다.	-	127.0.0.1	O
DBM_REPL_TARGET_PORT	dbmReplica의 listen port를 설정한다. (어플리케이션이 참조하는 값)	-	27584	O
DBM_REPL_LISTEN_PORT	dbmReplica의 listen port를 설정한다. (dbmReplica가 참조하는 값)	-	27584	O
DBM_REPL_CONN_TIMEOUT	이중화 연결을 시도할 때의 timeout 값을 지정한다.	예: 10000 (ms)	10000	O
DBM_REPL_RECV_TIMEOUT	이중화 수행 중 응답 수신을 기다리는 시간이다.	예: 10000 (ms)	10000	O
DBM_NO_INDEX_ERROR_AT_PREPARE	dbmPrepareTable 또는 dbmPrepareTableHandle를 호출할 때, 대상 테이블에 index가 없을 경우 오류를 반환할지 여부를 설정한다.	[0 | FALSE] = disable [1 | TRUE] = enable	[1 | TRUE]	O
DBM_LOADING_THREAD_COUNT	Startup 또는 recovery 시 데이터를 로딩하는 thread 개수를 지정한다.	1 이상의 수	8	O

프로퍼티 이름	설명	옵션	Default 값	create instance 후 변경 가능 여부
DBM_PERF_ENABLE	Session의 acitivity를 count 할지 여부를 설정한다. • DML/DCL 횟수 • Index operation 횟수 • Lock retry 횟수 • Delayed recovery 횟수 관련 내용은 **V$SESS_STAT**에서 조회 가능	[0 | FALSE] = disable [1 | TRUE] = enable	[0 | FALSE]	O
DBM_TRACE_LOG_FOR_PBT	상세한 trace log가 필요한 경우에 설정한다. (활성화 시 성능이 저하될 수 있다.)	[0 | FALSE] = disable [1 | TRUE] = enable	[0 | FALSE]	O
DBM_MVCC_ENABLE	조회 연산 시 lock 대기 여부를 설정한다. 변경 중인 레코드에 대한 조회 작업이 수행되는 경우, 이 설정에 따라 직전에 commit 된 데이터를 조회할 수 있다.	[0 | FALSE] = disable [1 | TRUE] = enable	[1 | TRUE]	O
DBM_LOCK_INTERVAL	Lock을 획득하는 과정에서 대기할 시간 (us단위)을 설정한다.	0 이상의 값일 때 적용	10(us)	O
DBM_LOCK_TIMEOUT	지정된 시간 (us단위) 내에 lock을 획득하지 못하면 에러를 발생시킨다.	0 이상의 값일 때 적용	3000000 (3초)	O
DBM_BLOCK_SIGNAL	Critical section 구간에서 signal에 의한 작업 중단을 방지하기 위해 block 할 signal을 지정한다.	SIGUSR1, SIGUSR2, SIGINT, SIGTERM 만 지정 가능	설정하지 않음	X

다음은 bash 환경에서 환경 변수를 설정하는 예이다.

- GOLDILOCKS LITE 기본 설정

```
export DBM_HOME=/home/lim272/dbm_home      ## 설치 경로
export PATH=${DBM_HOME}/bin:$PATH:.
export LD_LIBRARY_PATH=${DBM_HOME}/lib:$LD_LIBRARY_PATH:.
```

- 프로퍼티를 환경 변수로 설정

```
export DBM_DISK_LOG_ENABLE=1
export DBM_DISK_LOG_DIR=${DBM_HOME}/wal
export DBM_DISK_DATA_FILE_DIR=${DBM_HOME}/dbf
```

설치 및 라이센스

본 절에서는 GOLDILOCKS LITE의 설치 방법과 라이센스 발급 절차를 설명한다.

설치

GOLDILOCKS LITE는 다음과 같은 형식의 압축 파일로 제공된다.
패키지 파일명에서 각 항목이 의미하는 바는 아래와 같다.

```
goldilocks_lite-3.2.rev6762-linux4.18.0-305.3.1.el8.x86_64.nopmem.noflt128-debug.tar.gz
* goldilocks_lite-3.2 : Product Major Version
* rev???? : Patch Version
* linux4.18.0.-305.3.1.el8 : Linux version (el8 호환)
* X86_64 : CPU / 64 bit
* nopmem : NVDIMM 지원 여부
* noflt128 : float128 지원 여부
* debug : debug / release 여부
```

다음 명령어를 통해 설치 파일의 압축을 해제한다.

```
shell> tar -xzf goldilocks_3.1.1.tar.gz
```

압축을 해제하면 아래와 같은 경로가 생성되며, 각 경로는 다음과 같은 용도로 사용된다.

경로 이름	설명
arch	Archive log가 저장되는 기본 경로이다.
bin	각종 유틸리티 바이너리가 위치한 경로이다.
lib	API shared library가 위치한 경로이다.
include	API header 파일이 위치한 경로이다.
trc	Trace log가 저장되는 경로이다.
sample	API 활용 예제를 포함한 sample code 이다.
conf	dbm.cfg와 dbm.license 파일이 위치한 경로이다.
dbf	Disk mode로 설정 시, datafile이 저장되는 기본 경로이다.
repl	reserved
wal	Disk mode 설정 시, redo logfile이 저장되는 기본 경로이다.

bin 디렉토리의 각 binary는 다음과 같은 기능을 수행한다.

Binary 이름	설명
dbmMetaManager	SQL/Internal command를 실행할 수 있는 interactive 유틸리티 이다.
dbmListener	원격 서버에서 접속/처리를 위해 구동해야 하는 유틸리티 이다.

Binary 이름	설명
dbmLogFlusher	Log cache에 저장된 트랜잭션 로그를 디스크로 저장하는 유틸리티 이다.
dbmMonitor	현재 DB 상태를 간략하게 모니터링 하 유틸리티 이다.
dbmExp	Object 생성 script 및 데이터를 추출하는 유틸리티 이다.
dbmImp	구분자 (delimiter) 를 가진 파일의 데이터를 데이터베이스에 적재하는 유틸리티 이다.
dbmErrorMsg	전체 error code를 출력하는 유틸리티 이다.
dbmCkpt	Redo logfile을 이용하여 데이터 파일을 생성하는 유틸리티 이다.
dbmReplica	Replication 환경의 slave에서 구동되어 데이터를 수신하고 반영하는 유틸리티이다.
dbmDump	메모리 세그먼트와 파일을 추적하는 유틸리티 이다. 트랜잭션, 테이블, 인덱스, 데이터 파일, 로그 파일, anchor 파일 등의 정보를 확인할 수 있다.

라이센스

설치할 장비의 CPU core 개수와 hostname을 첨부하여, technet@sunjesoft.com 으로 라이센스를 요청한다.

시작하기

GOLDILOCKS LITE를 처음 시작하려면, dbmMetaManager를 실행한 후 initdb 명령을 수행하여 Dictionary Instance를 생성해야 한다.
(자세한 내용은 **DICTIONARY** 항목을 참조한다.)

```
[lim272@localhost 4th_iter]$ dbmMetaManager
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(unknown)> initdb;
success
```

노트

- Dictionary instance의 이름은 "dict"로 고정되어 있으며 변경할 수 없다.
- Dictionary instance에서는 사용자 객체 (user object)를 생성할 수 없다.

사용자 instance는 dictionary instance에 접속한 상태에서만 생성할 수 있다.

```
dbmMetaManager(unknown)> set instance dict;
success
dbmMetaManager(DICT)> create instance demo;
success
dbmMetaManager(DICT)> set instance demo;
success
dbmMetaManager(DEMO)>
```

다음과 같이 사용자 instance에서 table과 index를 생성하여 사용할 수 있다.

```
dbmMetaManager(DEMO)> create table t1 (c1 int, c2 int);
success
dbmMetaManager(DEMO)> create unique index idx_t1 on t1 (c1);
success
dbmMetaManager(DEMO)> insert into t1 values (1, 1);
success
dbmMetaManager(DEMO)> commit;
success
dbmMetaManager(DEMO)> update t1 set c2 = 100 where c1 = 1;
1 row updated.
dbmMetaManager(DEMO)> select * from t1;
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 100
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> delete from t1;
1 row deleted.
dbmMetaManager(DEMO)> select * from t1;
--------------------------------------------------------------------------------
0 row selected
dbmMetaManager(DEMO)> rollback;
success
dbmMetaManager(DEMO)> select * from t1;
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 1
--------------------------------------------------------------------------------
1 row selected
```

dbmMetaManager 실행 오류 시 참고 사항

환경 변수 "PATH" 에 설치 경로 "${DBM_HOME}/bin" 이 추가되지 않은 경우, 다음과 같은 오류가 발생한다.

```
$ dbmMetaManager
bash: dbmMetaManager: command not found...
```

환경 변수 "LD_LIBRARY_PATH"에 설치 경로 ${DBM_HOME}/lib 가 추가되지 않은 경우, 다음과 같이 shared library 로딩 오류가 발생한다.

```
$ dbmMetaManager
dbmMetaManager: error while loading shared libraries: libdbmCore.so: cannot open shared object file: No such file or directory
```

${DBM_HOME}/conf 내에 dbm.license 파일이 없거나 손상된 경우, 다음과 같이 라이센스 확인 오류가 발생한다.

```
$ dbmMetaManager
ERR] failed to check a license
ERR-21040] No such object (/mnt/md1/new_lite/pkg/conf/dbm.license): dbfOpen() returned errno (2)
$ dbmMetaManager
ERR] failed to check a license
ERR-22098] invalid license
```

${DBM_HOME}/conf 내에 dbm.cfg 파일이 존재하지 않는 경우, initdb 실행 시 오류가 발생한다.

```
dbmMetaManager(unknown)> initdb;
Command] <initdb>
ERR-21040] No such object (/mnt/md1/ssd_home/lim272/new_lite/pkg/conf/dbm.cfg): dbfOpen() returned errno(2)
```

1.3 구문

GOLDILOCKS LITE에서 사용 가능한 SQL syntax에 대해 설명한다.

Data Definition Language (DDL)

DDL은 instance, table, index 등과 같은 다양한 object 를 생성하거나 제거하는 데 사용되는 구문이다.
DDL에서 사용되는 각 object의 이름은 최대 64 byte까지 지정할 수 있으며, 반드시 문자로 시작해야 한다.

Table을 생성할 때 record 하나의 최대 크기는 1,048,247 byte이다.
Queue를 생성할 때 message 하나의 최대 크기는 1,048,208 byte이다.
Index를 생성할 때 key column 들의 전체 길이 합은 1,024 byte 를 초과할 수 없다.

주의

- DDL 간에는 instance lock으로 동시성을 보장한다.
- Commit 또는 rollback 되지 않은 트랜잭션이 존재할 경우, DDL은 수행되지 않는다.
- DDL과 DML 간에는 동시성이 보장되지 않는다.
  - DDL은 데이터가 변경되지 않는 상황에서 사용하는 것을 권장한다.
  - 운영 중에 DDL이 수행되면 기존에 attach 된 프로세스가 정상적으로 동작하지 않을 수 있다.

initdb

기능

GOLDILOCKS LITE를 처음 사용할 때 dictionary instance를 생성하는 명령이다.
생성된 dictionary instance의 이름은 "DICT" 이며, set instance 구문으로 접근할 수 있다.

구문

```
<initdb> ::= initdb
             ;
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
```

```
************************************************************
dbmMetaManager(unknown)> initdb;
success
dbmMetaManager(unknown)> set instance dict;
success
dbmMetaManager(DICT)>
```

노트

- 이 명령으로 생성되는 object에 대한 자세한 설명은 **DICTIONARY**를 참조한다.
- Dictionary instance에서는 DICT_INST 테이블만 유효한 정보를 포함하며, 그 외 object 들은 향후 dep recated 될 예정이다.
- 이 명령은 dbmMetaManager에서만 사용할 수 있다.

initdb를 수행하면 /dev/shm 경로에 아래 예시와 같은 segment file이 생성된다.
예: /dev/shm/lim272_DICT_DICT_000, /dev/shm/lim272_DICT_DIC_TABLE_000

- /dev/shm 에 생성되는 segment 파일명은 다음 규칙을 따른다.
  - <Prefix>_<Instance Name>_<Segment Name>_<생성번호>
- 위 예시의 각 구성 요소는 다음과 같은 의미를 갖는다.
  - lim272: 다중 사용자 환경에서 동일한 이름의 instance를 구분하기 위해 사용자 환경 변수 DBM_SHM_PR EFIX 에 설정된 값이다.
  - DICT: Segment가 속한 instance 의 이름이다.
  - DICT, DIC_TABLE: Instance 내에서 생성된 object의 이름이다.
  - 000: Segment의 생성 순서를 나타내는 번호이다.

노트

/dev/shm에 기존 파일이 존재할 경우 "initdb" 명령 실행 시 아래와 같은 오류가 발생할 수 있으며, 이 경우 사용자는 위의 파일명 규칙을 참고하여 관련 파일을 모두 삭제한 후에 다시 실행해야 한다.

```
dbmMetaManager(unknown)> initdb;
Command] <initdb>
ERR-22005] a shared memory already exists(DICT_DICT_000): dbmCreatePosixShm() returned errno
(17)
```

create instance

기능

사용자 instance를 생성하는 명령으로, 생성 후에는 테이블 등 다양한 object를 생성할 수 있다.
사용자 instance는 Dictionary instance (DICT)에 접속한 상태에서만 생성하거나 제거할 수 있다.

노트

- 사용자 instance 공간은 세션 정보 기록 및 트랜잭션 정보 저장에 사용된다.
- 하나의 사용자 instance에서 동시에 접속 가능한 세션 최대 개수는 1,023 개이다.

구문

```
<create instance> ::= CREATE INSTANCE instance_name
                      [ init <size> ]
                      [ extend <size> ]
                      [ max <size> ]
                      ;
```

- instance_name: 사용자가 지정하는 instance 이름이다.
- init <size>: Instance 생성 시 최초로 할당할 undo space의 크기를 지정한다. (각 size 단위는 1M 이다.)
- extend <size>: init 된 공간이 모두 사용될 경우 확장되는 크기이다.
- max <size>: Undo space가 확장될 수 있는 최대 크기이다.

노트

Instance 공간은 사용자가 수행하는 트랜잭션 이력을 저장하며, 변경 연산을 수행할 때 롤백을 위한 undo image가 함께 기록된다. 공간이 부족할 경우 자동으로 세션 내에서 확장되며, 사용자가 지정한 MAX 값까지 확장할 수 있다.

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(unknown)> set instance dict;
```

```
success
dbmMetaManager(DICT)> create instance demo init 1024 extend 1024 max 10240;
success
dbmMetaManager(DICT)> select * from DIC_INST;
--------------------------------------------------------------------------------
INST_NAME   : DICT
INIT_SIZE   : 128
EXTEND_SIZE : 128
MAX_SIZE    : 1048576
--------------------------------------------------------------------------------
INST_NAME   : DEMO
INIT_SIZE   : 1024
EXTEND_SIZE : 1024
MAX_SIZE    : 10240
--------------------------------------------------------------------------------
2 row selected
```

Instance가 생성되면, DBM_DISK_DATA_FILE_DIR 에 지정된 경로에 dictionary table 용 데이터 파일이 생성된다. 생성되는 파일과 용도는 다음 표와 같다.

데이터 파일 유형	설명
DIC_TABLE.dbf	테이블 형상 정보를 저장한다.
DIC_COLUMN.dbf	테이블의 column 구성 정보를 저장한다.
DIC_INDEX.dbf	테이블에 생성된 인덱스 형상 정보를 저장한다.
DIC_INDEX_COLUMN.dbf	인덱스의 key column 구성 정보를 저장한다.

Instance 내에서 테이블 생성/ 삭제/ 변경이 발생하면 메모리뿐 아니라 dictionary 데이터 파일에도 내용이 반영된다. Dictionary 데이터 파일은 디스크 모드에서 "startup" 복구를 수행할 때 복구 대상 테이블 목록을 구성하는 데 반드시 필요하므로, 유실되지 않도록 주의해야 한다.

노트

Disk mode로 설정한 경우, CREATE INSTANCE 시점에 DBM_DISK_LOG_DIR 에 설정된 경로에 <instance_name.anchor> 파일과 redo logfile이 생성되기 시작한다. Anchor file은 redo logfile의 메타 정보를 관리하므로 손상되지 않도록 주의해야 한다. (**디스크 로깅을 이용한 복원**을 참조한다.)

create table

기능

사용자 테이블을 생성한다. 테이블은 instance의 하위 개념이다.

구문

```
<create table> ::= CREATE [TABLE_TYPE] table_name
                    (
                       column_definition [, ...]
                    )
                    [ init <size> ]
                    [ extend <size> ]
                    [ max <size> ]
                    ;
<TABLE_TYPE> ::= TABLE
                ¦ DIRECT
                ¦ SPLAY
<table_name> :: 사용자가 지정하는 테이블 이름이다.
<column_definition> ::= column_name data_type_definition
<column_name> :: 사용자가 지정하는 column 이름이다.
<data_type_definition> ::= short
                         ¦ int
                         ¦ long
                         ¦ float
                         ¦ double
                         ¦ char (size)
                         ¦ date
init <size> :: 첫 번째 segment에 저장할 수 있는 row의 개수를 지정한다. (Default: 1,024개)
extend <size> :: init segment 공간이 모두 사용되어 확장되는 segment에 저장할 수 있는 row의
개수를 지정한다. (Default: 102,400 개)
max <size> :: 최대로 저장할 수 있는 row 개수를 지정한다. (Default: 4,096,000 개)
```

GOLDILOCKS LITE에서 지원하는 data type과 각 type의 크기는 아래와 같으며, 사용자 프로그램에서 변수를 선언할 때 참고한다.

Data type	C type과 크기 (64 bit OS 기준)
short	short 타입이며 2 byte 이다.
int	int 타입이며 4 byte 이다.
long	long 타입이며 8 byte 이다.
float	float 타입이며 4 byte 이다.

Data type	C type과 크기 (64 bit OS 기준)
double	double 타입이며 8 byte 이다.
char	char 타입이며 fixed size이다.
date	8 byte이며, 사용자 구조체의 멤버 변수는 8 byte 크기로 선언해야 한다.

노트

- 각 column의 offset은 c 언어 struct의 default padding/packing 규칙을 따른다.
- Column 정의는 최대 1024 개로 제한된다.

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DICT)> set instance demo;
success
dbmMetaManager(DEMO)> create table t1
                 2 (
                 3   c1 short,
                 4   c2 int,
                 5   c3 long,
                 6   c4 float,
                 7   c5 double,
                 8   c6 char(20),
                 9   c7 date
                10 );
success
dbmMetaManager(DEMO)> desc t1;
-------------------------------------------------------------------------------
Instance=(DEMO) Table=(T1) Type=(TABLE) RowSize=(64)
-------------------------------------------------------------------------------
C1                              short               2          0
C2                              int                 4          4
C3                              long                8          8
C4                              float               4          16
C5                              double              8          24
C6                              char                20         32
```

```
C7                              date                 8         56
-------------------------------------------------------------------------------
Any index not created
-------------------------------------------------------------------------------
success
dbmMetaManager(DEMO)> select * from dic_table where table_name = 'T1';
-------------------------------------------------------------------------------
INST_NAME               : DEMO
TABLE_NAME              : T1
ID                      : 25
TABLE_TYPE              : 1
COLUMN_COUNT            : 7
ROW_SIZE                : 64
LOCK_MODE               : 1
MSG_SIZE                : 0
INDEX_COUNT             : 0
INIT_SIZE               : 1024
EXTEND_SIZE             : 102400
MAX_SIZE                : 4096000
INDEX_ID                : 0
ONLY_UPDATE_SELECT_MODE : 0
CREATE_SCN              : 22
-------------------------------------------------------------------------------
1 row selected
```

노트

- 테이블 형상 정보에서 (column 이름, 데이터 타입) 이후에 표시되는 항목은 각각 해당 column의 크기 (byte 단위)와 레코드 내 시작 위치를 의미한다.
- 하나의 테이블에서 생성할 수 있는 확장 세그먼트의 최대 개수는 999개이다. 따라서 사용자는 테이블 생성 시 init, extend, max 값을 적절히 지정해야 한다.

create index

기능

테이블에 속한 index를 생성한다.

- Index 생성 개수에는 제한이 없으나, 인덱스 수가 증가할수록 삽입 성능에 영향을 줄 수 있다.
- Index key column 크기의 합은 1,024 byte를 초과할 수 없다.
- Index key column으로는 int, short, long, char 네 가지 데이터 타입만 허용된다.

• Direct 테이블과 splay 테이블에서 index key column을 지정할 때 이 구문을 사용한다.

구문

```
<create index> ::= CREATE [UNIQUE] INDEX index_name ON table_name
                   ( column_name <ordering> [, ... ] )
                   ;
<UNIQUE> :: 지정하면 동일한 key 값의 저장을 허용하지 않는다. (단, NULL 값의 저장은 허용된다.)
<index_name> :: 사용자가 지정하는 index 이름이다.
<table_name> :: Index를 생성할 대상 table 이름이다.
<column_name> :: Key column으로 사용할 column 이름이다.
<ordering : [ASC ¦ DESC ]> :: (정렬 순서를 지정한다. 지정하지 않는 경우 기본값은 ASC 이다.)
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> create unique index idx1_t1 on t1 (c1);
success
dbmMetaManager(DEMO)> create index idx2_t1 on t1 (c1, c2);
success
dbmMetaManager(DEMO)> desc t1;
--------------------------------------------------------------------------------
Instance=(DEMO) Table=(T1) Type=(TABLE) RowSize=(64)
--------------------------------------------------------------------------------
C1                              short               2         0
C2                              int                 4         4
C3                              long                8         8
C4                              float               4         16
C5                              double              8         24
C6                              char                20        32
C7                              date                 8        56
--------------------------------------------------------------------------------
IDX1_T1                         unique     (C1 asc)
IDX2_T1                                    (C1 asc, C2 asc)
--------------------------------------------------------------------------------
success
dbmMetaManager(DEMO)> select * from dic_index where table_name = 'T1';
--------------------------------------------------------------------------------
INST_NAME        : DEMO
```

```
TABLE_NAME       : T1
INDEX_NAME       : IDX1_T1
ID               : 15
IS_UNIQUE        : 1
KEY_SIZE         : 2
KEY_COLUMN_COUNT : 1
INDEX_ORDER      : 1
--------------------------------------------------------------------------------
INST_NAME        : DEMO
TABLE_NAME       : T1
INDEX_NAME       : IDX2_T1
ID               : 16
IS_UNIQUE        : 0
KEY_SIZE         : 6
KEY_COLUMN_COUNT : 2
INDEX_ORDER      : 2
--------------------------------------------------------------------------------
2 row selected
```

노트

- 테이블에는 하나 이상의 index가 반드시 존재해야 한다.
- Application에서는 **dbmSetIndex** API를 사용하여 index를 지정할 수 있다.

create queue

기능

First-In/ First-Out (FIFO) 방식으로 동작하는 queue 형태의 테이블을 생성한다.

구문

```
<create queue> ::= CREATE QUEUE queue_name SIZE msg_size
                       [ init <size> ]
                       [ extend <size> ]
                       [ max <size> ]
                   ;
<queue_name> :: 생성할 queue table의 이름이다.
<msg_size> :: 테이블에 저장될 메시지의 최대 크기이다.
init <size>: 첫 번째 segment에 저장할 수 있는 row 개수를 지정한다. (Default: 1,024개)
```

> extend <size>: init 된 공간이 모두 사용된 후 확장되는 segment에 저장할 수 있는 row 개수를 지정한다. (Default: 102,400개)
> max <size>: 최대로 확장할 수 있는 row 개수이다. (Default: 4,096,000개)

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> create queue que1 size 1024;
success
dbmMetaManager(DEMO)> desc que1;
-------------------------------------------------------------------------------
Instance=(DEMO) Table=(QUE1) Type=(QUEUE) RowSize=(1056)
-------------------------------------------------------------------------------
PRIORITY                        int             4       4
ID                              long            8       8
MSG_SIZE                        int             4       16
IN_TIME                         date            8       24
MESSAGE                         char            1024    32
-------------------------------------------------------------------------------
IDX_QUE1                        unique     (PRIORITY asc, ID asc)
-------------------------------------------------------------------------------
success
```

- Message의 ID는 enqueue 시점에 채번되는 internal sequence이다.
- Dequeue는 enqueue를 수행한 세션의 commit 순서에 따라 메시지를 가져오며, 동시에 commit된 경우에는 message의 ID가 더 작은 데이터를 먼저 읽는다.
  - 예를 들어, A 세션이 ID(1)을 부여받고 B 세션이 ID(2)를 부여받은 후 B 세션이 먼저 commit한 경우, dequeue는 ID(2)인 데이터를 먼저 읽는다.
  - 이후 A, B 세션이 모두 commit 된 상태에서 dequeue가 수행되면 ID(1)인 데이터를 먼저 읽는다.
- Enqueue로 삽입된 데이터는 commit이 완료된 이후에만 조회하거나 dequeue 할 수 있다.
  - Enqueue를 수행한 세션 역시 commit을 완료해야 자신이 enqueue한 데이터를 dequeue 할 수 있다.
- Dequeue로 한 건을 가져온 후 commit 하지 않은 세션이 존재하더라도, 다른 dequeue를 수행하는 세션은 이를 기다리지 않고 다음 데이터를 dequeue 한다.

노트

- Queue 생성 시, 메시지 크기는 사용자 입력값을 기준으로 8 바이트 단위로 정렬 (align)되어 설정된다. 따라서 API 사용 시에는 테이블의 MESSAGE column 크기를 참고하여 사용자 저장 변수를 할당해야 한다.

- Queue 테이블에는 생성 시 인덱스가 자동으로 생성되며, 해당 인덱스는 사용자가 임의로 조작하거나 변경할 수 없다.

create store

기능

String 또는 binary 형태의 데이터를 저장/ 변경/ 조회하기 위한 테이블을 생성한다.

구문

```
<create store> ::= CREATE STORE store_name KEY key_size VALUE value_size
                       [ init <size> ]
                       [ extend <size> ]
                       [ max <size> ]
                       ;
<store_name> :: 생성할 store table의 이름이다
<key_size> :: Key의 size 이다.
<value_size> :: Value의 size 이다.
INIT <size> :: 최초로 저장 가능한 record 개수이다. (Default: 1,024개)
EXTEND <size> :: init 된 공간이 모두 사용되어 확장될 때 저장 가능한 record 개수이다. (Default: 102,400개)
MAX <size> :: 최대로 확장할 수 있는 record 개수이다. (Default: 4,096,000개)
```

사용 예

```
************************************************************
* Copyright 2010. SUNJESOFT Inc. All rights reserved.
* Version (Debug 3.2-3.2.6 revision(6743))
* warning : open file limit 1024 is too low. : recommended 65536 or higher
************************************************************
dbmMetaManager(DEMO)> create store st1 key 32 value 1024;
success
dbmMetaManager(DEMO)> desc st1;
-------------------------------------------------------------------------------
Instance=(DEMO) Table=(ST1) Type=(STORE) RowSize=(1056) LockMode(1)
-------------------------------------------------------------------------------
ST_KEY                          char                32          0
ST_VALUE                        char                1024        32
-------------------------------------------------------------------------------
```

```
IDX_ST1                          unique     (ST_KEY asc)
-------------------------------------------------------------------------------
success
```

주의

- STORE 테이블의 column 이름을 임의로 변경하면 동작하지 않는다.
- STORE 테이블의 key-value 크기는 fixed size로 동작하며, 가변 크기는 지원하지 않는다.

create sequence

기능

Sequence 객체를 생성한다.

구문

```
<create sequence> ::= CREATE SEQUENCE sequence_name [options]
                    ;
<sequence_name> :: 생성할 sequence의 이름이다.
<options> ::= START WITH <value>
              ¦ INCREMENT BY <value>
              ¦ MAXVALUE <value>
              ¦ MINVALUE <value>
              ¦ CYCLE ¦ NOCYCLE
<START WITH>에 설정된 값은 <MAXVALUE>를 초과하여 생성할 수 없다.
<INCREMENT BY>를 생략할 경우 default는 1로 설정된다.
<MINVALUE>를 생략할 경우 default는 0으로 설정된다.
<MAXVALUE>를 생략할 경우 default는 LONG_MAX 값으로 설정된다.
<CYCLE> 옵션을 지정했을 때 sequence의 current value가 MaxValue를 초과하면 MinValue로 설정된다.
```

사용 예

```
*************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
*************************************************************
dbmMetaManager(DEMO)> create sequence seq1 start with 10 increment by 2 maxvalue 15 cycle;
success
dbmMetaManager(DEMO)> select seq1.nextval from dual;
-------------------------------------------------------------------------------
```

```
NEXTVAL : 10
--------------------------------------------------------------------------------
```

주의

Sequence는 트랜잭션 로깅 및 이중화가 지원되지 않는다.
따라서 failover 등으로 인해 서비스가 재시작되기 전에 사용자가 적절한 값으로 변경하여 사용해야 한다.
(자세한 내용은 **alter sequence [currval]**을 참조한다.)

create user_type

기능

사용자 데이터 형식을 정의한다.
char column에 별도의 구조화된 데이터를 저장하는 경우, 해당 데이터를 C 언어의 type-casting 과 유사한 방식으로 변환하여 출력하고 표현하기 위한 목적으로 사용한다.

구문

```
<create type> ::= CREATE USER_TYPE type_name
                      (
                         column_definition [, ...]
                      )
                      ;
<type_name> :: 사용자 지정 형식의 이름이다.
<column_definition> ::= column_name data_type_definition
<column_name> :: 사용자 지정 column의 이름이다.
<data_type_definition> ::= short
                               ¦ int
                               ¦ long
                               ¦ float
                               ¦ double
                               ¦ char (size)
                               ¦ date
```

사용 예

```
dbmMetaManager(DEMO)> create table t1 (c1 int, c2 char(100));
success
dbmMetaManager(DEMO)> insert into t1 values (1, 'ABCEFGHIJKLMN');
```

```
success
dbmMetaManager(DEMO)> commit;
success
dbmMetaManager(DEMO)> create user_type u1 (c1 char(3), c2 char(4));
success
dbmMetaManager(DEMO)> select user_type(c2, u1) from t1;
-------------------------------------------------------------------------------
USER_TYPE : C1=ABC C2=EFGH
-------------------------------------------------------------------------------
1 row selected
```

T1 테이블의 C2 column을 U1 구조로 출력하는 예제이다.

> 노트
> 세부 정의는 **USER_TYPE(Column_Name, Type_Name)**에 설명되어 있다.

drop index

기능

지정된 index를 제거한다.

구문

```
<drop index> ::= DROP INDEX index_name
                 ;
<index_name> :: 제거할 index의 이름이다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> desc t1;
-------------------------------------------------------------------------------
Instance=(DEMO) Table=(T1) Type=(TABLE) RowSize=(64)
-------------------------------------------------------------------------------
C1                              short               2          0
C2                              int                 4          4
```

```
C3                              long                8          8
C4                              float               4          16
C5                              double              8          24
C6                              char                20         32
C7                              date                 8         56
-------------------------------------------------------------------------------
IDX1_T1                         unique     (C1)
IDX2_T1                                    (C1, C2)
-------------------------------------------------------------------------------
success
dbmMetaManager(DEMO)> drop index idx2_t1;
success
dbmMetaManager(DEMO)> desc t1;
-------------------------------------------------------------------------------
Instance=(DEMO) Table=(T1) Type=(TABLE) RowSize=(64)
-------------------------------------------------------------------------------
C1                              short               2          0
C2                              int                 4          4
C3                              long                8          8
C4                              float               4          16
C5                              double              8          24
C6                              char                20         32
C7                              date                 8         56
-------------------------------------------------------------------------------
IDX1_T1                         unique     (C1)
-------------------------------------------------------------------------------
success
```

주의

STORE와 queue table에 생성된 index를 임의로 조작하는 것은 권장하지 않는다.

drop table (queue, store)

기능

사용자가 지정한 table (queue, store)과 해당 object에 생성된 모든 하위 index object를 제거한다.

구문

```
<drop table> ::= DROP TABLE table_name <force>
                 ¦
                 DROP QUEUE table_name
                 ¦
                 DROP STORE store_name
                 ;
<table_name> :: 제거 대상 table (queue, store)의 이름이다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> desc t1;
-------------------------------------------------------------------------------------
Instance=(DEMO) Table=(T1) Type=(TABLE) RowSize=(64)
-------------------------------------------------------------------------------------
C1                              short               2         0
C2                              int                 4         4
C3                              long                8         8
C4                              float               4         16
C5                              double              8         24
C6                              char                20        32
C7                              date                 8        56
-------------------------------------------------------------------------------------
IDX1_T1                         unique     (C1)
IDX2_T1                                    (C1, C2)
-------------------------------------------------------------------------------------
success
dbmMetaManager(DEMO)> drop table t1;
success
dbmMetaManager(DEMO)>
```

노트

FORCE 옵션은 동작 이상이나 사용자 실수로 인해 테이블 정보가 dictionary table에서 정상적으로 정리되지 않은 상태에서 강제로 제거하고자 할 때 지정한다.
FORCE 옵션으로도 제거되지 않은 세그먼트 파일이 /dev/shm에 남아있는 경우, 사용자가 직접 삭제해야 한

다.

drop sequence

기능

사용자가 지정한 sequence object를 제거한다.

구문

```
<drop sequence> ::= DROP SEQUENCE <sequence_name>
                    ;
<sequence_name> :: 제거할 sequence의 이름이다.
```

사용 예

```
dbmMetaManager(DEMO)> drop sequence seq10;
success
```

drop user_type

기능

사용자가 지정한 user_type object를 제거한다.

구문

```
<drop type> ::= DROP USER_TYPE <type_name>
                ;
<type_name> :: 제거할 type name이다.
```

사용 예

```
dbmMetaManager(DEMO)> drop user_type u1;
success
```

drop instance

기능

사용자가 지정한 instance 와 해당 instance에 포함된 모든 하위 object를 제거한다.
이 명령은 dictionary instance (dict)에 접속한 상태에서만 수행할 수 있다.

구문

```
<drop instance> ::= DROP INSTANCE instance_name [FORCE] [INCLUDE FILES]
                ;
<instance_name> :: 제거할 instance의 이름이다.
<FORCE> :: 복구 과정 등 특정 상황에서 메타 정보가 정상적으로 처리되지 않은 경우, instance를 강제로 제거하기 위해 지정한다.
<INCLUDE FILES> :: Instance에서 사용된 모든 파일 (logfile, anchor, datafile)을 함께 제거할 경우에 지정한다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> set instance dict;
success
dbmMetaManager(DICT)> drop instance demo;
success
dbmMetaManager(DICT)> set instance demo;
ERR-22008] fail to attach a shared memory segment
Command] <set instance demo>
```

노트

- FORCE 옵션은 동작 이상이나 사용자 실수로 인해 메타 정보가 정상적으로 정리되지 않은 상태에서 instance를 강제로 삭제할 때 사용한다. FORCE 옵션으로도 정리되지 않은 경우에는 사용자가 직접 삭제해야 한다.
- INCLUDE FILES 옵션으로도 삭제되지 않은 파일이 있는 경우, 해당 파일은 사용자가 직접 삭제해야 한다.

alter instance active/inactive

기능

사용자가 지정한 instance에 대한 접근 가능 여부를 설정한다.
Instance를 inactive로 설정하면, 해당 instance에 포함된 object에 대한 조회를 제외한 DDL/ DML 작업이 불가능해진다.
이 명령은 dictionary instance (dict)에 접속한 상태에서만 수행할 수 있다.

구문

```
<alter instance> ::= ALTER INSTANCE instance_name [ACTIVE ¦ INACTIVE]
                 ;
<instance_name> :: ACTIVE 혹은 INACTIVE 할 instance의 이름이다.
<ACTIVE> :: 해당 instance에 대한 DDL/DML을 허용한다.
<INACTIVE> :: 해당 instance에 대한 DDL/DML을 허용하지 않는다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> set instance dict;
success
dbmMetaManager(DICT)> alter instance demo inactive;
success
dbmMetaManager(DICT)> set instance demo;
success
dbmMetaManager(DEMO)> insert into t1 values (1, 1);
Command] <insert into t1 values (1, 1)>
ERR-22105] a instance not active-mode
dbmMetaManager(DEMO)> set instance dict;
success
dbmMetaManager(DICT)> alter instance demo active;
success
dbmMetaManager(DICT)> set instance demo;
success
dbmMetaManager(DEMO)> insert into t1 values (1, 1);
success
```

truncate table (queue, store)

기능

사용자가 지정한 테이블 (queue, store)을 초기화한다.
테이블에 저장된 모든 데이터와 extend 된 segment를 제거한 후, 테이블 생성 시점에 지정된 init size로 segment를 다시 생성한다.

구문

```
<truncate table> ::= TRUNCATE TABLE table_name
                     ¦
                     TRUNCATE QUEUE table_name
                     ¦
                     TRUNCATE STORE store_name
                     ;
<table_name> :: Truncate 할 table(queue, store)의 이름이다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(unknown)> set instance demo;
success
dbmMetaManager(DEMO)> truncate table t1;
success
```

주의
TRUNCATE 동작은 기존의 shared memory segment를 삭제한 후 새로운 세그먼트를 생성하는 과정으로 내부적으로는 DROP → CREATE 와 유사하게 동작한다.
이 과정에서 현재 진행 중인 트랜잭션과의 동시성은 보장되지 않으며, commit 시 오류로 처리될 수 있다.
따라서 TRUNCATE 수행 후 애플리케이션에 오류가 반환될 경우, 애플리케이션을 종료한 뒤 다시 시작하는 것을 권장한다.

rebuild index

기능

사용자가 지정한 index에 대해 rebuild 작업을 수행한다.

> 주의
> 이 명령은 기존 인덱스를 drop 하고 재생성하는 DDL으로 DML 과의 동시성이 보장되지 않는다. 따라서 작업 수행 전에 모든 application을 종료해야 한다.

구문

```
<compact table> ::= ALTER TABLE table_name REBUILD index_name
                    ;
<table_name> :: 대상 index의 table 이름이다.
<index_name> :: Rebuild 할 index의 이름이다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> ALTER TABLE t1 REBUILD idx1;
success
dbmMetaManager(DEMO)> SELECT * FROM t1;
-------------------------------------------------------------------------------
C1 : 3
C2 : 4
-------------------------------------------------------------------------------
1 row selected
```

> 노트
> Direct Table은 rebuild index 기능을 사용할 수 없다.

compact table

기능

사용자가 지정한 테이블에 대해 compaction 작업을 수행한다.
확장된 segment에 속한 데이터를 삭제하더라도 해당 메모리는 즉시 OS로 반환되지 않는다.
따라서 이 구문은 segment 내에서 사용하지 않는 메모리 공간을 OS에 반납해야 할 경우 사용할 수 있다.

주의

이 명령은 내부적으로 (데이터 export → truncate → 데이터 import) 과정을 수행하는 DDL이다. 따라서 작업 수행 전에는 모든 application을 모두 종료해야 한다.

구문

```
<compact table> ::= ALTER TABLE table_name COMPACT
                    ;
<table_name> :: Compact 할 table의 이름이다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(unknown)> set instance demo;
success
dbmMetaManager(DEMO)> ALTER TABLE t1 COMPACT;
success
```

노트

- Compact 기능은 B-tree 테이블에서만 지원된다.
- compac 명령이 수행되면 $DBM_HOME/trc 경로에 <instance name>_<table name>.dat.<scn> 형식의 백업 데이터 파일이 생성된다.
  Compact 작업이 실패하여 복구가 필요한 경우, 새로운 테이블을 생성한 후 dbmImp를 사용하여 데이터를 복구할 수 있다.

```
dbmImp -i demo -t t1 -d DEMO_T1.dat.123 -b
```

노트

- Compact 기능은 B-tree 테이블에서만 지원된다.
- compac 명령이 수행되면 $DBM_HOME/trc 경로에 <instance name>_<table name>.dat.<scn> 형식의 백업 데이터 파일이 생성된다.
  Compact 작업이 실패하여 복구가 필요한 경우, 새로운 테이블을 생성한 후 dbmImp를 사용하여 데이터를 복구할 수 있다.


```
dbmImp -i demo -t t1 -d DEMO_T1.dat.123 -b
```


add column

기능

사용자가 테이블에 column을 추가할 때 사용한다.
추가 위치는 지정할 수 없으며, 항상 테이블의 마지막 column으로 추가된다.
add column 기능은 디스크 모드에서는 사용할 수 없으며, normal 테이블 타입에서만 지원된다.

주의
이 명령은 내부적으로 (데이터 export → 테이블 재생성 → 데이터 import) 과정을 수행하는 DDL이다.
따라서 작업 수행 전에는 모든 application을 종료해야 한다.

구문

```
<add column> ::= ALTER TABLE table_name ADD COLUMN ( column_name datatype [default_value] )
                    ;
<table_name> :: Column을 추가할 table의 이름이다
<column_name> :: 추가할 column의 이름이다
<datatype> :: Column의 데이터 타입 이름이다. (int, short, long, date, char, float, double>
<default_value> :: date type을 제외한 데이터 타입에 대해 기본값을 지정할 경우에 정의한다.
(기본값은 상수만 허용된다.)
```

사용 예

```
dbmMetaManager(DEMO)> select * from t1 where c1 > 0
--------------------------------------------------------------------------------
C1                    : 1
C2                    : a
C3                    : 1
C4                    : a
--------------------------------------------------------------------------------
C1                    : 2
C2                    : b
C3                    : 2
C4                    : b
--------------------------------------------------------------------------------
C1                    : 3
C2                    : c
C3                    : 3
C4                    : c
--------------------------------------------------------------------------------
3 row selected
dbmMetaManager(DEMO)> alter table t1 add column (c5 char(10) default 'zz' )
success
dbmMetaManager(DEMO)> select * from t1 where c1 > 0
--------------------------------------------------------------------------------
C1                    : 1
C2                    : a
C3                    : 1
C4                    : a
C5                    : zz
--------------------------------------------------------------------------------
C1                    : 2
C2                    : b
C3                    : 2
C4                    : b
C5                    : zz
--------------------------------------------------------------------------------
C1                    : 3
C2                    : c
C3                    : 3
C4                    : c
C5                    : zz
```

```
--------------------------------------------------------------------------------
3 row selected
```

노트
add column 명령이 수행되면 $DBM_HOME/trc 경로에 <instance name>_<table name>.dat.<scn> 형식의 백업 데이터 파일이 생성된다.
add column 작업이 실패하여 복구가 필요한 경우, 새로운 테이블을 생성한 후 dbmImp를 사용하여 데이터를 복구할 수 있다.

```
dbmImp -i demo -t t1 -d DEMO_T1.dat.123 -b
```

주의
add/ drop column 기능은 디스크 로깅 모드에서 지원되지 않는다.

drop column

기능

사용자가 테이블에서 column을 제거할 때 사용한다.
drop column 기능은 normal 테이블 타입에서만 사용할 수 있고, 해당 column이 index key로 사용 중인 경우에는 수행할 수 없다.

주의
이 명령은 내부적으로 (데이터 export → 테이블 재생성 → 데이터 import) 과정을 수행하는 DDL이다.
따라서 작업 수행 전에는 모든 application을 종료해야 한다.

구문

```
<drop column> ::= ALTER TABLE table_name DROP COLUMN column_name
                  ;
<table_name> :: Column을 삭제할 table의 이름이다.
<column_name> :: 삭제할 column의 이름이다.
```

사용 예

```
dbmMetaManager(DEMO)> select * from t1 where c1 > 0;
--------------------------------------------------------------------------------
C1                    : 1
C2                    : a
C3                    : 1
C4                    : a
--------------------------------------------------------------------------------
C1                    : 2
C2                    : b
C3                    : 2
C4                    : b
--------------------------------------------------------------------------------
C1                    : 3
C2                    : c
C3                    : 3
C4                    : c
--------------------------------------------------------------------------------
3 row selected
dbmMetaManager(DEMO)> alter table t1 drop column c2;
success
dbmMetaManager(DEMO)> select * from t1 where c1 > 0;
--------------------------------------------------------------------------------
C1                    : 1
C3                    : 1
C4                    : a
--------------------------------------------------------------------------------
C1                    : 2
C3                    : 2
C4                    : b
--------------------------------------------------------------------------------
C1                    : 3
C3                    : 3
C4                    : c
--------------------------------------------------------------------------------
3 row selected
```

노트

drop column 명령이 수행되면 $DBM_HOME/trc 경로에 <instance name>_<table name>.dat.<scn> 형식의 백업 데이터 파일이 생성된다.
drop column 작업이 실패하여 복구가 필요한 경우, 새로운 테이블을 생성한 후 dbmImp를 사용하여 데이터를 복구할 수 있다.

```
dbmImp -i demo -t t1 -d DEMO_T1_3.dat -b
```

주의

add/ drop column 기능은 디스크 로깅 모드에서 지원되지 않는다.

rename column

기능

사용자가 테이블에서 column의 이름만 변경할 때 사용한다.

구문

```
<rename column> ::= ALTER TABLE table_name RENAME COLUMN org_column_name TO new_column_name
                  ;
<table_name> :: Column 이름을 변경할 대상 table의 이름이다.
<org_column_name> :: 변경 전 column의 이름이다.
<new_column_name> :: 변경 후 새로 사용할 column의 이름이다.
```

사용 예

```
dbmMetaManager(DEMO)> desc t1
-------------------------------------------------------------------------------
Instance=(DEMO) Table=(T1) Type=(TABLE) RowSize=(12)
-------------------------------------------------------------------------------
C1                              int                 4          0
C2                              int                 4          4
C3                              int                 4          8
-------------------------------------------------------------------------------
success
```

```
dbmMetaManager(DEMO)> alter table t1 rename column c2 to x359
success
dbmMetaManager(DEMO)> desc t1
-------------------------------------------------------------------------------
Instance=(DEMO) Table=(T1) Type=(TABLE) RowSize=(12)
-------------------------------------------------------------------------------
C1                              int                 4          0
X359                            int                 4          4
C3                              int                 4          8
-------------------------------------------------------------------------------
success
```

> 노트
> Store 테이블과 queue 테이블에서는 rename 명령을 사용할 수 없다.

create replication

기능

Replication 대상 테이블을 DIC_REPL_TABLE 목록에 등록한다.

구문

```
<create replication> ::= CREATE REPLICATION TABLE [TableList]
                  ;
<TableList> ::= TableName [, tableName]
```

사용 예

```
dbmMetaManager(DEMO)> create replication table t1, t2;
success
```

등록된 이중화 대상 테이블은 다음과 같이 조회할 수 있다.

```
dbmMetaManager(DEMO)> select * from DIC_REPL_TABLE;
-------------------------------------------------------------------------------
INST_NAME  : DEMO
TABLE_NAME : T1
-------------------------------------------------------------------------------
```

```
INST_NAME  : DEMO
TABLE_NAME : T2
------------------------------------------------------------------------------
2 row selected
```

노트

동일 트랜잭션 내에서도 DIC_REPL_TABLE에 등록되어 있지 않은 테이블은 이중화 대상에 포함되지 않는다.

alter replication

기능

Replication 대상 테이블을 추가/ 삭제한다.

구문

```
<alter replication> ::= ALTER REPLICATION [ADD|DROP] TABLE [TableList]
                    ;
<ADD> :: 테이블을 이중화 대상으로 추가한다.
<DROP> :: 테이블을 이중화 대상에서 제거한다.
<TableList> ::= TableName [, tableName]
```

사용 예

```
dbmMetaManager(DEMO)> alter replication add table t1, t2;
success
dbmMetaManager(DEMO)> alter replication drop table t1, t2;
success
```

노트

alter replication add/drop 구문은 현재 실행 중인 애플리케이션에 영향을 주지 않는다.
변경 사항을 적용하려면 애플리케이션을 재기동해야 한다.

drop replication

기능

Replication 대상 테이블을 DIC_REPL_TABLE 목록에서 제거한다.

구문

```
<drop replication> ::= DROP RPELICATION
                       ;
```

사용 예

```
dbmMetaManager(DEMO)> drop replication;
success
```

set instance

기능

사용자가 지정한 instance로 전환한다.

구문

```
<set instance> ::= SET INSTANCE instance_name
                   ;
<instance_name> :: 전환할 instance의 이름이다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(unknown)> set instance dict;
success
dbmMetaManager(DICT)> set instance demo;
success
dbmMetaManager(DEMO)>
```

노트

set instance 구문은 dbmMetaManager에서만 동작한다.

alter sequence [currval]

기능

사용자가 생성한 sequence의 현재 값을 지정된 값으로 변경한다.

구문

```
<alter sequence> ::= ALTER SEQUENCE sequence_name SET CURRVAL = value
                 ;
<sequence_name> :: 변경할 sequence의 이름이다.
<value> :: 사용자가 변경할 current value 이다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> select seq1.currval from dual;
-------------------------------------------------------------------------------
CURRVAL              : 3
-------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> alter sequence seq1 set currval = 1000;
success
dbmMetaManager(DEMO)> select seq1.currval from dual;
-------------------------------------------------------------------------------
CURRVAL              : 1000
-------------------------------------------------------------------------------
1 row selected
```

alter system replication sync

기능

이중화의 장애로 데이터를 동기화 해야 할 경우 사용하는 구문이다.

구문

```
<reset perf> ::= ALTER SYSTEM REPLICATION SYNC <ALL ¦ Tables >
                 ;
ALL : 전체 테이블 대상
Tables : 테이블 목록
```

사용 예

```
dbmMetaManager(DEMO)> alter system replication sync all;
success
```

> 노트
> 네트워크 장애등으로 동기화가 중단된 이후 (Master->Slave) 방향으로 데이터를 동기화 하고자 할 경우 사용한다. 구문은 Master side에서 실행해야 하며 Slave의 Segment는 conflict가 발생하지 않도록 비워진 상태이어야 한다.

alter system reset checkpoint

기능

체크포인트를 특정 로그 파일 번호부터 수행할 수 있도록 해당 로그 파일 번호를 설정한다.

구문

```
<reset perf> ::= ALTER SYSTEM RESET checkpoint <instanceName> <logfile_number>
                 ;
<instanceName> ::        ❶ 대상 instance 이름 입력
<logfile_number> ::      ❷ 특정 로그 파일 번호
```

사용 예

```
dbmMetaManager(DEMO)> alter system reset checkpoint demo -1;
success
```

노트

시스템 장애로 인해 데이터 파일이 삭제되었고, archive logfile을 통해 복구가 가능한 환경인 경우 체크포인트를 수행하여 데이터 파일을 다시 생성할 수 있다. 이때 이 구문을 사용하여 체크포인트 시작 파일 번호를 지정한다.

alter system reset perf

기능

DBM_PERF_ENABLE 속성이 활성화 된 경우, vsys_stat, vsess_stat 등에 성능 통계 정보가 누적된다. 이 명령은 이러한 누적 통계 정보를 모두 초기화한다.

구문

```
<reset perf> ::= ALTER SYSTEM RESET PERF
                 ;
```

사용 예

```
dbmMetaManager(DEMO)> alter system reset perf;
success
```

alter system refine [TableList]

기능

특정 테이블이나 인덱스에 대한 잠금을 유지한 상태에서 프로세스 등이 비정상적으로 종료된 경우 다른 세션의 접근도 차단된다. refine 과정은 해당 상태를 복구하기 위한 기능이다.
이 명령은 위와 같은 상태에 놓인 index segment를 찾아 정상화 작업을 수행한다. 정상화 과정은 대상 index를 재구축하는 방식 (index segment 재생성) 으로 수행된다.

구문

```
<reset perf> ::= ALTER SYSTEM REFINE [TableName, ...]
                 ;
<TableName> :: 특정 테이블에 대해서만 작업을 수행하고자 할 경우에 지정하는 옵션이다.
```

사용 예

```
dbmMetaManager(DEMO)> alter system refine;
success
```

> 주의
> 이 명령은 내부적으로 index를 rebuild 하는 DDL이므로, 모든 application을 종료한 후 실행해야 한다.

Data Manipulation Language(DML)

데이터를 저장/ 삭제/ 조회/ 변경하기 위한 구문과 enqueue/ dequeue 관련 구문을 설명한다.

insert

기능

사용자가 지정한 테이블에 데이터를 삽입한다.

구문

```
<insert> ::= INSERT INTO table_name
             [ column_name [, ...] ]
             VALUES
             ( value_expression [, ...] )
             ;
<table_name> :: 데이터를 삽입할 대상 테이블이다.
<column_name> :: 삽입할 데이터와 매핑되는 column을 지정한다. (생략할 수 있다.)
<value_expression> :: 지정한 column에 삽입할 값을 표현한다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> desc t1;
-------------------------------------------------------------------------------
Instance=(DEMO) Table=(T1) Type=(TABLE) RowSize=(48)
```

```
-------------------------------------------------------------------------------
C1                              int                 4          0
C2                              double              8          8
C3                              char                20         16
C4                              date                 8         40
-------------------------------------------------------------------------------
IDX_T1                          unique     (C1)
-------------------------------------------------------------------------------
success
dbmMetaManager(DEMO)> insert into t1 (c1, c2, c3, c4) values (1, 1, 1, sysdate);
success
dbmMetaManager(DEMO)> insert into t1 (c1, c2) values (2, 2);
success
dbmMetaManager(DEMO)> insert into t1 values (3, 3, 3, sysdate);
success
dbmMetaManager(DEMO)> select * from t1;
-------------------------------------------------------------------------------
C1                  : 1
C2                  : 1.000000
C3                  : 1
C4                  : 2019/01/02 16:44:17.227558
-------------------------------------------------------------------------------
C1                  : 2
C2                  : 2.000000
C3                  :
C4                  : 1970/01/01 09:00:00.000000
-------------------------------------------------------------------------------
C1                  : 3
C2                  : 3.000000
C3                  : 3
C4                  : 2019/01/02 16:44:37.283193
-------------------------------------------------------------------------------
3 row selected
```

노트

Unique index가 설정된 테이블에 동일한 key 값으로 insert가 수행될 경우, 선행 트랜잭션이 종료될 때까지 후행 트랜잭션은 대기한다.

update

기능

사용자가 지정한 테이블에서 하나 이상의 데이터를 갱신한다.

구문

```
<update> ::= UPDATE table_name
             SET column_name = value_expression [, ...]
             [ WHERE cond_expression ]
             ;
<table_name> :: 데이터를 갱신할 대상 테이블이다.
<column_name> :: 값을 변경할 테이블의 column 이름이다.
<value_expression> :: 해당 column에 갱신할 value를 표현한다.
<cond_expression> :: 갱신 대상 데이터를 탐색하기 위한 조건절이다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> select * from t1;
-------------------------------------------------------------------------------
C1                   : 1
C2                   : 1.000000
C3                   : 1
C4                   : 2019/01/02 16:44:17.227558
-------------------------------------------------------------------------------
C1                   : 2
C2                   : 2.000000
C3                   :
C4                   : 1970/01/01 09:00:00.000000
-------------------------------------------------------------------------------
C1                   : 3
C2                   : 3.000000
C3                   : 3
C4                   : 2019/01/02 16:44:37.283193
-------------------------------------------------------------------------------
3 row selected
dbmMetaManager(DEMO)> update t1 set c2 = 100 where c1 >= 1;
3 row updated.
```

```
dbmMetaManager(DEMO)> select * from t1;
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 100.000000
C3                   : 1
C4                   : 2019/01/02 16:44:17.227558
--------------------------------------------------------------------------------
C1                   : 2
C2                   : 100.000000
C3                   :
C4                   : 1970/01/01 09:00:00.000000
--------------------------------------------------------------------------------
C1                   : 3
C2                   : 100.000000
C3                   : 3
C4                   : 2019/01/02 16:44:37.283193
--------------------------------------------------------------------------------
3 row selected
```

노트

Update로 갱신할 데이터는 잠금 (lock) 되며, 해당 데이터에 대한 다른 세션의 접근은 차단된다.
다른 세션에서 갱신 중인 레코드를 select 할 경우, update 이전에 커밋된 데이터가 조회된다.
(단, DBM_MVCC_ENABLE = FALSE 로 설정된 경우에는 update가 완료될 때까지 대기한다.)

주의

Index key column은 변경할 수 없다.

delete

기능

사용자가 지정한 테이블에서 하나 이상의 데이터를 삭제한다.

구문

```
<delete> ::= DELETE FROM table_name
             [ WHERE cond_expression ]
             ;
<table_name> :: 데이터를 삭제할 대상 테이블이다.
<cond_expression> :: 삭제할 데이터를 탐색하기 위한 조건절이다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> select * from t1;
-------------------------------------------------------------------------------
C1                   : 1
C2                   : 1.000000
C3                   : 1
C4                   : 2019/01/02 16:44:17.227558
-------------------------------------------------------------------------------
C1                   : 2
C2                   : 2.000000
C3                   :
C4                   : 1970/01/01 09:00:00.000000
-------------------------------------------------------------------------------
C1                   : 3
C2                   : 3.000000
C3                   : 3
C4                   : 2019/01/02 16:44:37.283193
-------------------------------------------------------------------------------
3 row selected
dbmMetaManager(DEMO)> delete from t1 where c1 >= 1;
3 row deleted.
dbmMetaManager(DEMO)> select * from t1;
-------------------------------------------------------------------------------
0 row selected
```

노트

Delete로 삭제할 레코드는 잠금 (lock) 되며, 해당 데이터에 대한 다른 세션의 접근은 차단된다.

다른 세션에서 삭제 중인 레코드를 select 할 경우, delete 이전에 커밋된 데이터가 조회된다.
(단, DBM_MVCC_ENABLE = FALSE 로 설정된 경우에는 delete가 완료될 때까지 대기한다.)

주의
B-tree table의 경우 delete 연산이 수행되더라도 index key는 commit 시점에 삭제된다. 반면, splay table type에서는 delete가 수행될 때 index key가 즉시 삭제된다.
커밋이 완료되지 않았더라도, splay table에서 delete로 삭제된 데이터는 다른 세션에서 조회할 수 없다.
또한 delete를 포함한 트랜잭션을 롤백하는 시점에 이미 다른 세션에 의해 동일한 key가 삽입된 경우, delete에 대한 롤백 처리는 수행되지 않고 skip된다.

select 및 select for update

기능

사용자가 지정한 테이블에서 하나 이상의 데이터를 조회한다.

구문

```
<select> ::= SELECT target_list FROM table_name
             [ WHERE cond_expression ]
             [ FOR UPDATE ]
             ;

<target_list> ::=   *
                  ¦ column_name [, ... ]
<table_name> :: 데이터를 조회할 대상 테이블 이다.
<cond_expression> :: 조회할 데이터를 탐색하기 위한 조건절 이다.
<FOR UPDATE> :: Select 할 때 해당 데이터에 lock을 설정하여 다른 트랜잭션에서 데이터가 변경되는 것을 방지하기 위해 사용한다.
```

사용 예

```
dbmMetaManager(DEMO)> select * from user_data;
-------------------------------------------------------------------------------
EMPNO   : 1
EMPNAME : alice
DEPTNO  : 100
BIRTH   : 2025/07/30 08:08:15.484030
```

```
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select * from user_data for update;
--------------------------------------------------------------------------------
EMPNO   : 1
EMPNAME : alice
DEPTNO  : 100
BIRTH   : 2025/07/30 08:08:15.484030
--------------------------------------------------------------------------------
1 row selected
```

주의

Auto commit mode에서는 FOR UPDATE 기능을 사용할 수 없다.

enqueue

기능

사용자가 지정한 테이블에 데이터 한 건을 enqueue 한다.

구문

```
<enqueue> ::= ENQUEUE INTO table_name
              [ target_list ]
              VALUES
              ( value_expression [, ...] )
              ;
<target_list> ::= ( column_name [, ...] )
<table_name> :: 데이터를 삽입할 대상 테이블이다.
<value_expression> :: 삽입할 value 이다.
```

노트

Enqueue 구문에서 사용자가 지정할 수 있는 column은 priority, msg_size, message 이다.

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> create queue que1 size 100;
success
dbmMetaManager(DEMO)> enqueue into que1 (priority, msg_size, message) values (90, 10, 'msg
1234567');
success
dbmMetaManager(DEMO)> commit;
success
dbmMetaManager(DEMO)> select * from que1;
--------------------------------------------------------------------------------
PRIORITY : 90
ID       : 2
MSG_SIZE : 10
IN_TIME  : 2025/07/04 16:11:57.357598
MESSAGE  : msg1234567
--------------------------------------------------------------------------------
1 row selected
```

노트

- msg_size 와 message는 필수 입력 항목이다.
- Priority를 지정하지 않는 경우 , 기본값으로 0 이 저장된다.

Enqueue 대상 테이블에 대해 insert 구문을 사용하여 강제로 데이터를 삽입할 경우, 다음과 같은 오류가 발생한다.

```
dbmMetaManager(DEMO)> insert into que1 (priority, msg_size, message) values (90, 10, 'msg
1234567');
Command] <insert into que1 (priority, msg_size, message) values (90, 10, 'msg1234567')>
ERR-22073] a operation can not be executed on target-table (check table type or mode)
```

노트

내부적으로 관리되는 message의 ID는 enqueue 시점에 채번되며, 커밋된 데이터는 message ID 순서대로 dequeue된다.

dequeue

기능

사용자가 지정한 테이블에서 데이터 한 건을 dequeue 한다.

구문

```
<dequeue> ::= DEQUEUE FROM table_name
              [ WHERE cond_expression ]
              [ TIMEOUT seconds ]
              ;
<table_name> :: Dequeue를 수행할 대상 테이블이다.
<cond_expression> :: Dequeue 할 데이터를 선택하는 조건이다.
<seconds> :: 데이터가 없는 경우 대기할 시간을 지정한다. (단위: 초)
              0: 데이터가 없으면 즉시 에러를 반환한다.
             ¦ 음수: 데이터가 없으면 무한히 대기한다.
             ¦ 양수: 지정한 timeout 값만큼 대기한다.
```

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
ddbmMetaManager(DEMO)> select * from que1;
-------------------------------------------------------------------------------
PRIORITY : 0
ID       : 10
MSG_SIZE : 10
IN_TIME  : 2025/07/11 18:13:19.466761
MESSAGE  : msg1234
-------------------------------------------------------------------------------
PRIORITY : 0
ID       : 11
MSG_SIZE : 10
IN_TIME  : 2025/07/11 18:13:19.466762
MESSAGE  : msg5678
-------------------------------------------------------------------------------
2 row selected
dbmMetaManager(DEMO)> dequeue from que1 where ID = 11;
PRIORITY            : 0
ID                  : 11
```

```
MSG_SIZE             : 10
IN_TIME              : 2025/07/11 18:13:19.466762
MESSAGE              : msg5678
2 row selected
```

노트

- Priority가 낮은 값의 레코드부터 dequeue 된다.
- Priority 값이 같을 경우에는, commit 된 데이터의 message ID 순서대로 dequeue 된다.
- 한 건을 dequeue 한 세션이 commit 하지 않더라도, 다른 세션은 이를 기다리지 않고 다음 데이터를 바로 dequeue 한다.

set

기능

Store 타입 테이블에서 지정한 key에 대응하는 value를 저장하거나 갱신한다.

- 사용자가 지정한 테이블에 key가 존재하지 않으면 value가 새로 삽입된다.
- 사용자가 지정한 테이블에 key가 이미 존재하면 value가 갱신된다.

구문

```
<set> ::= SET key value AT table_name
          [ nx ]
          ;
* key, value: Store에 저장할 사용자 key와 value 이다.
* nx: Key가 이미 존재할 경우 기본 동작은 갱신이지만, nx 옵션이 지정되면 중복 오류를 반환한다.
* table_name: 데이터를 삽입하거나 갱신할 대상 store 타입 테이블의 이름이다.
```

사용 예

다음은 store 테이블에 데이터를 저장하는 예이다.

```
dbmMetaManager(DEMO)> create store st1 key 32 value 100;
success
dbmMetaManager(DEMO)> set 'k1' 'v1' at st1;
success
dbmMetaManager(DEMO)> select * from st1;
--------------------------------------------------------------------------------
ST_KEY   : k1
```

```
ST_VALUE : v1
--------------------------------------------------------------------------------
1 row selected
```

> 노트
> 서로 다른 세션에서 같은 key를 set 할 경우, insert와 동일하게 선행 트랜잭션이 완료될 때까지 후행 트랜잭션은 대기한다. 동일한 세션에서 같은 key 값으로 삽입을 시도하면, 값이 변경되지 않고 duplicated 오류가 발생한다.

NX 옵션 사용 시 key가 이미 존재하면 오류가 반환된다.

```
dbmMetaManager(DEMO)> set 'k1' 'v99' at st1 nx;
Command] <set 'k1' 'v99' at st1 nx>
ERR-22055] key value duplicated (IDX_ST1)
```

다음은 기존 key value를 갱신하는 예이다.

```
dbmMetaManager(DEMO)> set 'k1' 'v99' at st1;
success
dbmMetaManager(DEMO)> select * from st1;
--------------------------------------------------------------------------------
ST_KEY   : k1
ST_VALUE : v99
--------------------------------------------------------------------------------
1 row selected
```

get

기능

Store 타입 테이블에서 지정한 key에 대응하는 value를 조회한다.

구문

```
<get> ::= GET key AT table_name
          ;
* key: 조회할 key 값이다.
* table_name: 데이터를 조회할 대상 테이블의 이름이다.
```

사용 예

```
dbmMetaManager(DEMO)> get 'k1' at st1;
--------------------------------------------------------------------------------
ST_KEY   : k1
ST_VALUE : v1
--------------------------------------------------------------------------------
1 row selected
```

> 노트
> Store 타입 테이블은 SELECT 문을 이용하여 조회할 수도 있다.

del

기능

Store 타입 테이블에서 지정한 key에 해당하는 value를 삭제한다.

구문

```
<del> ::= DEL key AT table_name
          ;
* key: 삭제할 key 값이다.
* table_name: 삭제할 데이터의 대상 테이블 이름이다.
```

사용 예

```
dbmMetaManager(DEMO)> get 'k1' at st1;
--------------------------------------------------------------------------------
ST_KEY   : k1
ST_VALUE : v1
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> del 'k1' at st1;
1 row deleted.
dbmMetaManager(DEMO)> get 'k1' at st1;
--------------------------------------------------------------------------------
0 row selected
```

Data Control Language (DCL)

DCL은 사용자가 수행한 각 트랜잭션을 commit/ rollback 하는 데 사용되는 구문이다.

commit

기능

사용자가 수행한 트랜잭션의 변경 사항을 데이터베이스에 영구적으로 반영한다.

구문

```
<commit> ::= COMMIT
             ;
```

사용 예

```
dbmMetaManager(DEMO)> commit;
success
```

> 주의
> Commit 과정에서 대상 object에 DDL이 수행되면 commit은 오류와 함께 중단된다.
> 이 경우에는 commit, rollback 및 기타 operation을 수행할 수 없는 상태가 되므로, 프로세스를 종료한 후 새로운 세션에서 복구 과정을 수행해야 한다.
> 따라서 commit 과정에서 오류가 발생한 경우에는 트랜잭션이 정상적으로 종료되었는지 반드시 확인해야 한다.

rollback

기능

사용자가 수행한 트랜잭션의 변경 사항을 취소하고, 트랜잭션 시작 이전 상태로 복원한다.

구문

```
<rollback> ::= ROLLBACK
               ;
```

사용 예

```
dbmMetaManager(DEMO)> rollback;
success
```

> 주의
> Rollback 과정에서 대상 object에 DDL이 수행되면 rollback은 오류와 함께 중단된다.
> 이 경우에는 commit, rollback 및 기타 operation을 수행할 수 없는 상태가 되므로, 프로세스를 종료한 후 새로운 세션에서 복구 과정을 수행해야 한다.
> 따라서 rollback 과정에서 오류가 발생한 경우에는 트랜잭션이 정상적으로 종료되었는지 반드시 확인해야 한다.

Built-in Function

- 사용자의 편의를 위해 다음과 같은 built-in function을 제공한다.
- 반환되는 문자열의 최대 크기는 32 Kbytes 이다.
- GOLDILOCKS LITE에서 제공하는 모든 숫자형 함수는 overflow/ underflow 관련 오류를 체크하지 않는다.

Category	Function name	Return type	Desc
날짜/ 시간	sysdate	long (8 bytes)	내부 저장용 8 byte long long 형태의 값으로 저장하고 출력한다.
	extract	int (4 bytes)	날짜 타입 값에서 지정된 항목의 값을 숫자형으로 반환한다.
	datetime_str	char (64 bytes 이내)	Date column을 문자열로 출력한다.
	to_date	date (8 bytes)	사용자 입력 문자열을 date type 값으로 변환한다.
	datediff	long (8 bytes)	두 날짜 타입 인자 간의 차이를 초 단위로 출력한다.
Dump	dump	char (최대 1Mbyte)	ascii 값을 출력한다. Byte당 2~4 byte로 출력된다.
	hex	char (최대 1Mbyte)	Hex 값을 출력한다. byte당 2 byte로 출력된다.
	concat	char (최대 1Mbyte)	두 번째 인자의 문자열을 첫 번째 인자의 문자열 뒤에 결합하여 출력한다.
			소스 문자열 내에 지정 문자열

Category	Function name	Return type	Desc
문자형	instr	int (4 bytes)	이 존재할 경우, 시작 위치를 1로 하는 offset을 출력한다.
	replace	char (최대 1Mbyte)	검색 문자열을 찾아 지정된 문자열로 치환한다.
	substr	char (최대 1Mbyte)	문자열의 지정 위치부터 입력된 크기만큼 잘라낸다.
	length	int (4 bytes)	NULL 지점까지의 길이를 반환하며, NULL이 없을 경우 오류가 발생할 수 있다.
	ltrim	char (최대 1 Mbyte)	문자열의 왼쪽 공백 또는 지정된 문자를 제거한다.
	rtrim	char (최대 1 Mbyte)	문자열의 오른쪽 공백 또는 지정된 문자를 제거한다.
	lpad	char (최대 1 Mbyte)	문자열의 왼쪽에 지정한 문자열을 추가한다.
	rpad	char (최대 1 Mbyte)	문자열의 오른쪽에 지정한 문자열을 추가한다.
	upper	char (최대 1 Mbyte)	값을 대문자로 변환한다.
	lower	char (최대 1 Mbyte)	값을 소문자로 변환한다.
숫자형	abs	double (8 bytes)	절대값으로 변환한다.
	power	double (8 bytes)	입력값의 제곱을 출력한다.
	sqrt	double (8 bytes)	입력값의 제곱근을 출력한다.
	log	double (8 bytes)	지정된 base 기준의 자연 로그 값을 출력한다.
	exp	double (8 bytes)	e의 제곱 값을 출력한다.
	mod	double (8 bytes)	나머지 연산을 한다.
	ceil	double (8 bytes)	소수점을 올림한다.
	floor	double (8 bytes)	소수점을 내림한다.
	round	double (8 bytes)	반올림 한다.
	trunc	double (8 bytes)	절사 연산을 한다.
	random	Int (4 bytes)	주어진 입력값 내의 random 정수를 출력한다.
Sequence	currval	long long (8 bytes)	-
	nextval	long long (8 bytes)	-
단방향 hash 암호화	digest	char (64 bytes)	알고리즘에 따라 서로 다른 길이값을 반환한다.
Aggregation	min	double (8 bytes)	group by를 지원하지 않는다.
	max	double (8 bytes)	
	avg	double (8 bytes)	
	sum	double (8 bytes)	
			일반 테이블 조회 결과를 JSO

Category	Function name	Return type	Desc
JSON OBJECT	JSON_STRING	char	N 형태로 반환한다.
기타	decode	char (최대 1 Mbyte)	결과와 일치할 경우 사용자가 지정한 값을 반환한다.
	nvl	char (최대 1 Mbyte)	지정된 값이 NULL일 경우 사용자가 지정한 값을 반환한다.
	user_type	char (최대 1 Mbyte)	Column을 user_type으로 캐스팅하여 출력한다.

sysdate

현재 시스템 시간을 조회하여 date type으로 반환한다.
(단, dbmMetaManager에서 조회할 경우에는 long long type이 아닌 string 형태로 반환된다.)

사용 예

```
dbmMetaManager(DEMO)> create table t1 (c1 int, c2 date);
success
dbmMetaManager(DEMO)> insert into t1 values (1, sysdate);
success
dbmMetaManager(DEMO)> commit;
success
dbmMetaManager(DEMO)> select * from t1;
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 2019/04/04 13:08:47.416266
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select sysdate from dual;
--------------------------------------------------------------------------------
SYSDATE              : 2021/02/01 15:32:50.110855
--------------------------------------------------------------------------------
1 row selected
```

노트

Sysdate은 unix time 값을 반환한다.

extract

지정한 날짜/시간 타입 값에서 입력된 항목에 해당하는 필드를 숫자형으로 추출하여 반환한다.
추출 가능한 항목은 다음과 같다.

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND

사용 예

```
dbmMetaManager(DEMO)> select extract( 'year' from to_date('20211231115859', 'yyyymmddhhmiss')
) from dual;
--------------------------------------------------------------------------------
EXTRACT              : 2021
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select extract( 'month' from to_date('20211231115859', 'yyyymmddhhmiss')
) from dual;
--------------------------------------------------------------------------------
EXTRACT              : 12
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select extract( 'day' from to_date('20211231115859', 'yyyymmddhhmiss') )
from dual;
--------------------------------------------------------------------------------
EXTRACT              : 31
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select extract( 'hour' from to_date('20211231115859', 'yyyymmddhhmiss')
) from dual;
--------------------------------------------------------------------------------
EXTRACT              : 11
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select extract( 'minute' from to_date('20211231115859', 'yyyymmddhhmiss
') ) from dual;
--------------------------------------------------------------------------------
EXTRACT              : 58
```

```
------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select extract( 'second' from to_date('20211231115859', 'yyyymmddhhmiss
') ) from dual;
------------------------------------------------------------------------------
EXTRACT              : 59
------------------------------------------------------------------------------
1 row selected
```

datetime_str

Date type의 column을 string 형태로 변환하여 출력한다.

- 출력 포맷은 YYYY/MM/DD H24:MI:SS.SSSSSS로 고정되어 있다.
- 인자로는 date type 값을 사용할 수 있다.

사용 예

```
dbmMetaManager(DEMO)> select datetime_str(sysdate) from dual;
------------------------------------------------------------------------------
DATETIME_STR         : 2025/07/04 17:34:32.436717
------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)>
```

to_date('문자열', 'Format')

사용자 입력 문자열을 지정한 형식에 따라 date 타입 값으로 변환한다.
Format을 지정하지 않은 경우, default 형식은 YYYY/MM/DD H24:MI:SS.S6 로 고정되어 있다.

변환에 사용 가능한 형식 요소는 다음 표를 참조한다.

Format	설명
YYYY	4 자리 연도이다.
MM	월의 단위로서 (01~12) 범위이다.
DD	일의 단위로서 (01~31) 범위이다.
HH	시간의 단위로서 (00~23) 범위이다.
MI	분의 단위로서 (00~59) 범위이다.
SS	초의 단위로서 (00~59) 범위이다.
S3	Millisecond까지 사용하는 경우로서 3 자리이다.
S6	Microsecond까지 사용하는 경우로서 6 자리이다.

사용 예

```
dbmMetaManager(DEMO)> select to_date( '2020/12/31 15:38:41.123456', 'yyyy/mm/dd hh:mi:ss.s6')
from dual;
------------------------------------------------------------------------------
TO_DATE              : 2020/12/31 15:38:41.123456
------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select to_date( '2020/12/31 15:38:41.123', 'yyyy/mm/dd hh:mi:ss.s3')
from dual;
------------------------------------------------------------------------------
TO_DATE              : 2020/12/31 15:38:41.123000
------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select to_date( '2020/12/31 15:38:41', 'yyyy/mm/dd hh:mi:ss') from dual;
------------------------------------------------------------------------------
TO_DATE              : 2020/12/31 15:38:41.000000
------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select to_date( '2020/12/31 15:38', 'yyyy/mm/dd hh:mi') from dual;
------------------------------------------------------------------------------
TO_DATE              : 2020/12/31 15:38:00.000000
------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select to_date( '2020/12/31 15', 'yyyy/mm/dd hh') from dual;
------------------------------------------------------------------------------
TO_DATE              : 2020/12/31 15:00:00.000000
------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select to_date( '2020/12/31', 'yyyy/mm/dd') from dual;
------------------------------------------------------------------------------
TO_DATE              : 2020/12/31 00:00:00.000000
------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select to_date( '2020/12', 'yyyy/mm') from dual;
------------------------------------------------------------------------------
TO_DATE              : 2020/12/01 00:00:00.000000
------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select to_date( '2020', 'yyyy') from dual;
------------------------------------------------------------------------------
```

```
TO_DATE              : 2020/12/01 00:00:00.000000
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select to_date( '11:31', 'hh:mi') from dual;
--------------------------------------------------------------------------------
TO_DATE              : 2020/12/01 11:31:00.000000
--------------------------------------------------------------------------------
1 row selected
```

Format이 지정되지 않은 경우, 사용자 입력 문자열은 기본 포맷 형태와 일치해야 한다.
이 때 문자열에는 연/월/일 정보가 반드시 포함되어야 하며, 포함되지 않은 경우 오류가 발생한다.

```
dbmMetaManager(DEMO)> select to_date('12:40') from dual;
Command] <select to_date('12:40') from dual>
ERR-22047] invalid expression type
ERR-22001] invalid parameters or usage at internal processing
```

다음은 DML 문에 함수를 포함하여 사용하는 예이다.

```
dbmMetaManager(DEMO)> insert into t1 values (1, sysdate);
success
dbmMetaManager(DEMO)> insert into t1 values (1, to_date('2002/05/26 11:31:45.123456') );
success
dbmMetaManager(DEMO)> commit;
success
dbmMetaManager(DEMO)> select * from t1;
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 2020/12/28 16:26:18.548964
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 2002/05/26 11:31:45.123456
--------------------------------------------------------------------------------
```

datediff(From, to)

입력된 두 date 타입 값 (From ~ To) 사이의 간격을 초 단위로 계산하여 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select datediff( to_date('20201231', 'yyyymmdd'), to_date('20200101', '
yyyymmdd') ) from dual;
--------------------------------------------------------------------------------
DATEDIFF             : 31536000
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select datediff( to_date('20201231', 'yyyymmdd'), to_date('20200101', '
yyyymmdd') ) / (60 * 60 * 24) from dual;
--------------------------------------------------------------------------------
DIVIDE               : 365
--------------------------------------------------------------------------------
1 row selected
```

dump(value, [base])

지정된 문자열 value를 byte 단위로 변환하여 ascii 값으로 표현한 text를 반환한다.

- Base를 별도로 지정하지 않을 경우, default로 10 진수 ascii를 사용한다.
- 16 진수 변환도 지원한다.

사용 예

```
dbmMetaManager(DEMO)> select c1 from t1;
--------------------------------------------------------------------------------
C1                   : a
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select dump(c1, 16) from t1;
--------------------------------------------------------------------------------
DUMP                 : len=1: 61
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select dump(c1, 10) from t1;
--------------------------------------------------------------------------------
DUMP                 : len=1: 97
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select dump(c1) from t1;
--------------------------------------------------------------------------------
DUMP                 : len=1: 97
```

```
--------------------------------------------------------------------------------
1 row selected
```

Hex (Value)

인자로 주어진 문자열 value를 hex code로 변환하여 출력한다.

사용 예

```
dbmMetaManager(DEMO)> select hex( 'abc' ) from dual;
--------------------------------------------------------------------------------
HEX                  : 616263
--------------------------------------------------------------------------------
1 row selected
```

concat(target, append)

target 문자열 뒤에 append 문자열을 이어 붙여 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select concat( 'abc', 'def') from dual;
--------------------------------------------------------------------------------
CONCAT               : abcdef
--------------------------------------------------------------------------------
1 row selected
```

instr(source, keyword, [start, #appearance])

Source 문자열 내에서 keyword를 찾아 위치를 반환한다.
start가 생략되면 문자열의 처음부터 탐색하며, 음수이면 역순으로 검색한다.
start가 0 인 경우, keyword와 상관없이 항상 0을 반환한다.
#appearance는 keyword가 source 내에 출현한 횟수에 해당하는 위치를 탐색한다.

사용 예

```
dbmMetaManager(DEMO)> select instr( 'abcdef abcdef', 'def' ) from dual;
--------------------------------------------------------------------------------
INSTR                : 4
--------------------------------------------------------------------------------
1 row selected
```

```
dbmMetaManager(DEMO)> select instr( 'abcdef abcdef', 'def', 1, 2 ) from dual;
--------------------------------------------------------------------------------
INSTR                : 11
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select instr('abcd abcd abcd', 'bc', -5, 2) from dual;
--------------------------------------------------------------------------------
INSTR                : 2
--------------------------------------------------------------------------------
1 row selected
```

replace(source, keyword, replace)

Source 문자열에서 지정한 keyword를 찾아 replace 문자열로 대체한다.

사용 예

```
dbmMetaManager(DEMO)> select replace( 'abc, abc ing', 'abc', 'xxxxx' ) from dual;
--------------------------------------------------------------------------------
REPLACE              : xxxxx, xxxxx ing
--------------------------------------------------------------------------------
1 row selected
```

substr(source, start_position, count)

Source 문자열의 start_position 위치에서 시작하여, count 길이만큼 잘라 반환한다.

- Count가 생략되면, start_position 위치부터 문자열 끝까지 반환한다.
- Start_position이 source 문자열 길이를 초과하면 NULL을 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select substr( 'abcdefghi', 4) from t1;
--------------------------------------------------------------------------------
SUBSTR               : defghi
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select substr( 'abcdefghi', 9, 5) from t1;
--------------------------------------------------------------------------------
SUBSTR               : i
--------------------------------------------------------------------------------
```

```
1 row selected
dbmMetaManager(DEMO)> select dump(substr('abc', 5, 2)) from dual;
--------------------------------------------------------------------------------
DUMP : len=0:
--------------------------------------------------------------------------------
1 row selected
```

length(source)

Source 문자열의 길이를 반환한다.
문자열 중간에 NULL 값이 존재하면, NULL이 나타난 위치까지의 길이만 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select length( 'abcdefg' ) from dual;
--------------------------------------------------------------------------------
LENGTH : 7
--------------------------------------------------------------------------------
1 row selected
```

ltrim / rtrim(source)

Source 문자열의 왼쪽 (ltrim) 또는 오른쪽 (rtrim)에 있는 공백 문자를 제거하고 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select ltrim( '   xyz' ) from dual;
--------------------------------------------------------------------------------
LTRIM               : xyz
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select rtrim( 'xyz   ' ) from dual;
--------------------------------------------------------------------------------
RTRIM               : xyz
--------------------------------------------------------------------------------
1 row selected
```

lpad / rpad(source, size, padding_string)

Source 문자열의 왼쪽 (lpad) 또는 오른쪽 (rpad)에 지정한 padding_string을 추가하여, 최종 문자열 길이가 size가 되도록 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select lpad('aa', 10, 'x') from dual;
--------------------------------------------------------------------------------
LPAD                 : xxxxxxxxaa
--------------------------------------------------------------------------------
dbmMetaManager(DEMO)> select rpad('aa', 10, 'x') from dual;
--------------------------------------------------------------------------------
RPAD                 : aaxxxxxxxx
--------------------------------------------------------------------------------
1 row selected
```

abs(value)

입력한 숫자 value의 절대값을 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select abs(-123) from dual;
--------------------------------------------------------------------------------
ABS                  : 123
--------------------------------------------------------------------------------
1 row selected
```

mod(value1, value2)

value1을 value2로 나눈 나머지 값을 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select mod(14, 3) from dual;
--------------------------------------------------------------------------------
MOD                  : 2
--------------------------------------------------------------------------------
1 row selected
```

ceil(value)

Value 보다 큰 가장 가까운 정수를 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select ceil( 12.345) from dual;
--------------------------------------------------------------------------------
CEIL                : 13
--------------------------------------------------------------------------------
1 row selected
```

floor(value)

Value 보다 작은 가장 가까운 정수를 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select floor(13.678) from dual;
--------------------------------------------------------------------------------
FLOOR               : 13
--------------------------------------------------------------------------------
1 row selected
```

round(value)

Value를 반올림한 결과를 정수로 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select round(12.345) from dual;
--------------------------------------------------------------------------------
ROUND               : 12
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select round(12.678) from dual;
--------------------------------------------------------------------------------
ROUND               : 13
--------------------------------------------------------------------------------
```

trunc(value, [pos])

Value가 소수점일 경우, 지정된 pos 위치에서 소수점 이하를 잘라내고 반환한다.
pos가 지정되지 않으면, 소수점 이하를 모두 제거하고 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select trunc(13.34567) from dual;
--------------------------------------------------------------------------------
TRUNC                : 13
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select trunc(13.34567, 2) from dual;
--------------------------------------------------------------------------------
TRUNC : 13.34
--------------------------------------------------------------------------------
```

random(from, to)

from과 to로 지정된 범위 내에서 random 정수를 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select random(1, 10) from dual;
--------------------------------------------------------------------------------
RANDOM               : 2
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select random(100, 200) from dual;
--------------------------------------------------------------------------------
RANDOM               : 144
--------------------------------------------------------------------------------
1 row selected
```

nextval

지정한 sequence의 다음 값을 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select seq1.nextval from dual;
--------------------------------------------------------------------------------
NEXTVAL              : 2
--------------------------------------------------------------------------------
1 row selected
```

currval

지정한 sequence의 현재 값을 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select seq1.currval from dual;
--------------------------------------------------------------------------------
CURRVAL              : 2
--------------------------------------------------------------------------------
1 row selected
```

> 노트
> NEXTVAL이 호출되지 않은 상태에서 sequence 객체에 대해 currval을 호출하면 오류가 발생한다.

Digest (Value, SHA-type)

입력된 문자열 value를 지정한 SHA-type으로 단방향 해시 암호화하며, 결과는 바이너리(binary) 형태로 반환된다.

- Digest 처리된 결과는 binary 형태이므로 화면에 정상적으로 출력되지 않을 수 있다.
- 이 경우 다음과 같이 hex 함수 등을 사용하여 결과를 확인할 수 있다.

사용 예

```
dbmMetaManager(DEMO)> select hex( digest( 'my password', 'SHA256' ) ) from dual;
--------------------------------------------------------------------------------
HEX                  : BB14292D91C6D0920A5536BB41F3A50F66351B7B9D94C804DFCE8A96CA1051F2
--------------------------------------------------------------------------------
1 row selected
```

Min(Column)

Select 결과 집합에서 min 함수에 정의된 column 값 중 가장 작은 값을 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select * from t1 where c1 = 1;
-------------------------------------------------------------------------------
C1                    : 1
C2                    : -1
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 1
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 2
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 3
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 4
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 5
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 6
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 7
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 8
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 9
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 10
-------------------------------------------------------------------------------
11 row selected
```

```
dbmMetaManager(DEMO)> select min(c2) from t1 where c1 = 1;
--------------------------------------------------------------------------------
MIN_VALUE            : -1.000000000
--------------------------------------------------------------------------------
1 row selected
```

Max(Column)

Select 결과 집합에서 max 함수에 정의된 column 값 중 가장 큰 값을 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select * from t1 where c1 = 1;
--------------------------------------------------------------------------------
C1                   : 1
C2                   : -1
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 1
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 2
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 3
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 4
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 5
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 6
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 7
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 8
--------------------------------------------------------------------------------
```

```
C1                    : 1
C2                    : 9
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 10
-------------------------------------------------------------------------------
11 row selected
dbmMetaManager(DEMO)> select max(c2) from t1 where c1 = 1;
-------------------------------------------------------------------------------
MAX_VALUE             : 10.000000000
-------------------------------------------------------------------------------
1 row selected
```

Avg(Column)

AVG 함수에 지정된 column 값의 평균값을 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select * from t1 where c1 = 1;
-------------------------------------------------------------------------------
C1                    : 1
C2                    : -1
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 1
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 2
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 3
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 4
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 5
-------------------------------------------------------------------------------
C1                    : 1
C2                    : 6
```

```
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 7
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 8
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 9
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 10
--------------------------------------------------------------------------------
11 row selected
dbmMetaManager(DEMO)> select avg(c2) from t1 where c1 = 1;
--------------------------------------------------------------------------------
AVG                 : 4.909090909
--------------------------------------------------------------------------------
1 row selected
```

Sum(Column)

Sum 함수에 지정된 column 값의 합계를 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select * from t1 where c1 = 1;
--------------------------------------------------------------------------------
C1                  : 1
C2                  : -1
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 1
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 2
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 3
--------------------------------------------------------------------------------
C1                  : 1
```

```
C2                  : 4
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 5
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 6
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 7
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 8
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 9
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 10
--------------------------------------------------------------------------------
11 row selected
dbmMetaManager(DEMO)> select sum(c2) from t1 where c1 = 1;
--------------------------------------------------------------------------------
SUM                 : 54.000000000
--------------------------------------------------------------------------------
1 row selected
```

Decode(cond_expr, case_cond, value_expr, ..., [else_expr])

cond_expr 값과 일치하는 case_cond 뒤에 지정된 value를 반환한다.

- 일치하는 case_cond가 없을 경우, else_expr이 존재하면 해당 값을 반환하고, 그렇지 않으면 길이가 0인 문자열을 반환한다.
- cond_expr과 case_cond의 데이터 타입은 동일해야 하며, double 또는 문자열을 사용할 수 있다.
- 반환되는 value_expr과 else_expr은 문자열 데이터 타입이다.

사용 예

```
dbmMetaManager(DEMO)> create table t1 (c1 int, c2 int);
success
dbmMetaManager(DEMO)> insert into t1 values (1, 10);
success
```

```
dbmMetaManager(DEMO)> insert into t1 values (2, 20);
success
dbmMetaManager(DEMO)> insert into t1 values (3, 30);
success
dbmMetaManager(DEMO)> commit;
success
dbmMetaManager(DEMO)> select c1, c2, decode( c1, 1, 'a', 2, 'b', 3, 'c', 'k' ) from t1;
--------------------------------------------------------------------------------
C1                  : 1
C2                  : 10
DECODE              : a
--------------------------------------------------------------------------------
C1                  : 2
C2                  : 20
DECODE              : b
--------------------------------------------------------------------------------
C1                  : 3
C2                  : 30
DECODE              : c
--------------------------------------------------------------------------------
3 row selected
```

Upper(expr)

주어진 문자열 expr이 text 인 경우, 이를 대문자로 변환하여 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select upper( 'aaaaabzc' ) from dual;
--------------------------------------------------------------------------------
UPPER               : AAAAABZC
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select upper( 123 ) from dual;
--------------------------------------------------------------------------------
UPPER               : 123
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select upper( 'a1b1C34' ) from dual;
--------------------------------------------------------------------------------
UPPER               : A1B1C34
```

```
--------------------------------------------------------------------------------
1 row selected
```

Lower(expr)

주어진 문자열 expr이 text인 경우, 이를 소문자로 변환하여 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select lower( 'AAAAABZC' ) from dual;
--------------------------------------------------------------------------------
LOWER                : aaaaabzc
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select lower( 123 ) from dual;
--------------------------------------------------------------------------------
LOWER                : 123
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select lower( 'A1B1c34' ) from dual;
--------------------------------------------------------------------------------
LOWER                : a1b1c34
--------------------------------------------------------------------------------
1 row selected
```

NVL(orgnExpr, valueExpr)

주어진 문자열 orgnExpr의 첫 번째 byte가 NULL인 경우, ValueExpr로 변환된 값을 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select nvl( 'x', 'not_null') from dual;
--------------------------------------------------------------------------------
NVL                  : x
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select nvl( '', 'isnull') from dual;
--------------------------------------------------------------------------------
NVL                  : isnull
--------------------------------------------------------------------------------
1 row selected
```

JSON_STRING(Column_Name_List)

일반 테이블을 대상으로, 전체 column 또는 지정한 column 목록에 해당하는 데이터만 JSON 형식 string으로 변환하여 반환한다.

사용 예

```
dbmMetaManager(DEMO)> create table t1 (c1 int, c2 int, c3 char(20) );
success
dbmMetaManager(DEMO)> insert into t1 values (1, 2, 'xyz1' );
success
dbmMetaManager(DEMO)> insert into t1 values (100, 200, 'zyx2' );
success
dbmMetaManager(DEMO)> insert into t1 values (10, 20, 'ppppppppp1' );
success
dbmMetaManager(DEMO)> commit;
success
dbmMetaManager(DEMO)> select json_string(*) from t1;
--------------------------------------------------------------------------------
JSON_STRING          : { "C1": "1", "C2": "2", "C3": "xyz1" }
--------------------------------------------------------------------------------
JSON_STRING          : { "C1": "100", "C2": "200", "C3": "zyx2" }
--------------------------------------------------------------------------------
JSON_STRING          : { "C1": "10", "C2": "20", "C3": "ppppppppp1" }
--------------------------------------------------------------------------------
3 row selected
dbmMetaManager(DEMO)> select json_string(c1, c2, c3) from t1;
--------------------------------------------------------------------------------
JSON_STRING          : { "C1": "1", "C2": "2", "C3": "xyz1" }
--------------------------------------------------------------------------------
JSON_STRING          : { "C1": "100", "C2": "200", "C3": "zyx2" }
--------------------------------------------------------------------------------
JSON_STRING          : { "C1": "10", "C2": "20", "C3": "ppppppppp1" }
--------------------------------------------------------------------------------
3 row selected
```

노트

json_string 에는 GOLDILOCKS LITE에서 지원하는 데이터 타입만 사용할 수 있으며, json_string 타입 자체를 새로 추가할 수는 없다.

USER_TYPE(Column_Name, Type_Name)

Column_Name을 지정한 user type (Type_Name) 으로 캐스팅하여 값을 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select * from t2;
--------------------------------------------------------------------------------
C1 : 100
C2 : 200
C3 :
--------------------------------------------------------------------------------
C1 : 200
C2 : 300
C3 :
--------------------------------------------------------------------------------
2 row selected
dbmMetaManager(DEMO)> select c1, c2, user_type(c3, u2) from t2;
--------------------------------------------------------------------------------
C1        : 100
C2        : 200
USER_TYPE : C1=-100 C2=-200
--------------------------------------------------------------------------------
C1        : 200
C2        : 300
USER_TYPE : C1=-200 C2=-300
--------------------------------------------------------------------------------
dbmMetaManager(DEMO)> select user_type(c3, u2) from t2 where user_type(c3, u2.c1 ) = -100;
--------------------------------------------------------------------------------
USER_TYPE : C1=-100 C2=-200
--------------------------------------------------------------------------------
1 row selected
```

1.4 DICTIONARY

Dictionary는 instance가 생성될 때 모든 segment의 메타 정보를 관리하기 위해 생성되는 built-in table과 view의 집합이다.

- DICT (initdb 실행 시 생성)
  - DICT_INST: Instance meta 정보 관리
- 사용자 instance (create instance 실행 시 생성)
  - 사용자 table 등과 같은 meta 정보 관리
    - DIC_TABLE
    - DIC_INDEX
    - DIC_COLUMN
    - DIC_INDEX_COLUMN
    - DIC_SEQUENCE
    - DIC_REPL_INST
    - DIC_REPL_TABLE
  - Information View
    - V$INSTANCE
    - V$SESSION
    - V$TRANSACTION
    - V$SYS_STAT
    - V$SESS_STAT
    - V$TABLE_USAGE
    - V$REPL_STAT
    - V$LOG_STAT

노트

- Dictionary table은 DDL/ DML 작업을 허용하지 않는다.
- 또한, 테이블 목록에는 표시되지만 별도의 설명이 없는 dictionary table은 향후 deprecated 될 예정이다.

DICTIONARY TABLES

DIC_INST

DIC_INST는 DICT 인스턴스 안에 생성되며, 사용자 인스턴스 관련 정보를 저장한다.

Column name	설명
INST_NAME	Instance name
INIT_SIZE	생성 시 할당되는 초기 크기 (단위: undo page 개수)
EXTEND_SIZE	공간이 부족할 때 확장되는 크기
MAX_SIZE	확장 가능한 최대 크기

DIC_TABLE

DIC_TABLE은 table 관련 정보를 저장한다.

Column name	설명
INST_NAME	Instance name
TABLE_NAME	Table name
TABLE_TYPE	TABLE 유형 • 1: TABLE • 2: QUEUE • 3: STORE • 4: SEQUENCE • 5: DIRECT • 7: SPLAY • 10: USER_TYPE
COLUMN_COUNT	TABLE을 구성하는 column의 개수
ROW_SIZE	TABLE에 저장되는 레코드 크기
LOCK_MODE	1: 동시성 제어모드 (0: deprecate 예정)
MSG_SIZE	QUEUE TABLE인 경우 message의 최대 크기
INDEX_COUNT	현재 테이블에 생성된 INDEX 개수
INIT_SIZE	생성 시 초기 segment의 크기 (단위: 레코드 개수)
EXTEND_SIZE	공간 확장 시 segment의 크기 (단위: 레코드 개수)
MAX_SIZE	최대 확장 가능한 segment의 크기 (단위: 레코드 개수)
INDEX_ID	Index 생성 시점마다 부여되는 index 고유 번호 채번 용도
ONLY_UPDATE_SELECT_MODE	테이블을 생성할 때 단일 프로세스만 접근할 수 있게 설정한 경우 (deprecate 예정)
CREATE_SCN	테이블 생성 시점의 SCN (system commit number)

DIC_COLUMN

DIC_COLUMN은 table을 구성하는 column 정보를 저장한다.

Column name	설명
INST_NAME	Instance name
TABLE_NAME	Table name
COLUMN_NAME	Column name
USER_TYPE_NAME	User type으로 지정된 경우 해당 type name
DATA_TYPE	Column의 데이터 타입 • 1: short • 2: int • 3: double • 4: float • 5: long • 6: char • 7: date • 15: USER_TYPE
COLUMN_OFFSET	레코드 전체 영역 중에 column이 저장되는 위치
COLUMN_SIZE	Column의 데이터 크기
COLUMN_ORDER	Column 순서

DIC_INDEX

DIC_INDEX는 table에 생성한 index 정보를 저장한다.

Column name	설명
INST_NAME	Instance name
TABLE_NAME	Table name
INDEX_NAME	Index name
IS_UNIQUE	Unique 여부 • 1: unique • 0: non-unique
KEY_SIZE	Key column 크기의 합
KEY_COLUMN_COUNT	Key column의 총 개수
INDEX_ORDER	Index가 생성된 순서

DIC_INDEX_COLUMN

DIC_INDEX_COLUMN은 index를 구성하는 key column 정보를 저장한다.

Column name	설명
INST_NAME	Instance name
TABLE_NAME	Table name
INDEX_NAME	Index name

Column name	설명
COLUMN_NAME	Column name
ID	Index key column 의 고유 번호
COLUMN_JSON	json path
JSON_KEY_SIZE	json path 내의 key 크기
KEY_COLUMN_ORDER	Key column의 나열 순서 (Ordering 순서를 의미한다.)
COLUMN_ORDER	테이블 내의 column 순서
IS_ASC	오름차순 여부 (1: ASC, 0: DESC)

DIC_SEQUENCE

DIC_SEQUENCE는 sequence object를 구성하는 정보를 저장한다.

Column name	설명
INST_NAME	Instance name
SEQUENCE_NAME	Sequence name
START_VALUE	초기값
INCREMENT_VALUE	증가값
CURRENT_VALUE	미사용 column
MIN_VALUE	MinValue
MAX_VALUE	MaxValue
IS_CYCLE	• 1: CYCLE • 0: NOCYCLE

DIC_REPL_INST

DIC_REPL_INST 는 create replication 명령 실행 시 해당 instance 이름을 저장한다.

Column name	설명
INST_NAME	Instance name

DIC_REPL_TABLE

DIC_REPL_TABLE 은 replication 대상 테이블 정보를 저장한다.

Column name	설명
INST_NAME	Instance name
TABLE_NAME	Table name

Information View

GOLDILOCKS LITE의 각종 정보를 table 형태로 제공하는 view이다.

노트

- Information view는 DDL/ DML을 허용하지 않는다.
- 또한, 일부 reserved column은 향후 기능 확장을 위한 항목으로, 현재는 출력 값에 의미가 없다.

V$INSTANCE

현재 instance의 정보를 출력한다.

Column 이름	설명
CURR_SCN	현재 instance의 SCN 값
MIN_SCN	reserved
CURR_MIN_SCN	reserved
CURR_MIN_SCN_TRANS	reserved
ACTIVE_MODE	reserved
DISK_ENABLE	Disk LogFile 설정 여부
LOGFILE_SIZE	Disk LogFile 한 개의 크기
LOGCACHE_MODE	LogCache 설정 정보
LOGCACHE_CHUNK_SIZE	LogCache chunk 한 개의 크기
LOGCACHE_CHUNK_COUNT	LogCache chunk 최대 개수
LOGCACHE_RANGE	LogCache chunk의 사용 범위
CREATE_TIME	Instance를 생성한 시각

```
dbmMetaManager(DEMO)> select * from v$instance;
--------------------------------------------------------------------------------
SCN                 : 11
MIN_SCN             : 11
MIN_SCN_TRANS_ID    : 1
ACTIVE_MODE         : 1
DISK_ENABLE         : 0
LOGFILE_SIZE        : 0
LOGCACHE_MODE       : no cache mode
LOGCACHE_CHUNK_SIZE : 0
LOGCACHE_CHUNK_COUNT : 1
```

```
LOGCACHE_RANGE       : 0
CREATE_TIME          : 2020-03-25 12:57:02
--------------------------------------------------------------------------------
```

V$SESSION

현재 instance를 사용 중인 세션의 정보를 출력한다.

Column 이름	설명
ID	Session의 고유 ID 이다.
TRANS_ID	Session이 보유한 트랜잭션 식별 번호 이다.
PID	현재 Trans Id를 점유하고 있는 OS process ID 이다.
TID	현재 Trans Id를 점유하고 있는 OS thread ID 이다.
OLD_TID	비정상 종료 시점에 할당된 OS thread ID 이다.
CURR_UNDO_PAGE	트랜잭션 진행 중 현재 사용되는 undo page의 ID 이다.
FIRST_UNDO_PAGE	트랜잭션 진행 중에 사용된 첫 번째 undo page의 ID 이다.
LAST_UNDO_PAGE	트랜잭션 진행 중에 사용된 마지막 undo page의 ID 이다.
SAVEPOINT_UNDO_PAGE	Reserved
SAVEPOINT_UNDO_OFFSET	Reserved
WAIT_TRANS_ID	Lock 대기 중일 경우 LOCK을 점유한 상대 Trans ID 이다.
WAIT_OBJECT	Lock 대기 중일 경우 대상 테이블 이름이다.
WAIT_SLOT_ID	Lock 대기 중일 경우 데이터 공간의 고유 ID 이다.
SESSION_STATUS	트랜잭션의 현재 상태이다. • transaction: Transaction을 시작할 수 있거나 진행 중이다. • commit start: Commit이 시작되었다. • commit completed: Memory 까지 commit이 완료되었다. • rollback completed: Rollback이 완료되었다. • recovery completed: 비정상 종료로 인한 recovery가 완료되었다.
IS_LOGGING	트랜잭션의 메모리 logging mode 이다. (0: No Logging, 1: Logging) No logging으로 설정할 경우 트랜잭션을 복구하지 않는다. (Rollback이 불가능하다.)
LOGFILE_NO	현재 세션이 디스크 병렬 로깅 모드로 동작 중일 때, 세션이 사용 중인 logfile sequence 이다.
CKPT_NO	dbmCkpt에 의해 데이터 파일에 반영된 logfile sequence 이다.
IS_REPL	Reserved
REPL_SEND_SCN	세션이 마지막으로 전송한 이중화 트랜잭션의 SCN 이다.
REPL_RECV_ACK_SCN	세션이 상대편으로부터 반영 완료 통지를 받은 마지막 트랜잭션의 SCN 이다.
AUTOCOMMIT_MODE	AutoCommit Mode (0: Non-AutoCommit, 1:Auto-Commit)
BEGIN_TIME	세션 접속 시각이다.
PROGRAM	프로그램 이름이다.
REMOTE_PID	원격 접속 시 원격 서버의 프로세스 ID 이다.
REMOTE_ADDR	원격 접속 시 원격 서버 주소이다.
REMOTE_PROGRAM	원격 접속 시 원격 서버에서 구동 된 프로그램 이름이다.

```
dbmMetaManager(DEMO)> select * from v$session;
--------------------------------------------------------------------------------
TRANS_ID             : 1
PID                  : 35612
TID                  : 35612
OLD_TID              : 35612
CURR_UNDO_PAGE       : 6
FIRST_UNDO_PAGE      : 6
LAST_UNDO_PAGE       : 11
SAVEPOINT_UNDO_PAGE  : -1
SAVEPOINT_UNDO_OFFSE : -1
WAIT_TRANS_ID        : -1
WAIT_OBJECT          :
WAIT_SLOT_ID         : -1
STATUS               : transaction ready or running
IS_REPL              : 0
BEGIN_TIME           : 2024/10/18 08:46:47
PROGRAM              : dbmListener
REMOTE_PID           : 0001513772
REMOTE_ADDR          : 192.168.0.26
REMOTE_PROGRAM       : dbmMetaManager
--------------------------------------------------------------------------------
```

V$TRANSACTION

각 세션의 트랜잭션 정보를 확인할 수 있다.

Column 이름	설명
TRANS_ID	Transaction의 고유 번호
TRANS_SEQ	트랜잭션 내에서 발생한 순서
TRANS_TYPE	트랜잭션 유형
OBJECT_NAME	대상 테이블 이름
SLOT_ID	데이터 공간의 고유 ID
EXTRA_KEY	데이터 공간 내부 관리를 위한 고유 ID
COMMIT_FLAG	트랜잭션의 memory commit 여부 (1: Commit)
SKIP_FLAG	트랜잭션이 commit 되지 않도록 설정된 flag (1: Skip)
VALID_FLAG	트랜잭션 로그의 유효성 (1: 정상)

V$LOG_STAT

Disk log와 관련된 각종 설정 정보를 조회할 수 있는 view이다.

Column 이름	설명
DISKLOG_ENABLE	Disk logging 설정 여부이다.
CACHE_MODE	LOG CACHE MODE 설정값이다. (0: 병렬 로깅, 1: Cache, 2: NVDIMM)
DIRECT_IO_ENABLE	DIRECT IO 활성화 여부이다.
ARCHIVE_ENABLE	Archive 설정 여부이다.
CURR_FILE_NO	Reserved (Log cache 모드가 활성화 된 경우에만 의미가 있다.)
CURR_FILE_OFFSET	Reserved
LAST_CKPT_FILE_NO	CheckPoint가 시작될 logfile의 번호 이다. (Log cache 모드가 활성화 된 경우에만 의미가 있다.)
LAST_ARCHIVE_FILE_NO	Archive가 시작될 대상 logfile의 번호이다. (Log cache 모드가 활성화 된 경우에만 의미가 있다.)
LAST_CAPTURE_FILE_NO	Reserved
LOGCACHE_WRITE_IND	LogCache 영역 내에서 트랜잭션이 기록을 수행할 수 있는 현재 위치이다.
LOGCACHE_READ_IND	LogCache 영역 내에서 flusher가 데이터를 읽을 현재 위치이다.
FLUSHER_FILE_NO	Flushser가 마지막으로 기록한 logfile의 번호이다. (Log cache 모드가 활성화 된 경우에만 의미가 있다.)
FLUSHER_FILE_OFFSET	Flushser가 마지막으로 기록한 logfile의 offset 이다.
LOG_DIR	Disk LogFile이 저장되는 디렉토리 경로이다.
DATAFILE_DIR	dataFile이 저장되는 디렉토리 경로이다.
ARCHIVE_DIR	Checkpoint 시점에 archiving 되는 logfile이 저장되는 디렉토리 경로이다.

```
dbmMetaManager(DEMO)> select * from v$log_stat;
--------------------------------------------------------------------------------
DISKLOG_ENABLE       : 0
CACHE_MODE           : 0
DIRECT_IO_ENABLE     : 0
ARCHIVE_ENABLE       : 0
CURR_FILE_NO         : -1
CURR_FILE_OFFSET     : -1
LAST_CKPT_FILE_NO    : -1
LAST_ARCHIVE_FILE_NO : -1
LAST_CAPTURE_FILE_NO : -1
LOGCACHE_WRITE_IND   : -1
LOGCACHE_READ_IND    : -1
FLUSHER_FILE_NO      : -1
FLUSHER_FILE_OFFSET  : -1
LOG_DIR              : /home/majaehwa/work/new_lite/pkg/wal
```

```
DATAFILE_DIR          : /home/majaehwa/work/new_lite/pkg/dbf
ARCHIVE_DIR           : /home/majaehwa/work/new_lite/pkg/arch
--------------------------------------------------------------------------------
1 row selected
```

V$REPL_STAT

Replication 환경에서 이중화 상태 정보를 보여주는 view 이다.

Column 이름	설명
TARGET_IP	Slave IP 주소이다.
TARGET_PORT	Slave port 번호 이다.
LISTEN_PORT	Slave 측의 dbmReplica listen port number 이다.
SEND_SCN	Master 측에서 전송한 SCN (System Commit Number) 이다.
RECV_SCN	Master 측에서 전송한 SCN 중에 마지막으로 수신한 Ack SCN 이다.
UNSENT_START_FILENO	reserved
UNSENT_END_FILENO	reserved

```
dbmMetaManager(DEMO)> select * from v$repl_stat;
--------------------------------------------------------------------------------
TARGET_IP           : 127.0.0.1
TARGET_PORT         : 29002
LISTEN_PORT         : 29002
SEND_SCN            : -1
RECV_SCN            : -1
UNSENT_START_FILENO : -1
UNSENT_END_FILENO   : -1
--------------------------------------------------------------------------------
1 row selected
```

V$TABLE_USAGE

Instance 내에 생성된 모든 테이블의 사용량 정보를 보여주는 view 이다.

- SIZE는 테이블에 저장된 row의 개수를 가리킨다.

Column 이름	설명
OBJECT_NAME	테이블의 이름이다.
MAX_SIZE	해당 테이블이 확장될 수 있는 최대 크기를 나타낸다.
TOTAL_SIZE	테이블에 현재 할당된 전체 크기를 나타낸다.
USED_SIZE	확장된 크기 중 실제로 사용 중인 공간의 크기를 나타낸다.
FREE_SIZE	최대 확장 크기 기준으로 남아 있는 여유 공간의 크기를 나타낸다.

Column 이름	설명
USED_MEM	내부 헤더를 포함한 테이블의 사용량을 byte 단위로 환산한 크기를 나타낸다.

```
dbmMetaManager(DEMO)> select * from v$table_usage;
------------------------------------------------------------------------------
OBJECT_NAME : T1
MAX_SIZE    : 4096000
TOTAL_SIZE  : 1024
USED_SIZE   : 0
FREE_SIZE   : 1024
USED_MEM    : 90488
------------------------------------------------------------------------------
1 row selected
```

V$SYS_STAT

Instance에서 발생한 각 유형별 수행 횟수에 대한 정보를 제공한다.

Column 이름	설명
NAME	누적된 수행 유형의 이름
ACCUM_COUNT	해당 유형의 누적 수행 횟수

```
dbmMetaManager(DEMO)> select * from v$sys_stat
------------------------------------------------------------------------------
NAME                : init_handle_op
ACCUM_COUNT         : 2
------------------------------------------------------------------------------
NAME                : free_handle_op
ACCUM_COUNT         : 1
------------------------------------------------------------------------------
NAME                : prepare_op
ACCUM_COUNT         : 20
------------------------------------------------------------------------------
NAME                : execute_op
ACCUM_COUNT         : 20
------------------------------------------------------------------------------
NAME                : insert_op
ACCUM_COUNT         : 2
------------------------------------------------------------------------------
NAME                : update_op
```

```
ACCUM_COUNT          : 1
------------------------------------------------------------------------------
NAME                 : delete_op
ACCUM_COUNT          : 1
------------------------------------------------------------------------------
NAME                 : scan_op
ACCUM_COUNT          : 3
------------------------------------------------------------------------------
NAME                 : enqueue_op
ACCUM_COUNT          : 0
------------------------------------------------------------------------------
NAME                 : dequeue_op
ACCUM_COUNT          : 0
------------------------------------------------------------------------------
NAME                 : aging_op
ACCUM_COUNT          : 0
------------------------------------------------------------------------------
NAME                 : commit_op
ACCUM_COUNT          : 11
------------------------------------------------------------------------------
NAME                 : rollback_op
ACCUM_COUNT          : 1
------------------------------------------------------------------------------
NAME                 : recovery_rollback_op
ACCUM_COUNT          : 0
------------------------------------------------------------------------------
NAME                 : recovery_commit_op
ACCUM_COUNT          : 0
------------------------------------------------------------------------------
```

누적 항목은 아래 표와 같으며 v$sess_stat의 항목도 이와 동일하다. 각 항목에 누적된 수치는 성공/실패 여부와 관계없이 내부 처리 과정에 진입한 횟수를 의미한다.

누적 항목	설명
init_handle_op	D/A mode로 instance에 접속한 횟수
free_handle_op	Instance에서 해제된 횟수
prepare_op	Prepare statement가 호출된 횟수
execute_op	Execute statement가 호출된 횟수
insert_op	Insert가 수행된 횟수
update_op	Update가 수행된 횟수
scan_op	Select를 포함하여 대상을 scan 하는 모든 횟수

누적 항목	설명
delete_op	Delete가 수행된 횟수
enqueue_op	Enqueue가 수행된 횟수
dequeue_op	Dequeue가 수행된 횟수
aging_op	참조되지 않는 공간이 회수된 횟수
commit_op	Commit이 호출된 횟수
rollback_op	Rollback이 호출된 횟수
recovery_rollback_op	비정상 복구 (rollback과 동일한 과정)를 수행한 횟수
recovery_commit_op	비정상 복구 (기존에 commit 된 row에 대한 lock을 해제하는 과정)를 수행한 횟수

노트

DBM_PERF_ENALBE 속성이 활성화된 경우에만 정보가 누적된다.

V$SESS_STAT

Instance 생성 이후, 세션별로 발생한 각 유형의 수행 횟수를 누적하여 출력한다.

Column 이름	설명
TRANS_ID	세션의 고유번호
STAT_NAME	누적된 수행 유형의 이름
ACCUM_COUNT	해당 유형의 누적 수행 횟수

```
dbmMetaManager(DEMO)> select * from v$sess_stat
--------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : init_handle_op
ACCUM_COUNT          : 2
--------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : free_handle_op
ACCUM_COUNT          : 1
--------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : prepare_op
ACCUM_COUNT          : 21
--------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : execute_op
```

```
ACCUM_COUNT          : 21
-------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : insert_op
ACCUM_COUNT          : 2
-------------------------------------------------------------------------------
TRANS_ID             : 1633972341
NAME                 : te_op
ACCUM_COUNT          : 0
-------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : delete_op
ACCUM_COUNT          : 1
-------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : scan_op
ACCUM_COUNT          : 3
-------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : enqueue_op
ACCUM_COUNT          : 0
-------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : dequeue_op
ACCUM_COUNT          : 0
-------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : aging_op
ACCUM_COUNT          : 0
-------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : commit_op
ACCUM_COUNT          : 11
-------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : rollback_op
ACCUM_COUNT          : 1
-------------------------------------------------------------------------------
TRANS_ID             : 1
NAME                 : recovery_rollback_op
ACCUM_COUNT          : 0
```

```
--------------------------------------------------------------------------------
TRANS_ID              : 1
NAME                  : recovery_commit_op
ACCUM_COUNT           : 0
--------------------------------------------------------------------------------
```

노트

해당 정보는 DBM_PERF_ENALBE 속성이 활성화 된 경우에만 누적된다.
또한, 이 값은 세션 접속 이후의 데이터가 아니라 instance 전체에 대해 누적된 값을 의미한다.
따라서 접속 시점 이후의 변동량을 분석하려면, before/after 방식으로 스냅샷을 조회한 뒤 이를 비교해야 한다.

1.5 dbmMetaManager

개요

dbmMetaManager는 DDL과 DML 작업을 수행하기 위한 utility이다.
사용 가능한 옵션은 아래 표와 같다.

입력 옵션	설명
-i <Instance name>	접근할 instance name을 지정한다.
-f <script file>	입력한 script file 에 포함된 SQL을 순차적으로 실행한 뒤 종료한다.
-p <process alias name>	dbmMetaManager 프로세스를 식별하기 위한 별칭을 지정한다.
-e	Instance password가 설정된 경우, 암호를 입력한다.
-v	제품 버전 정보를 출력한다.
-s	제품 사용 권한, 버전 정보 등의 출력 내용을 생략한다.
-a	세션 접근이 불가능한 상황에서도 하나의 접속을 허용한다.

```
[majaehwa@tech9 new_lite]$ dbmMetaManager -h
Usage] dbmMetaManager
-f : input a file-name to execute: -f <file name>
-p : specify a alias-name
-i : specify a instance name to attach: -i <instance name>
-e : specify a password
-v : print version info
-s : to silent option
-a : attach as admin-mode
-h : Display this information
```

Internal command는 SQL과 달리 dbmMetaManager 환경 내에서만 동작하는 명령이다.

Internal command	설명
initdb	최초로 DICTIONARY INSTANCE를 생성할 때 사용하는 명령이다.
list	현재 instance에 생성된 object 목록을 출력한다.
desc [table name]	입력된 테이블의 상세 정보를 출력한다.
h / hist / history	현재까지 실행한 명령의 목록을 출력한다. (최대 20개)
ed [number]	직전에 수행한 명령 또는 history에 저장된 목록 중에 입력된 번호의 명령을 편집한다. 환경 변수 DBM_EDITOR에 설정된 편집기가 구동된다. (기본은 vi 로 설정된다.)
/ [number]	직전에 수행한 명령 또는 history에 저장된 목록 중에 입력된 번호의 명령을 재수행한다.
q / quit / exit	dbmMetaManager를 종료한다. (수행한 트랜잭션은 모두 rollback 처리된다.)

Internal command	설명
set vertical [on | off]	결과가 column/ line 단위로 출력되도록 설정한다.
struct out [table name]	테이블의 형상을 C 구조체에 맞게 출력한다.
set instance [instance name]	Multi instance 환경에서 작업 대상 instance를 전환할 때 사용한다.
set password [old pwd] [new_pwd]	Instance의 old password를 new password로 변경한다. • 최초 설정 시, old password는 null 로 입력한다. • new password를 null 로 입력하면 암호가 해제된다. • Password 변경 시 old password는 기존 password와 일치해야 한다.
set index [table name] [index name]	입력된 테이블의 index를 사용하도록 설정한다.
startup [instance_name]	dbmCkpt에 의해 생성된 데이터 파일을 사용하여 instance를 복구한다.

노트

- dbmMetaManager의 internal command는 소문자로 입력해야 한다.
- SQL 방식의 처리 구조는 반환 대상인 모든 레코드를 임시 메모리에 할당하여 저장하므로, 처리 대상이 많은 경우 메모리 부족 등의 오류가 발생할 수 있다.
  이러한 경우에는 조회 (처리) 대상 레코드의 범위를 분할하여 나누어 처리해야 한다.

Internal Commands

Internal command는 dbmMetaManager에서만 실행할 수 있는 명령을 의미한다.

list

현재 접속된 instance 하위에 생성된 object 목록을 출력한다.

```
dbmMetaManager(DEMO)> list;
OBJECT                                    MAX       TOTAL        USED        FREE
================================================================================
DUAL                                   102400        1024           1        1023
REPL_LOG                             10000000        1024           0        1024
REPL_UNSENT                          10000000        1024           0        1024
SEQ1                                        1           1           0           1
success
```

노트

list 명령으로 출력된 결과 중에 DIRECT TABLE 유형은 구조적인 특성상 사용량을 계산할 수 없다.
해당 테이블의 실제 레코드 수가 필요한 경우에는 select count(*) 구문을 사용하여 직접 조회해야 한다.

desc

테이블 생성 정보를 화면에 출력한다.

```
dbmMetaManager(DICT)> desc dic_index_column;
-------------------------------------------------------------------------------
Instance=(DICT) Table=(DIC_INDEX_COLUMN) Type=(TABLE) RowSize=(136) LockMode(1)
-------------------------------------------------------------------------------
INST_NAME                       char                32          0
TABLE_NAME                      char                32          32
INDEX_NAME                      char                32          64
COLUMN_NAME                     char                32          96
KEY_COLUMN_ORDER                int                 4           128
COLUMN_ORDER                    int                 4           132
-------------------------------------------------------------------------------
IDX_DIC_INDEX_COLUMN            unique      (INST_NAME asc, TABLE_NAME asc, INDEX_NAME asc,
COLUMN_NAME asc, COLUMN_JSON asc, JSON_KEY_SIZE asc, KEY_COLUMN_ORDER asc)
-------------------------------------------------------------------------------
success
```

set index

테이블을 조회할 때 사용할 특정 index를 지정한다.

```
dbmMetaManager(DEMO)> create table t1 (c1 int, c2 int);
success
dbmMetaManager(DEMO)> create unique index idx1_t1 on t1 (c1);
success
dbmMetaManager(DEMO)> create unique index idx2_t1 on t1 (c2);
success
dbmMetaManager(DEMO)> insert into t1 values (1, 10);
success
dbmMetaManager(DEMO)> insert into t1 values (2, 20);
success
dbmMetaManager(DEMO)> select * from t1 where c2 = 20;
```

```
--------------------------------------------------------------------------------
C1                   : 2
C2                   : 20
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> set index t1 idx2_t1;
success
dbmMetaManager(DEMO)> select * from t1 where c2 = 20;
--------------------------------------------------------------------------------
C1                   : 2
C2                   : 20
--------------------------------------------------------------------------------
1 row selected
```

노트

GOLDILOCKS LITE는 별도의 SQL 최적화 기능을 제공하지 않는다.
INDEX는 SQL의 WHERE 절에 index key column이 모두 포함된 EQUAL 조건인 경우에만 사용되며, 그 외의 경우에는 모두 full scan 방식으로 처리된다.

set vertical [on/off]

dbmMetaManager의 조회 결과는 기본적으로 column 별로 한 줄씩 출력된다.
set vertical option을 사용하면 조회 결과를 레코드 단위로 하나씩 출력되도록 설정할 수 있다.

```
dbmMetaManager(DEMO)> set vertical off
success
dbmMetaManager(DEMO)> select * from t1
--------------------------------------------------------------------------------
             C1              C2 C3                                   C4
--------------------------------------------------------------------------------
              1               1 sunjesoft                             1
           1024            1024 hey hey hey                        1024
--------------------------------------------------------------------------------
2 row selected
```

OS Command 수행

dbmMetaManager에서 OS command를 실행해야 할 경우에는 명령 앞에 +를 붙여 입력한다.
자세한 사용 방법은 아래 예제를 참고한다.

```
dbmMetaManager(unknown)> + ls -lrt ${DBM_HOME};
합계 12
drwxrwxr-x. 2 lim272 lim272 4096 12월  5 12:59 sample
drwxrwxr-x. 2 lim272 lim272  136 12월 11 15:58 conf
drwxrwxr-x. 2 lim272 lim272  177 12월 19 10:47 include
drwxrwxr-x. 2 lim272 lim272   27 12월 19 10:47 lib
drwxrwxr-x. 2 lim272 lim272 4096 12월 19 10:47 bin
drwxrwxr-x. 2 lim272 lim272 4096 12월 19 11:19 trc
drwxrwxr-x. 2 lim272 lim272    6 12월 19 11:37 arch
drwxrwxr-x. 2 lim272 lim272    6 12월 19 11:43 wal
drwxrwxr-x. 2 lim272 lim272  169 12월 19 11:48 dbf
drwxrwxr-x. 2 lim272 lim272   66 12월 19 11:49 repl
success
```

원격 연결

dbmMetaManager에서 원격으로 연결해야 할 경우, 다음과 같이 수행한다.
원격지에는 반드시 dbmListener가 구동된 상태이어야 한다.

```
Connect := CONNECT <remote ip> <remote listen port number> <remote instance name>
```

다음은 Listener에 접속하여 원격 노드에 명령을 실행하는 예이다.

```
dbmMetaManager(unknown)> connect 127.0.0.1 27584 dict
success
dbmMetaManager(127.0.0.1:DICT)> create instance demo
success
```

struct out (구조체 출력)

현재 생성된 테이블을 C type 구조체 형태로 출력한다.

```
dbmMetaManager(DEMO)> create table t1
(c1 int, c2 short, c3 long, c4 float, c5 double, c6 date, c7 char(33) );
success
dbmMetaManager(DEMO)> struct out t1;
```

```
typedef struct T1
{
    int C1;
    short C2;
    long long C3;
    float C4;
    double C5;
    struct timeval C6;
    char C7[33];
} T1
success
```

history

현재까지 수행된 명령 중에 최근 20개를 출력한다.

```
dbmMetaManager(DEMO)> history;
   0: select 1 from dual
   1: select sysdate from dual
success
```

"/" (재수행 명령)

직전에 수행한 명령 (history의 마지막 명령) 또는 history에 저장된 번호를 지정하여 해당 명령을 재수행한다.

```
dbmMetaManager(DEMO)> history;
   0: select 1 from dual
   1: select sysdate from dual
success
dbmMetaManager(DEMO)> /
--------------------------------------------------------------------------------
SYSDATE : 2025/01/08 08:19:35.806824
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> /0
--------------------------------------------------------------------------------
1 : 1
--------------------------------------------------------------------------------
1 row selected
```

ed

직전에 수행한 명령 (history의 마지막 명령) 또는 history에 저장된 번호를 지정하여 해당 명령을 편집한다.

```
dbmMetaManager(DEMO)> history;
   0: select 1 from dual
   1: select sysdate from dual
success
dbmMetaManager(DEMO)> ed
success
####################################
## vi 모드에서 아래와 같이 편집한 경우
##  select sysdate from dual ==> select sysdate, sysdate from dual
####################################
dbmMetaManager(DEMO)> /
--------------------------------------------------------------------------------
SYSDATE : 2025/01/08 08:20:52.439592
SYSDATE : 2025/01/08 08:20:52.439592
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> history
   0: select 1 from dual
   1: select sysdate from dual
   2: select sysdate, sysdate from dual
success
```

> 노트
> 명령 편집기는 환경 변수 DBM_EDITOR 를 통해 지정할 수 있으며, 기본값은 vi 이다.

set instance

지정한 instance로 접속을 전환한다.

```
dbmMetaManager(DICT)> set instance demo;
success
dbmMetaManager(DEMO)>
```

> 노트
> 전환되기 전의 instance에서 수행된 트랜잭션은 자동으로 rollback 처리된다.

set password

현재 instance에 대한 접근 암호를 설정하거나 해제한다.

```
dbmMetaManager(DEMO)> set password null lite_pwd;    # 암호 설정
success
dbmMetaManager(DEMO)>  set password lite_pwd null;   # 설정된 암호 해제
success
```

- set password의 첫 번째 인자는 현재 암호를, 두 번째 인자는 변경할 암호를 의미한다.
- null은 암호가 설정되지 않았거나, 암호를 해제할 때 사용한다.

암호가 설정된 이후에는 패스워드가 일치하지 않으면 아래와 같이 접속 또는 수행이 불가능하다.

```
$ dbmMetaManager
************************************************************
* Copyright 2010. SUNJESOFT Inc. All rights reserved.
* Version (Debug 3.2-3.2.6 revision(6754))
************************************************************
dbmMetaManager(DEMO)> set password null lite_pwd;
success
dbmMetaManager(DEMO)> quit
$ dbmMetaManager
************************************************************
* Copyright 2010. SUNJESOFT Inc. All rights reserved.
* Version (Debug 3.2-3.2.6 revision(6754))
************************************************************
ERR] invalid passwd, use '-e' option as instance need passwd
$ dbmMetaManager -e abcd
************************************************************
* Copyright 2010. SUNJESOFT Inc. All rights reserved.
* Version (Debug 3.2-3.2.6 revision(6754))
************************************************************
ERR] invalid passwd, use '-e' option as instance need passwd
```

노트

- 이 규칙은 모든 API 와 제공되는 utility에도 동일하게 적용되므로, password 설정 시 주의해야 한다.
- 인증 관련 API는 **dbmAuthorize**를 참조한다.

startup

모든 instance 또는 지정한 특정 instance를 복구한다. 복구를 수행하려면 디스크 모드로 운영 중이어야 하며, dbm Ckpt에 의해 생성된 데이터 파일이 필요하다.

```
dbmMetaManager(DICT)> startup DEMO
success
```

Startup 명령의 내부 처리 과정은 다음과 같다.

- Instance 를 지정하지 않으면 dictionary instance의 DIC_INST.dbf에 저장된 모든 instance를 복구한다.
- 대상 instance segment와 dictionary built-in table을 생성한다.
- 기존에 생성되어 있는 DIC_TABLE.dbf 를 참조하여 복구할 테이블 목록을 구성한다.
- DIC_TABLE.dbf, DIC_COLUMN.dbf, DIC_INDEX.dbf, DIC_INDEX_COLUMN.dbf를 참조하여 table 과 index를 생성한다.
- 각 테이블 별 데이터 파일을 참조하여 데이터를 로딩한다.

노트

Startup 과정에서는 instance가 복구 명령 이전에 생성한 dictionary 데이터 파일에 저장된 내용을 기준으로 복구 대상 테이블 목록을 구성한다. 따라서 create instance 시점에 생성된 dictionary 데이터 파일이 반드시 존재해야 한다.

1.6 복구 가이드

본 장에서는 GOLDILOCKS LITE 환경에서 장애 또는 데이터 손상으로 인해 데이터 복구가 필요한 상황에 대비하여, 사용자가 활용할 수 있는 복구 방안을 설명한다.

스냅샷 (SnapShot) 저장 및 복원

dbmExp를 사용하면 instance에 포함된 전체 object와 data를 text 형태로 내보낼 수 있으며, dbmImp를 통해 해당 데이터를 다시 복원할 수 있다. 복원 시점은 dbmExp를 수행한 시점으로 고정된다.

다음은 전체 instance에 포함된 모든 object와 data를 내려받는 예이다.

```
[lim272@tech10 tmp]$ dbmExp -a -d
[ DEMO ] Instance start...
   + (T1) (10 rows) download
   + (T2) (10 rows) download
   + (T3) (10 rows) download
   + (T4) (10 rows) download
   + (T5) (12 rows) download
```

명령을 실행한 경로에는 다음 예시와 같이 여러 개의 파일이 생성된다.

```
-rw-rw-r--. 1 lim272 lim272    20 Jun 24 14:38 DEMO_T1.fmt
-rw-rw-r--. 1 lim272 lim272 10221 Jun 24 14:38 DEMO_T1.dat
-rw-rw-r--. 1 lim272 lim272    20 Jun 24 14:38 DEMO_T2.fmt
-rw-rw-r--. 1 lim272 lim272 10221 Jun 24 14:38 DEMO_T2.dat
-rw-rw-r--. 1 lim272 lim272    20 Jun 24 14:38 DEMO_T3.fmt
-rw-rw-r--. 1 lim272 lim272 10221 Jun 24 14:38 DEMO_T3.dat
-rw-rw-r--. 1 lim272 lim272    20 Jun 24 14:38 DEMO_T4.fmt
-rw-rw-r--. 1 lim272 lim272 10221 Jun 24 14:38 DEMO_T4.dat
-rw-rw-r--. 1 lim272 lim272    20 Jun 24 14:38 DEMO_T5.fmt
-rw-rw-r--. 1 lim272 lim272  1485 Jun 24 14:38 DEMO_create.sql
-rw-rw-r--. 1 lim272 lim272 12265 Jun 24 14:38 DEMO_T5.dat
-rw-rw-r--. 1 lim272 lim272   180 Jun 24 14:38 DEMO_in.sh
-rw-rw-r--. 1 lim272 lim272    20 Jun 24 14:38 DEMO_create_proc.sql
```

각 파일은 데이터 복원 과정에서 서로 다른 역할을 수행하며, 파일 형식별 용도는 다음과 같다.

형식	설명
<INSTANCE_NAME>_<OBJECT_NAME>.dat	테이블 별 데이터가 저장된다.
<INSTANCE_NAME>_<OBJECT_NAME>.fmt	데이터 로딩을 위한 테이블 별 column 구성 정보가 저장된다.
<INSTANCE_NAME>_create.sql	Object 생성 구문이 저장된다.
<INSTANCE_NAME>_create_proc.sql	Procedure 생성 구문이 저장된다. (deprecated)
<INSTANCE_NAME>_in.sh	Instance 내 모든 테이블을 대상으로 dbmImp를 일괄 실행하기 위한 스크립트 파일이 저장된다.

노트

한편, LITE 환경 간 (LITE ↔ LITE) dbmExp/dbmImp 작업에서 binary 옵션을 활성화하면 데이터 변환 과정을 생략하고 즉시 적재할 수 있다. 이를 통해 대량 데이터 업로드 시 성능을 향상시킬 수 있다.
(단, binary 옵션은 GOLDILOCKS LITE 전용 기능으로, 다른 DBMS와는 호환되지 않는다.)

디스크 로깅을 이용한 복원

GOLDILOCKS LITE의 disk logging은 트랜잭션이 commit 되는 시점을 기준으로 redo log를 로그 파일에 기록하는 매커니즘을 의미한다.

GOLDILOCKS LITE에서는 다음 절차를 통해 디스크 로깅 기반 데이터 복구를 수행한다.
- dbmCkpt 프로세스를 실행하여 로그 파일에 기록된 트랜잭션 이력을 반영한 데이터 파일을 생성한다.
- dbmMetaManager에서 startup 명령을 수행한다.

GOLDILOCKS LITE는 다음과 같은 디스크 로깅 방식을 제공한다.

DBM_LOG_CACHE_MODE	설명
NONE (0)	각 세션이 자신의 로그 파일에 트랜잭션 로그를 직접 기록한다.
NVDIMM (1)	NVDIMM을 log cache로 사용한다. 디스크에 영구 저장하려면 dbmLogFluhser 프로세스를 반드시 구동해야 한다.
SHM (2)	Shared memory를 log cache로 사용한다. 디스크에 영구 저장하려면 dbmLogFluhser를 반드시 구동해야 한다.

- Non-log cache 방식에서는 각 세션이 자신의 로그 파일에 직접 트랜잭션 로그를 기록한다. 이 방식은 구조가 단순하지만, 디스크 I/O 부하로 인해 성능 저하가 발생할 수 있다.
- 반면, log cache 방식에서는 메모리에 일정 공간을 할당하고 여러 세션이 해당 공간에 트랜잭션 로그를 기록한다. 따라서 log cache 방식을 사용하는 경우, 메모리에 기록된 트랜잭션 로그를 로그 파일에 반영하기 위해서는 dbmLogFlusher 프로세스가 반드시 구동되어야 한다.

주의

만약 log cache 모드에서 dbmLogFlusher가 동작하지 않으면, log cache에 할당된 메모리가 비워지지 않아 새로운 트랜잭션이 모두 대기 상태에 진입하게 된다.

디스크 로깅에 의해 생성되는 파일은 DBM_DISK_LOG_DIR 경로에 저장되며, 다음과 같은 유형의 파일이 생성된다.

File	설명
<Instance_name>.anchor	create instance 시점에 생성되며, 세션 및 dbmLogFlusher, dbmCkpt에 의해 참조/갱신되는 로그 파일의 상태 정보를 기록한다. 예: DEMO.anchor
<Instance_name>.<session_id>.<logfile sequence>	세션 또는 dbmLogFlusher에 의해 생성되는 로그 파일이다. 예: DEMO.1.0

로그 파일은 buffered i/o 또는 direct i/o 방식으로 저장되며, DBM_DIRECT_IO_ENABLE 속성을 통해 제어할 수 있다. 또한 commit 시점에 디스크에 저장되는 것을 보장하려면 DBM_COMMIT_WAIT_MODE 속성을 설정해야 한다. (자세한 내용은 **환경 변수 및 프로퍼티**를 참조한다.)

노트

성능과 데이터 안정성은 trade-off 관계에 있으므로, DBM_COMMIT_WAIT_MODE 속성은 데이터 무결성이 특히 중요한 환경에서만 설정하는 것이 권장된다.

체크포인트

체크포인트란 디스크 로그 파일에 기록된 트랜잭션 이력을 반영하여 데이터 파일을 생성하거나 갱신하는 과정을 의미한다.
체크포인트가 수행되면 반영이 완료된 로그 파일은 DBM_ARCHIVE_LOG_ENABLE 설정값에 따라 삭제되거나, 지정된 archive 경로로 이동된다.
체크포인트 과정에서 생성되는 데이터 파일의 저장 경로는 DBM_DISK_DATA_FILE_DIR 속성을 통해 지정하며, 생성되는 데이터 파일의 이름은 다음 규칙을 따른다.

```
<Instance Name>_<Object Name>.dbf
```

노트

체크포인트 수행 중에는 I/O가 발생하므로 system 성능에 영향을 줄 수 있다. 또한 체크포인트 수행 간격 동안 로그 파일이 누적되기 때문에 디스크 사용량이 증가할 수 있다. 따라서 안정적인 운영을 위해서는 데이터

> 파일과 로그 파일을 저장하기 위한 충분한 디스크 공간을 사전에 확보하고, 시스템 부하를 고려하여 체크포인트 주기를 적절히 설정해야 한다.

복구

디스크 로깅과 체크포인트를 이용한 복구는 다음 순서로 수행한다.

```
(1) shell> dbmCkpt -i demo -f
(2) dbmMetaManager(unknown)> startup;
success
```

dbmCkpt는 기본적으로 기록이 완료된 로그 파일만 데이터 파일에 반영한다.
따라서 위의 예시 (1)과 같이 -f option을 지정하여, 현재 기록 중인 로그 파일까지 포함해 데이터 파일에 반영하도록 한다.
이후 예시 (2)와 같이 dbmMetaManager에서 startup 명령을 수행하면 복구가 완료된다.
Startup 과정에서는 dictionary table 정보를 기반으로 object를 생성하고, 체크포인트를 통해 생성된 데이터 파일을 참조하여 데이터를 복원한다.

Startup 명령은 dictionary instance의 DIC_INST에 기록된 instance를 대상으로 수행된다.
특정 instance만 복구하려면 **startup** 명령 실행 시 대상 instance를 지정하여 수행해야 한다.

> 노트
> 한편, 운영 중 복구 과정에서 현재 로그 파일을 반영하지 않은 상태로 startup을 실행한 경우, 기존 shared memory를 제거한 후 체크포인트를 다시 수행하고 startup 명령을 재실행해야 한다.

Archive 로그를 기반으로 체크포인트를 수행해야 하는 경우에는, 다음과 같이 반영할 로그 파일의 시작 번호를 지정하여 체크포인트를 수행할 수 있다.

```
dbmMetaManager> alter system reset checkpoint demo -1;
```

1.7 Utility

본 절에서는 GOLDILOCKS LITE에서 제공하는 사용자 utility에 대해 설명한다.

dbmExp

dbmExp는 GOLDILOCKS LITE 내에 존재하는 object 생성과 관련된 SQL script와 데이터를 추출하는 프로세스이다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-a	모든 instance를 export 할 경우에 지정하며 -i 옵션과 함께 지정할 수 없다.
-i <instance Name>	특정 instance를 지정하여 export 할 경우에 입력한다.
-t <table Name>	특정 table을 지정하여 export 할 경우에 입력한다. 입력하지 않으면 instance 내 전체 table을 대상으로 추출한다.
-r <delimeter>	데이터 추출 옵션이 활성화 된 경우, 레코드 단위 구분자를 지정한다.
-c <delimeter>	데이터 추출 옵션이 활성화 된 경우, column 단위 구분자를 지정한다.
-d	테이블 데이터를 함께 추출하고자 할 경우에 입력한다.
-b	데이터를 binary 형태로 추출할 경우에 입력한다. 추후 dbmImp로 로딩할 때 parsing 비용을 줄일 수 있다. (단, 타 DBMS와는 호환되지 않는다.)
-n	Column 타입이 char 형식일 경우, 데이터 내에서 null 이전까지의 값만 출력하고자 할 때 사용된다.
-p	Instance에 password 가 설정된 경우에 입력한다.

노트

- dbmExp에 의해 추출된 데이터는 기본적으로 text로 저장되며, 다른 DBMS로 데이터를 이관하여 사용할 수도 있다.
- 별도의 옵션을 지정하지 않은 경우 column delimeter와 row delimeter가 자동으로 설정되어 csv 형식으로 출력된다.
- 사용자가 column delimeter와 row delimeter를 직접 지정할 경우, 두 delimiter는 서로 다른 값이어야 하며 데이터 자체에 포함되지 않는 문자를 사용해야 한다.
- Date column은 기본적으로 microseconds까지 출력된다. 다른 DBMS로 이관하여 적재할 경우에는, 대상 DBMS에서 지원하는 적절한 데이터 타입과 포맷에 맞게 변환하여 사용해야 한다.

사용 예 (특정 instance의 하위 object 및 데이터 추출)

다음은 특정 instance(demo)에 포함된 object와 데이터를 추출하는 예이다.

```
$ dbmExp -i demo -d
[ DEMO ] Instance start...
   + (QUE1) (1 rows) download
   + (T1) (1 rows) download
```

- 추출된 각 파일의 이름은 <InstanceName>_<ObjectName> 형식으로 출력된다.
- Sequence object의 경우, current 값을 start with 값으로 설정하여 생성 스크립트가 출력된다.

dbmExp 추출 결과물

dbmExp 명령을 실행하면 instance 와 테이블 정보에 따라 여러 개의 결과 파일이 생성된다.
각 결과 파일은 고유한 이름 형식과 역할을 가지며, 용도는 아래와 같다.

File name	설명
<Instance name>_create.sql	해당 instance에 포함된 모든 object의 생성 구문을 하나의 스크립트로 저장한 파일이다.
DEMO_create_proc.sql	deprecated
<Instance name>_in.sh	Instance에 포함된 모든 테이블 데이터를 로딩하기 위한 명령어를 저장한 파일이다.
<Instance name>_<Table name>.dat	지정된 테이블의 실제 데이터가 저장되는 파일이다.
<Instance name>_<Table name>.fmt	테이블의 column 순서와 사용 여부를 나열한 파일이다.

dbmImp

dbmImp는 csv 형식 또는 사용자 정의 구분자를 사용하는 데이터 파일을 분석하여 지정된 테이블에 데이터를 적재하는 프로세스이다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance Name>	데이터를 적재할 대상 instance name을 지정한다.
-t <table Name>	데이터를 적재할 대상 table name을 지정한다.
-r <delimeter>	레코드 단위의 구분자를 지정한다.
-c <delimeter>	Column 단위의 구분자를 지정한다.

입력 옵션	설명
-d <data file name>	적재할 데이터 파일의 이름을 지정한다.
-f <form file name>	데이터 포맷 정보를 정의한 form file의 이름을 지정한다.
-p	Instance에 password가 설정되어 있는 경우에 입력한다.
-b	데이터 파일이 dbmExp에 의해 생성된 binary format일 경우에 지정한다. (이 옵션을 사용하면 parsing 비용을 줄일 수 있다.)

> 노트
>
> Form file의 내용을 수정하면 특정 column을 제외하거나, column의 순서를 변경하여 데이터를 적재할 수 있다.

레코드 또는 column 구분자로 특수 문자를 사용할 경우, 아래와 같은 입력 방식을 사용해야 한다.

구분자	Ascii 값	입력 방식 (on dbmImp)
\t (tab)	9	\\t
\r (carriage return)	13	\\r
\n (line feed)	10	\\n

사용 예

다음은 dbmImp를 이용하여 데이터 파일을 테이블에 적재하는 예이다.

```
$ dbmImp -i demo -t t1 -d DEMO_T1.dat
(DEMO.T1) Import Success. (ReadCount=1, Inserted=1)
$ dbmImp -i demo -t que1 -d DEMO_QUE1.dat
(DEMO.QUE1) Import Success. (ReadCount=1, Inserted=1)
```

추출된 form 파일을 수정하여, 데이터 적재 시 특정 column을 제외하려면 해당 column의 Y 값을 N으로 변경한다.

```
$ cat DEMO_T1.fmt
C1 Y
C2 Y
C3 Y
C4 Y
C5 Y
C6 Y    ❶ N 으로 변경
C7 Y
C8 Y
```

```
C9 Y
C10 Y
```

- 데이터 파일에 C2, C3 column이 없다면, 위 예제에서는 해당 column에 해당하는 행(C2, C3)을 제거한 후 dbmImp를 실행한다.
- 데이터 파일에서 C2, C3 column의 순서가 서로 바뀌어 있다면, form 파일에서도 C2, C3 line의 순서를 동일하게 변경하여 처리한다.

노트

DATE 타입은 기본적으로 YYYY/MM/DD HH:MI:SS.SSSSSS 형식의 문자열을 기준으로 데이터를 해석한다. 따라서 다른 DBMS에서 추출한 데이터를 적재하는 경우, 해당 DBMS에서도 DATE 값을 동일한 형식의 문자열로 변환하여 추출해야 한다.

dbmCkpt

dbmCkpt는 디스크 모드 운영 중에 생성된 로그 파일을 읽어 데이터 파일로 반영하는 프로세스이다.
로그 파일은 세션 또는 dbmLogFlusher에 의해 다음과 같은 형식의 파일명으로 기록된다.

```
<Instance Name>.<Session ID>.<LogFile Sequence>
* dbmLogFlusher는 Session ID를 0으로 설정하여 로그 파일을 생성한다.
```

각 세션 및 dbmLogFlusher는 로그를 기록하여 DBM_DISK_LOG_FILE_SIZE 크기에 도달하면 다음 sequence 번호로 새로운 로그 파일을 생성한다.
이러한 과정을 log switching 이라 하며, sequence 변경 정보는 모두 log anchor에 기록된다.
dbmCkpt 는 log anchor에 저장된 현재 읽고 있는 로그 파일의 sequence와 각 세션의 log switching 이력을 기반으로, 기록이 완료된 로그 파일만 읽어 데이터 파일에 반영한다.
따라서 복구를 위해 startup 명령을 실행하기 전에, 현재 기록 중인 로그 파일까지 반영하려면 반드시 -f 옵션을 사용하여 체크포인트를 수행해야 한다.

노트

각 로그 파일의 크기가 정확히 DBM_DISK_LOG_FILE_SIZE 와 일치하는 것은 아니다. 마지막에 기록되는 트랜잭션 로그의 크기가 큰 경우, log switching 이후 새 로그 파일에 기록될 수 있다. dbmLogFlusher가 생성하는 로그 파일은 경우에 따라 최대 크기를 초과하여 기록될 수 있다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance name>	대상 instance name을 지정한다.
-v	체크포인트를 한 번만 수행한 후 종료한다.
-f	로그 파일이 아직 모두 기록되지 않은 상태라도 강제로 체크포인트를 수행하도록 지정한다. (Startup 직전에는 반드시 해당 옵션을 포함하여 체크포인트를 수행해야 한다.)
-s	체크포인트를 수행하는 간격을 지정한다. (단위: 초)

dbmDump

dbmDump는 shared memory의 내용과 디스크에 저장된 파일의 내용을 출력하는 tool 이다.
입력 옵션과 지정한 파일 유형에 따라 해당되는 정보를 출력한다.

> 노트
> dbmDump의 출력 형식은 버전에 따라 변경될 수 있다.

Dump Memory

dbmDump를 사용하면 현재 메모리에 존재하는 instance, table, index 등의 주요 정보를 확인할 수 있다.

연관 object	입력 가능 옵션	설명
ALL	-h	도움말을 출력한다.
instance, table, index	-i <instance name>	대상 instance name을 지정한다.
instance	-x <session ID>	Instance dump 수행 시, 입력한 session ID와 연관된 정보만 출력한다.
table	-t <table name>	대상 table name을 지정한다.
table	-s	Slot-area 정보를 출력한다.
index	-d <index name>	대상 index name을 지정한다.

dbmDump (Instance)

dbmDump (Instance)는 instance의 헤더 정보와 세션별 주요 트랜잭션 정보를 dump 한다.

사용 예

```
[majaehwa@tech9 new_lite]$ dbmDump -i demo;
InstName=DEMO, TransId=-1
Segment Init=128, extend=128, max=1048576
SegmentNo=0, Alloc=12, Free=139, Gap=128
Lock=-1, SCN=29, ObjectId=12, Name=DEMO, Active=1,DiskLogMode=1, LogFileSize=104857600,
CreateTime=2025-07-24 15:17:12
================================================================================================
==============
TxInfo(T=1, P:3560962, T=3560962), Stat=transaction, First/Last(11:11), SavePoint(-1:-1)
WaitTrans=-1(,-1), SCN=9223372036854775807, Repl=0, BeginTime=2025/07/28 15:50:27.098864
Page=11, PrevOffset=-1, Offset=32, NextLogPage=11, NextLogOffset=32, Size=0, LogType=INSERT_
TABLE, RelSlot=0, RelExtra=1, ObjectId=(12)
Page=11, PrevOffset=32, Offset=256, NextLogPage=11, NextLogOffset=256, Size=80, LogType=UPDATE
_TABLE, RelSlot=0, RelExtra=1, ObjectId=(12)
================================================================================================
==============
```

Instance header에서 제공되는 주요 정보는 다음과 같다.

항목	설명
Lock	Instance lock 설정 여부
SCN	System commit number 용도
ObjectId	Reserved
Name	Instance 이름
Active	Reserved
DiskLogMode	디스크 모드 활성화 여부
LogFileSize	디스크 모드 사용 시 로그 파일 크기
CreateTime	Instance 생성 시각

각 session에 대해 출력되는 기본 정보는 다음과 같다.

항목	설명
TxInfo(T: P: T)	Session 정보 • T: 내부에서 할당된 session ID • P: Process ID • T: Thread ID
Stat	트랜잭션 상태 • transaction: 트랜잭션 진행 가능 또는 진행 중 • commit start: 메모리 커밋 시작 • commit completed: 메모리 커밋 완료 • rollback completed: 메모리 롤백 완료 • recovery completed: 메모리 비정상 복구 완료

항목	설명
	※ 실제 운영에서는 Stat=transaction 외의 상태는 거의 출력되지 않는다.
First/ Last	Session이 사용 중인 undo 공간의 첫 번째와 마지막 PageId
SavePoint	Reserved
WaitTrans	트랜잭션이 LockWait 상태일 경우 대기 중인 상대 transactionId (대상 테이블, 대상 slot Id 포함)
SCN	Commit 시점에 할당된 SCN
Repl	이중화 모드 설정 여부
BeginTime	세션이 할당된 시간

현재 session에 진행 중인 트랜잭션이 있을 경우. 출력되는 항목은 다음 표와 같다.

항목	설명
Page	현재 트랜잭션 로그가 기록된 page ID 이다.
PrevOffset	이전 트랜잭션 로그가 기록된 page 내 offset 정보이다.
Offset	현재 트랜잭션 로그가 기록된 page 내 offset 정보이다.
NextLogPage	다음 트랜잭션 로그가 기록된 page ID 이다.
NextLogOffset	다음 트랜잭션 로그가 기록될 page 내 offset 정보이다.
Size	Rollback image의 크기이다.
LogType	트랜잭션 로그 유형이다.
RelSlot	Table 내 slot ID 이다.
RelExtra	Table 내 고유 record ID 이다.
ObjectId	Object에 부여되는 고유 ID 이다.

dbmDump (Table)

dbmDump (Table)은 테이블의 header 상태 정보와 레코드가 저장된 slot 영역의 row header 및 데이터를 dump 한다.

사용 예

```
[majaehwa@tech9 wal]$ dbmDump -i demo -t t1;
InstName=DEMO, ObjectId=11, TableName=T1, Lock=-1, RowSize=4, CreateTime=2025-07-24 15:17:24,
CreateSCN=22, ExtraKey=2, Root=-1
1] C1 int(Type=2) 4 0
--------------------------------------------------------------------------------
SlotID=0, sExtraKey=1, Lock=13313, Size=4, SCN=24
C1 = [4]
SlotID=1, sExtraKey=2, Lock=14337, Size=4, SCN=25
C1 = [2]
```

출력 상단에는 table header 정보가 표시되며, 각 항목의 의미는 다음과 같다.

항목	설명
InstName	테이블이 속한 instance 이다.
ObjectId	테이블의 object ID 이다.
Lock	테이블 lock 정보이다.
RowSize	테이블에 저장되는 레코드 크기이다.
CreateTime	테이블 생성 시각이다.
CreateSCN	테이블 생성 시점의 SCN 이다.
ExtraKey	레코드 저장 시마다 채번되는 고유 번호이다.
Root	Splay table인 경우, 트리의 root 노드를 나타내는 root slot ID 이다.

레코드 별로 출력되는 항목은 다음과 같다.

항목	설명
SlotID	테이블 내 레코드의 위치 정보이다. 레코드는 고정 크기의 배열 형태로 저장되며, segment 내 N번째 저장 위치를 의미한다.
sExtraKey	레코드의 고유 ID 이다.
Lock	현재 또는 직전에 lock을 점유한 TransID 이다.
Size	레코드 크기이다.
SCN	레코드의 커밋 번호이다.

노트

Splay table의 경우, row header에 tree 구조 정보가 저장된다.
또한 table header에 출력된 "Root" 항목은 splay tree 구조에서 최상위 노드 (시작점)를 의미한다.

```
[majaehwa@tech9 src]$ dbmDump -i demo -t t4;
InstName=DEMO, ObjectId=14, TableName=T4, Lock=-1, RowSize=4, CreateTime=2025-07-29 14:40:34,
CreateSCN=41, ExtraKey=1, Root=0
1] C1 int(Type=2) 4 0
--------------------------------------------------------------------------------
SlotID=0, ExtraKey=1, Lock=31745, Size=4, SCN=45, parent=-1, left=-1, right=-1
C1 = [5]
```

dbmDump (Index)

dbmDump (Index)는 B-tree index의 헤더 정보와 index가 저장된 구조를 dump한다.

사용 예

```
[majaehwa@tech9 dbf]$ dbmDump -i demo -t t1 -d idx_t1;
InstName=DEMO, TableName=T1, IndexName=IDX_T1
try to attach a index segment
Lock=-1, RootNodeId=0, Unique=0, CompareCount=1, Depth=0, SplitCount=0, RefCount=0, CreateTime
=2025-07-24 15:21:12
==== NODE (0) : Valid=1 Cnt:2 (Prev:-1, Next:-1)
SlotNo=0] DataSlotId=1 ExtraKey=2 (Left=-1, Right=-1)
Key(C1) Val=[2]
SlotNo=1] DataSlotId=0 ExtraKey=1 (Left=-1, Right=-1)
Key(C1) Val=[4]
```

Index header의 주요 항목은 다음과 같다.

항목	설명
Lock	현재 index header에 lock을 점유하고 있는 TransId 이다. 이 값이 계속해서 -1이 아닌 동일한 값으로 유지된다면 index를 재구축해야 한다.
RootNodeId	Root node의 ID 이다.
Unique	0: non-unique 1: unique
CompareCount	Index가 사용될 때마다 증가하는 값이다.
Depth	Btree의 깊이를 표현한다.
SplitCount	Leaf node가 split 될 때마다 증가하는 값이다.
RefCount	현재 index를 탐색 중인 트랜잭션의 개수이다.
CreateTime	Index가 생성된 시간이다.

노트

DataSlotId에 해당하는 데이터를 확인하려면 **dbmDump (Table)**을 참조한다.

Dump File

Dump file 기능은 anchor, data, log 파일을 dump 하여 파일에 기록된 주요 정보를 확인하는 데 사용한다.

입력 옵션	설명
-h	도움말을 출력한다.
-f <anchor filename, data filename, log filename>	Dump 할 대상 filename을 지정한다

dbmDump (Anchor)

dbmDump (Anchor)는 log anchor 파일의 내용을 dump 한다.

사용 예

```
[majaehwa@tech9 wal]$ dbmDump -f DEMO.anchor
File Info : Anchor, version 1
MemAnchor(Trans=1): mLogFileNo=0, LogFileOffset=9728, CkptFileNo=-1, ArchiveFileNo=-1,
CaptureFileNo=-1
MemCacheInd: CacheWriteInd=0, CacheReadInd=0, LogFileNo=0, LogFileOffset=0
LogAnchor(Trans=1): mLogFileNo=0, LogFileOffset=9728, CkptFileNo=-1, ArchiveFileNo=-1,
CaptureFileNo=-1
LogCacheInd: CacheWriteInd=0, CacheReadInd=0, LogFileNo=0, LogFileOffset=0
```

Log anchor 파일에는 세션별 로깅 동작 방식과 log cache 방식에 관한 정보가 기록되어 있으며, 출력 결과의 해석은 시스템에서 설정한 디스크 로깅 모드에 따라 달라진다.

DBM_LOG_CACHE_MODE가 0 인 경우, MemAnchor의 주요 항목은 다음과 같다.

항목	설명
MemAnchor(Trans=N)	세션 ID가 N임을 나타낸다.
mLogFileNo	해당 세션이 현재 사용 중인 로그 파일 번호 이다.
LogFileOffset	세션이 기록 중인 로그 파일 내 offset 이다.
CkptFileNo	체크포인트 수행 대상이 되는 파일의 시작 번호 이다.
ArchiveFileNo	아카이브 작업의 수행 대상이 되는 파일의 시작 번호 이다.
CaptureFileNo	Reserved

DBM_LOG_CACHE_MODE가 (1, 2) 인 경우, MemCacheInd의 주요 항목은 다음과 같다.

항목	설명
CacheWriteInd	Log cache 내에서 새로운 로그를 기록할 수 있는 위치이다.
CacheReadInd	dbmLogFlusher 가 flush 작업 수행 시 읽어야 하는 위치이다.
LogFileNo	dbmLogFlusher가 기록 중인 로그 파일 번호 이다.
LogFileOffset	dbmLogFlusher가 현재 기록 중인 로그 파일 내 위치이다.

노트

Log anchor는 mmap 방식으로 메모리에 공유되며, 동시에 동일한 내용이 파일에도 기록된다. 출력 시 "Mem" prefix가 붙은 라인은 메모리에 저장된 내용을 dump 한 결과이며, prefix가 없는 라인은 실제 파일에 기록된 정보를 의미한다.

dbmDump (Datafile)

dbmDump (Datafile)은 datafile을 dump한다.

사용 예

```
[majaehwa@tech9 dbf]$ dbmDump -f DEMO_T1.dbf
File Info : Data, version 1
InstName=DEMO, TableName=T1, SlotSize=76, CreateSCN=22
ColumnName (C1), Order=1, Type=int, Offset=0, Size=4
0) Size=4, SCN=24                          ## 0) 0의 의미는 SlotID이다.
(C1=4)
1) Size=4, SCN=25
(C1=2)
Dump (DEMO_T1.dbf) completed
```

dbmDump (Logfile)

dbmDump (Logfile)은 redo logfile의 주요 정보를 dump한다.

사용 예

```
[majaehwa@tech9 wal]$ dbmDump -f DEMO.1.0 | tail
   + logType(CREATE_TABLE), object(V$TYPE_INFO), SlotId(-1), sPos(48)
sFileOffset=8192, BlockSCN=22, BlockCount=1, logCount=1, Time=2025/07/24 15:17:24.641676
   + logType(CREATE_TABLE), object(T1), SlotId(-1), sPos(48)
sFileOffset=8704, BlockSCN=24, BlockCount=1, logCount=1, Time=2025/07/24 15:17:32.444694
   + logType(INSERT_TABLE), object(T1), SlotId(0), sPos(48) RowHdr(scn=24, size=4) :
sFileOffset=9216, BlockSCN=25, BlockCount=1, logCount=1, Time=2025/07/24 15:17:38.542943
   + logType(INSERT_TABLE), object(T1), SlotId(1), sPos(48) RowHdr(scn=25, size=4) :
sFileOffset=9728, BlockSCN=26, BlockCount=1, logCount=1, Time=2025/07/24 15:21:12.871522
   + logType(CREATE_INDEX), object(IDX_T1), SlotId(-1), sPos(48)
sReadSz=0, LastOffset=10240, errno=0 at reading blockHdr
```

Redo log는 커밋 시점마다 트랜잭션 내의 모든 로그를 하나의 block이라 불리는 공간에 모아 저장한다. Block의 크기는 DBM_DISK_BLOCK_SIZE 속성으로 지정된다.

각 block에는 header가 존재하며, 주요 항목은 다음과 같다.

항목	설명
sFileOffset	Log가 기록된 파일 내 위치이다.
BlockSCN	Commit SCN 이다.
BlockCount	Block의 개수이다.
logCount	Block 내에 포함된 세부 트랜잭션 로그의 개수이다.

항목	설명
Time	트랜잭션 로그를 디스크에 기록하기 직전의 시각이다.

Block 헤더 정보 다음에 위치한 로그들은 트랜잭션 내의 상세 로그를 의미한다.
각 상세 로그는 다음과 같은 항목으로 구성된다.

항목	설명
logType	트랜잭션 로그의 상세 유형이다.
object	트랜잭션이 발생한 대상 object 이다.
SlotId	대상 object 내의 slot ID 이다.
sPos	현재 읽은 block의 header 크기이다.
RowHdr(SCN, Size)	커밋된 레코드의 SCN 및 크기 정보이다.

dbmListener

dbmListener는 원격 노드에 접속할 때 대상 서버에서 구동해야 하는 프로세스이다. 주요 기능은 다음과 같다.

- 원격 노드에서 접속하는 세션 관리
- 원격 노드로 요청되는 트랜잭션 처리

> 노트
> GOLDILOCKS LITE의 원격 접속 방식은 네트워크 통신 비용이 발생하기 때문에 local processing 대비 성능이 낮다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance name>	dbmListener가 사용할 property section을 지정한다.
-v	Verbose mode로 동작한다.

dbmListener를 구동한 후 사용자가 접속하려면 다음 명령을 사용한다.

```
dbmMetaManager> connect <ip> <port> <instanceName> <password> ;
* password는 instance에 암호가 설정된 경우에만 필요하다.
```

사용자 프로그램에서는 **dbmConnect** API를 통해 연결할 수 있다.

dbmLogFlusher

dbmLogFlusher는 디스크 모드에서 log cache 방식을 사용할 때, cache에 기록된 트랜잭션 로그를 디스크로 저장하는 프로세스이다.

Log cache 공간은 DBM__LOG_CACHE_SIZE로 제한된 범위 내에서 운영된다.

Log cache를 사용하는 설정에서는 트랜잭션의 commit이 다음 절차로 완료된다.

1. 모든 세션은 log cache에서 기록할 공간을 할당받는다.
   (이 과정에서 세션 간 경합이 발생할 수 있다.)
2. 세션은 할당된 공간에 로그 기록을 완료한 후, 해당 로그 블록을 valid 상태로 마크한다.
3. Commit 결과를 사용자에게 반환한다.

dbmLogFlusher는 위 과정에서 트랜잭션들이 기록한 공간 정보의 변화를 감지하여, valid 상태의 트랜잭션 로그를 디스크로 기록한다.
• 특정 세션의 로그 기록이 지연되면, dbmLogFlusher 또한 해당 세션의 기록이 완료될 때까지 기다린다.
• DBM_DISK_COMMIT_WAIT 옵션이 설정된 경우, 세션이 commit을 호출하면 dbmLogFlusher가 로그를 디스크에 완전히 기록한 이후에야 응답을 받게 된다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance name>	대상 instance name을 지정한다

dbmMonitor

dbmMonitor는 GOLDILOCKS LITE의 상태를 모니터링 하는 프로세스이다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance name>	대상 instance name을 지정한다
-s <interval time>	모니터링 결과의 출력 간격을 설정한다. (Second 단위)
-u	Usage info에서 사용자가 입력한 임계값(%)을 초과하는 테이블만 표시한다.
-v	Verbose mode로 구동된다.

사용 예

```
[nh39@tech9 new_lite]$ dbmMonitor -i demo
2025/07/28 20:11:29
[Table Usage]
TableName                                MAX           TOTAL            USED
FREE  Usage Info
-------------------------------------------------------------------------------------------------
--------------
DIC_CAPTURE_HOST                      4096000            1024               0
1024   0.00
DIC_CAPTURE_TABLE                     4096000            1024               0
1024   0.00
DIC_COLUMN                            4096000            1024             176
848   0.00
DIC_INDEX                             4096000            1024              15
1009   0.00
DIC_INDEX_COLUMN                      4096000            1024              41
983   0.00
DIC_INST                              4096000            1024               1
1023   0.00
DIC_PROCEDURE (deprecated)            4096000            1024               0
1024   0.00
DIC_PROCEDURE_TEXT (deprecated)       4096000            1024               0
1024   0.00
DIC_REPL_INST                         4096000            1024               0
1024   0.00
DIC_REPL_TABLE                        4096000            1024               0
1024   0.00
DIC_SEQUENCE                          4096000            1024               0
1024   0.00
DIC_TABLE                             4096000            1024              25
999   0.00
DIC_USER_LIBRARY                      4096000            1024               0
1024   0.00
DIC_USER_LIBRARY_FUNCTION             4096000            1024               0
1024   0.00
DIC_USER_LIBRARY_PARAMETER            4096000            1024               0
1024   0.00
-------------------------------------------------------------------------------------------------
---------
```

```
[Lock Status]
 WaitSessionId   WaitTransPID   LockTransId   LockTransPID LockObject   LockSlot
-------------------------------------------------------------------------------
            2         599736         31745         599672 USER_DATA           0
-------------------------------------------------------------------------------
```

Lock status의 각 항목은 다음과 같다.

항목	설명
WaitSessionId	Lock wait 중인 세션 ID 이다.
WaitTransPID	Lock wait 중인 프로세스 ID 이다.
LockTransID	Lock을 점유한 트랜잭션 ID 이다.
LockTransPID	Lock을 점유한 프로세스 ID 이다.
LockObject	Lock이 발생한 대상 테이블이다.
LockSlot	대상 테이블 내의 SlotID 이다.

dbmReplica

dbmReplica는 slave node에서 데이터를 수신하고 반영하기 위해 구동하는 프로세스이다.

Input Option

입력 옵션	설명
-i	dbmReplica를 여러 개 실행할 경우, 프로세스를 구분하기 위한 alias를 지정한다.
-h	도움말을 출력한다.
-v	Verbose mode로 구동한다.

사용 예

```
$ dbmRepica -i demo
```

노트

dbmReplica는 실행 시점에 dbm.cfg의 "COMMON" section 또는 환경 변수에 설정된 속성을 사용한다.

1.8 Sizing

GOLDILOCKS LITE 사용 전, 필요한 memory segment에 대한 sizing 방안을 설명한다.

> 주의
> - 각 설명에서는 segment 생성 옵션의 init size만을 기준으로 하며, extend/max 옵션을 고려한 추가 산정이 필요할 수 있다. (extend 단위 계산 방식은 동일하게 적용된다.)
> - Sizing 방식은 패치 또는 신규 버전 릴리즈 시점에 기능 개선 및 추가에 따라 변경될 수 있다.

Instance Sizing

Instance segment는 세션 정보와 undo logging 공간으로 사용된다.
- 세션은 헤더 공간에 고정 크기로 포함된다.
- 각 undo logging 공간의 크기는 1M 이다.
- 세션 생성 시 (dbmInitHandle 시점), 세션당 한 개의 undo logging 공간이 할당된다.

```
Instance Sizing =  sizeof(segment Header area) +          // 240 byte
                   sizeof(session area) +                 // 1392640 byte
                   sizeof(long long) * (init_size + 1) +
                   (1M * init_size)
                  ;
```

Table Sizing

Table 크기를 계산할 때는 table header, row header 등을 반드시 고려해야 한다.

```
Table Sizing = sizeof(segment header)  +          // 240 byte
               sizeof(table header)    +          // 136 byte
               sizeof(long long) * (init_size + 1) +
               ( (Record Header=72byte) + RecordSize ) * init_size
               ;
```

노트

디스크 모드 사용 시, 생성되는 데이터 파일의 크기는 table segment의 크기와 동일하다.

Index Sizing

테이블 유형별 index sizing은 다음과 같다.

Table 유형	설명
Btree	아래 산정식을 참조한다.
Splay	별도로 공간을 산정하지 않는다.
Store	Btree index와 동일하다.
Queue	Btree index와 동일하다.
Direct	별도로 공간을 산정하지 않는다.

Btree index의 크기는 다음과 같이 산정한다.

```
init size = Table Segment Init Size * 4 + 4
Index Sizing = sizeof(segment header)  +          // 240 byte
               sizeof(index header) +             // 1192 byte
               ( sizeof(indexNodeHeader) +        // 32 byte
                 sizeof(index slot Header) +      // 32 byte
                 index key column size의 합 ) * 128 * init size
```

노트

Splay, direct table 유형은 dummy index segment 만 생성되며, 크기는 6K이다.

디스크 공간 산정

디스크 모드로 운영할 경우, 다음의 파일들이 생성된다.

- 트랜잭션 로그 파일
- 데이터 파일

트랜잭션 로그 파일

트랜잭션 로그 파일은 체크포인트 과정에서 사용이 완료되면 삭제되거나, 필요 시 archive 경로로 이동된다.
따라서, 체크포인트 수행 간격 동안 디스크 공간이 부족하지 않도록 충분히 산정해야 한다.
만약 디스크 공간이 부족하여 트랜잭션 로그가 commit 시점에 기록되지 못할 경우, 해당 트랜잭션은 실패로 처리되며 모든 작업이 롤백된다.

각 트랜잭션 로그 파일의 최대 크기는 DBM_DISK_LOG_FILE_SIZE 속성으로 정의된다.
트랜잭션 처리량, 체크포인트 수행 간격, 디스크 장비의 I/O 처리 성능을 고려하여, 운영 환경에 맞는 적절한 로그 파일 크기를 산정해야 한다.

데이터 파일

데이터 파일은 table segment가 모두 extend 된 경우를 가정하여, 저장 가능한 최대 공간으로 산정한다.

1.9 Monitoring

본 장에서는 GOLDILOCKS LITE을 모니터링하는 방법에 대해 설명한다.

LOCK 정보 확인

모든 변경 연산은 record 단위로 lock을 점유하여, 다른 세션이 해당 데이터를 변경하지 못하도록 보호한다.
세션이 장시간 대기하는 경우, 원인을 파악하기 위해 다음과 같은 방법을 사용할 수 있다.

```
dbmMetaManager(DEMO)> select * from v$session;
--------------------------------------------------------------------------------
ID                    : 1
TRANS_ID              : 26625
PID                   : 3388915
TID                   : 3388915
OLD_TID               : 3388915
VIEWSCN               : 9223372036854775807
CURR_UNDO_PAGE        : 4
FIRST_UNDO_PAGE       : 4
LAST_UNDO_PAGE        : 4
SAVEPOINT_UNDO_PAGE   : -1
SAVEPOINT_UNDO_OFFSET : -1
WAIT_TRANS_ID         : 2                     <<= Lock을 점유한 트랜잭션 ID
WAIT_OBJECT           : T1                    <<= Lock OBJECT
WAIT_SLOT_ID          : 0                     <<= Object 내의 레코드 위치
SESSION_STATUS        : transaction
IS_LOGGING            : 0
LOGFILE_NO            : -1
CKPT_NO               : -1
IS_REPL               : 0
REPL_SEND_SCN         : -1
REPL_RECV_ACK_SCN     : -1
AUTOCOMMIT_MODE       : 0
BEGIN_TIME            : 2025/07/18 06:14:09
PROGRAM               : dbmMetaManager
REMOTE_PID            :
REMOTE_ADDR           :
REMOTE_PROGRAM        :
```

```
-------------------------------------------------------------------------------
ID                    : 2
TRANS_ID              : 2
PID                   : 3389780
TID                   : 3389780
OLD_TID               : 3389780
VIEWSCN               : 9223372036854775807
CURR_UNDO_PAGE        : 5
FIRST_UNDO_PAGE       : 5
LAST_UNDO_PAGE        : 5
SAVEPOINT_UNDO_PAGE   : -1
SAVEPOINT_UNDO_OFFSET : -1
WAIT_TRANS_ID         : -1
WAIT_OBJECT           :
WAIT_SLOT_ID          : -1
SESSION_STATUS        : transaction
IS_LOGGING            : 0
LOGFILE_NO            : -1
CKPT_NO               : -1
IS_REPL               : 0
REPL_SEND_SCN         : -1
REPL_RECV_ACK_SCN     : -1
AUTOCOMMIT_MODE       : 0
BEGIN_TIME            : 2025/07/18 06:26:58
PROGRAM               : dbmMetaManager
REMOTE_PID            :
REMOTE_ADDR           :
REMOTE_PROGRAM        :
-------------------------------------------------------------------------------
2 row selected
```

위 결과에서 WAIT_TRANS_ID 항목이 -1이 아닌 경우, 해당 세션이 대상 TransID를 가진 다른 세션을 기다리고 있는 상태를 의미한다.
예를 들어, 위의 예제에서는 ID=1 세션이 ID=2 세션을 기다리고 있음을 나타낸다.
대기 중인 대상 테이블은 T1 이며, 0번 slot을 가진 record에서 기다리고 있다는 의미이다.

위 예제와 같이 TransID=2인 세션이 수행 중인 작업은 다음과 같은 방법으로 조회할 수 있다.

```
dbmMetaManager(DEMO)> select * from v$transaction where trans_id = 2;
-------------------------------------------------------------------------------
TRANS_ID    : 2
TRANS_SEQ   : 1
```

```
TRANS_TYPE   : UPDATE_TABLE
OBJECT_NAME : T1
SLOT_ID      : 0
EXTRA_KEY    : 1
COMMIT_FLAG : 0
SKIP_FLAG    : 0
VALID_FLAG   : 1
--------------------------------------------------------------------------------
1 row selected
```

SlotID=0에 해당하는 record를 변경 중이며, 아직 commit/rollback이 수행되지 않은 상태가 유지되고 있다.
TransID=2를 가진 세션이 어떤 프로세스인지 확인하려면, v$session view의 PID 및 PROGRAM column을 참조하면 된다.
(원격 노드에서 접속한 경우에는 REMOTE_PID, REMOTE_PROGRAM 항목을 통해 확인할 수 있다.)

> 노트
> 각 항목의 의미에 대해서는 **V$SESSION** 및 **V$TRANSACTION** view를 참조한다.

처리량 확인

GODILOCKS LITE는 각 주요 operation에 대한 누적 정보를 기록할 수 있다.
다음과 같은 방법으로 전체 누적 처리량을 확인할 수 있다.

```
dbmMetaManager(DEMO)> select * from v$sys_stat;
--------------------------------------------------------------------------------
STAT_NAME    : init_handle_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME    : free_handle_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME    : prepare_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME    : execute_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
```

```
STAT_NAME   : insert_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : update_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : delete_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : scan_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : enqueue_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : dequeue_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : aging_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : commit_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : rollback_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : recovery_rollback_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : recovery_commit_op
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : split_index_node
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
STAT_NAME   : retry_lock_count
ACCUM_COUNT : 0
--------------------------------------------------------------------------------
17 row selected
```

노트

각 항목의 의미에 대해서는 **V$SYS_STAT** view를 참조한다.

Log Cache 및 Checkpoint 상태

디스크 로깅과 관련된 설정값과 동작 상태를 확인할 수 있다.

```
dbmMetaManager(DEMO)> select * from v$log_stat;
--------------------------------------------------------------------------------
DISKLOG_ENABLE       : 0
CACHE_MODE           : 0
DIRECT_IO_ENABLE     : 0
ARCHIVE_ENABLE       : 0
CURR_FILE_NO         : -1
CURR_FILE_OFFSET     : -1
LAST_CKPT_FILE_NO    : -1
LAST_ARCHIVE_FILE_NO : -1
LAST_CAPTURE_FILE_NO : -1
LOGCACHE_WRITE_IND   : -1
LOGCACHE_READ_IND    : -1
FLUSHER_FILE_NO      : -1
FLUSHER_FILE_OFFSET  : -1
LOG_DIR              : /mnt/md1/ssd_home/lim272/new_lite/pkg/wal
DATAFILE_DIR         : /mnt/md1/ssd_home/lim272/new_lite/pkg/dbf
ARCHIVE_DIR          : /mnt/md1/ssd_home/lim272/new_lite/pkg/arch
--------------------------------------------------------------------------------
1 row selected
```

노트

각 column의 의미에 대해서는 **V$LOG_STAT** view를 참조한다.

2.

API Reference

2.1 API 공통사항

- 모든 API 호출은 정상적으로 처리되면 0을 반환하며, 그 외의 경우에는 에러 코드를 반환한다. (**2.6 Error Message**참조)
- 헤더파일은 $DBM_HOME/include/dbmUserAPI.h 를 사용한다.
- 라이브러리는 $DBM_HOME/lib/libdbmCore.so 를 사용하며, 이를 환경 변수 LD_LIBRARY_PATH에 추가해야 한다.
- 변경 연산 API는 non auto commit 모드로 동작한다.
- dbmHandle은 API 사용을 위한 entry object 로서 thread-safe 하지 않다.
- GOLDILOCKS LITE의 모든 API 는 reentrant를 보장하지 않는다.

2.2 C/C++ APIs

API는 사용자의 선택에 따라 다음 유형의 구조체 변수를 사용한다.

Handle Type	설명
dbmHandle	트랜잭션을 수행하기 위해 instance에 접근할 때 사용하는 변수 타입
dbmTableHandle	테이블에 접근하기 위해 사용하는 변수 타입
dbmStmt	SQL 방식을 사용하기 위한 변수 타입

노트

GOLDILOCKS LITE는 SIGUSR1, SIGUSR2, SIGINT, SIGTERM 시그널에 대해 dbmInitHandle 수행 시 user signal handler를 저장한다. 따라서 사용자 signal handler는 최초 dbmInitHandle 호출 전에 반드시 등록되어 있어야 한다.
이후 해당 signal 이 발생하면 내부 처리 (critical section) 가 완료된 후 user signal handler를 호출한다. User signal handler가 등록되지 않은 경우에는 SIG_DFL 을 기준으로 동작한다. 또한 SA_SIGINFO 가 설정되어 있더라도 stack 정보는 전달되지 않는다.
Critical section 에서 비정상 종료가 발생하면 수동 복구가 필요할 수 있으므로, 가능한 한 LITE library 내부 처리 중에는 프로세스가 중단되지 않도록 graceful-exit 방식으로 구현하는 것을 권장한다.

dbmInitHandle

기능

API 사용을 위한 초기화 작업을 수행한다.
Local server 에서 다른 API를 사용하기 전에 반드시 선행되어 호출되어야 한다.
dbmInitHandle은 아래의 과정을 수행한 후 성공 또는 실패를 반환한다.

- 트랜잭션 수행을 위한 session 등록
  - Instance segment attach 및 관련 자원 할당
- 디스크 모드인 경우 관련 자원 할당
  - Logging mode에 따른 자원 준비
- 이중화 모드인 경우 관련 자원 할당
  - 이중화 작업을 위한 thread 생성 및 원격 연결

인자

```
int dbmInitHandle( dbmHandle     ** aHandle,
                   const char     * aInstanceName )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle **	In/ out	변수는 NULL로 초기화한 후에 사용해야 한다.
aInstanceName	const char *	In	-

> 노트
> aInstanceName이 NULL인 경우, 환경 변수 DBM_INSTANCE 값을 default로 사용한다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );
}
```

노트

dbmHandle 변수는 (void *) 형태이며, dbmInitHandle을 통해 내부적으로 필요한 공간을 할당한 후 사용자에게 반환된다.

주의

사용자 signal handler를 사용하려면 dbmInitHandle 호출 전에 signal handler 등록이 완료되어 있어야 한다. dbmInitHandle 함수는 signal에 대한 지연 처리를 위해 사용자 handler를 저장하며, 시그널이 발생하면 내부 함수 처리가 완료된 후 저장된 handler를 호출한다.

dbmConnect

원격 노드에 접속하기 위해 dbmInitHandle 을 대신하여 사용한다.
이 기능을 사용하려면 원격 노드에서 dbmListener가 실행 중이어야 한다.

인자

```
int dbmConnect( dbmHandle     ** aHandle,
                const char     * aTargetIP,
                int              aTargetPort,
                const char     * aInstName )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle **	In/ out	변수는 NULL로 초기화한 후에 사용해야 한다.
aTargetIP	const char *	In	접속할 원격 노드의 IP 이다.
aTargetPort	int	In	접속할 원격 노드에 구동된 dbmListener의 PortNo 이다.
aInstanceName	const char *	In	접속할 원격 노드의 대상 instance name 이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    int            rc;
    rc =  dbmConnect( &sHandle, "127.0.0.1", 27584, "demo" )
}
```

노트

- 원격 노드에서 dbmListener 가 실행 중이어야 한다.
- Handle을 해제하려면 dbmFreeHandle 함수를 사용한다.

dbmFreeHandle

기능

dbmInitHandle 또는 dbmConnect API를 통해 할당된 자원을 해제한다.

인자

```
int dbmFreeHandle( dbmHandle     ** aHandle )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle **	In/ out	해제 후 변수는 NULL로 초기화 된다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );
    // 사용자코드
    rc = dbmFreeHandle( &sHandle );
}
```

노트

dbmFreeHandle 호출 시점에 완료되지 않은 트랜잭션은 자동으로 롤백 처리 된다.

dbmPrepareTable

기능

입력된 테이블을 사용 가능한 상태로 준비한다.

- Table 및 index shared memory segment를 attach 한다.
- Table 과 column 정보를 검증한다.
- Internal 처리에 필요한 관련 자원을 할당한다.

인자

```
int dbmPrepareTable( dbmHandle       * aHandle,
                     const char      * aTableName );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수를 사용해야 한다.
aTableName	const char *	In	Prepare 대상 TableName을 입력한다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareTable( sHandle,
                          "table1" );
}
```

dbmPrepareTableHandle

기능

dbmPrepareTable 과 동일한 기능을 수행하며, 테이블 핸들을 생성하여 반환한다.
dbmPrepareTable 함수는 사용자가 테이블 이름을 지정하는 API를 사용하는 방식인 반면, dbmPrepareTableHandle 함수는 사용자가 직접 테이블 핸들을 관리할 수 있는 방식이다.

인자

```
int dbmPrepareTableHandle( dbmHandle         * aHandle,
                           const char        * aTableName,
                           dbmTableHandle    ** aTableHandle );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수를 사용해야 한다.
aTableName	const char *	In	Prepare 대상 TableName을 입력한다.
aTableHandle	dbmTableHandle **	out	Prepare 된 table의 handle 이다.

사용 예

```
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareTableHandle( sHandle,
                                "table1",
                                &sTableHandle );
}
```

노트

API 처리 중 대상 table에 DDL이 발생하여 shared memory segment에 변화가 있을 경우, 자동으로 prepare를 재수행한다.
단, 이 과정을 수행할 수 없는 경우에는 에러가 반환될 수 있다.

prepare 과정에서는 새로운 shared memory segment attach가 수행되어 수행 속도에 영향을 줄 수 있으므로, 운영 중 DDL 수행은 권장되지 않는다.

주의

dbmPrepareTableHandle 에 의해 생성된 테이블 핸들도 dbmHandle 내에서 관리되므로, 동일한 테이블명을 연속 호출할 경우 동일한 테이블 핸들 주소가 반환된다.

dbmAuthorize

기능

Instance에 암호가 설정되어 있는 경우, 접근을 허용하기 위해 해당 암호를 검증하는 API이다.
(자세한 내용은 **set password**를 참조한다.)

인자

```
int dbmAuthorize( dbmHandle     * aHandle,
                  const char    * aPassword )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In/ out	dbmInitHandle을 통해 반환된 handle 변수이다.
aPasswd	const char *	In	Instance에 설정된 암호이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmAuthorize( sHandle, "lite_pwd" );
}
```

노트

dbmAuthorize API는 dbmInitHandle 또는 dbmConnect 이후에 호출해야 하며, 인증이 성공한 경우에만 다른 API를 사용할 수 있다.

dbmSetLockTimeout

기능

Lock을 획득하기 위해 대기하는 시간을 지정한다. 해당 설정은 handle 단위로만 적용된다.
값이 "0" 인 경우, timeout은 적용되지 않으며, 단위는 us (마이크로초) 이다.

인자

```
int dbmSetLockTimeout( dbmHandle     * aHandle,
                       long            aTimeout )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In/ out	dbmInitHandle을 통해 반환된 handle 변수이다.
aTimeout	long	In	지정할 Timeout 값을 입력한다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmSetLockTimeout( sHandle, 3000000 );
}
```

노트

지정된 시간 내에 lock을 획득하지 못하면 "ERR-22171] a lock timeout raised" 에러가 발생한다.

dbmPrepareStmt

기능

사용자가 수행할 SQL에 대해 parsing 과 validation을 수행한다.
(SQL syntax에 대한 자세한 내용은 **구문**을 참조한다.)

인자

```
int dbmPrepareStmt( dbmHandle       * aHandle,
                    const char      * aSQLString,
                    dbmStmt        ** aStmt )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수가 사용되어야 한다.
aSQLString	const char *	In	Prepare 할 SQL string 이다.
aStmt	dbmStmt **	In/ out	Prepared 된 stmt가 반환된다. (NULL로 초기화 된 변수여야 한다.)

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmStmt      * sStmt   = NULL;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareStmt( sHandle,
                         "select * from t1 where c1 = :v1",
                         & sStmt );
}
```

노트

- SQL 내에서 parameter를 지정하는 방법은 다음 두 가지이다.
  두 방식은 혼용할 수 없으며, 하나의 SQL 문에서는 동일한 방식으로만 나열해야 한다.
  - Marker (?) 를 사용하여 순서대로 지정하는 방법
  - :v1 과 같이 이름을 명시하여 지정하는 방법

- dbmStmt는 (void *) 형태의 변수이며, dbmPrepareStmt는 prepared 된 결과를 dbmStmt 변수에 반환한다.
- GOLDILOCKS LITE는 별도의 optimization 기법을 제공하지 않는다.
  따라서 SQL 방식의 index scan은 지정된 index의 모든 key column이 binding 된 경우에만 동작하며, 그 외의 경우에는 모두 full scan 방식으로 처리된다.

주의

- dbmPrepareStmt를 통해 생성된 dbmStmt 객체는 서로 다른 handle 간에 공유할 수 없다.
- 동일한 dbmStmt 변수를 사용하여 서로 다른 SQL에 대해 dbmPrepareStmt를 수행하려면, 먼저 dbmFreeStmt를 호출하여 기존 객체를 해제한 후 dbmPrepareStmt 를 다시 호출해야 한다.

dbmFreeStmt

기능

dbmStmt에 할당된 자원을 해제한다.

인자

```
int dbmFreeStmt( dbmHandle       * aHandle,
                 dbmStmt        ** aStmt )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수가 사용되어야 한다.
aStmt	dbmStmt **	In/ out	Prepared 된 stmt pointer를 입력한다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmStmt      * sStmt   = NULL;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareStmt( sHandle,
                         "select * from t1 where c1 = :v1",
                         & sStmt );
    // ....
    rc = dbmFreeStmt( sHandle, &sStmt );
}
```

dbmBindParamById

기능

dbmPrepareStmt에서 사용된 사용자 parameter marker에 대응되는 변수를 binding 한다.

- 사용자 변수는 input mode로만 사용할 수 있다.
- Binding 할 변수의 pointer를 지정해야 하며, dbmPrepareStmt와 dbmExecuteStmt 사이에서 호출해야 한다.
- Binding 되는 data type이 CHAR인 경우 길이 정보가 지정되지 않으면, dbmExecuteStmt 호출 시점에 미리 바인딩 된 변수 값의 길이를 사용한다.
  이로 인해 null-terminated 되지 않은 변수가 사용될 경우, overflow로 인한 오류가 발생할 수 있다.

인자

```
int dbmBindParamById( dbmHandle          * aHandle,
                      dbmStmt            * aStmt,
                      int                  aBindId,
                      dbmBindDataType      aBindDataType,
                      void               * aData,
                      long               * aSizePtr );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	Prepared 된 stmt 변수이다.
aBindId	int	In	Prepare 시점에 나열된 parameter의 순서 (base=1) 이다.
aBindDataType	dbmBindDataType	In	Binding 변수의 데이터 타입이다. • DBM_BIND_DATA_TYPE_SHORT • DBM_BIND_DATA_TYPE_INT • DBM_BIND_DATA_TYPE_DOUBLE • DBM_BIND_DATA_TYPE_FLOAT • DBM_BIND_DATA_TYPE_LONG • DBM_BIND_DATA_TYPE_CHAR • DBM_BIND_DATA_TYPE_DATE • DBM_BIND_DATA_TYPE_TIMESTAMP
aData	void *	In	Binding 할 사용자 변수의 포인터이다.
aSizePtr	long *	In	Binding 할 사용자 변수에 저장된 데이터 크기를 보관하는 8 byte의 변수 포인터이다.

노트

null 값을 포함하는 binary data를 binding 하는 경우, binding data type을 DBM_BIND_DATA_TYPE_CHAR로 설정하고 aSizePtr 변수를 통해 데이터 길이를 지정해야 한다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmStmt      * sStmt   = NULL;
    int            sVar;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareStmt( sHandle,
                         "select * from t1 where c1 = ?",
                         & sStmt );
    rc = dbmBindParamById( sHandle,
                           sStmt,
                           1,
                           DBM_BIND_DATA_TYPE_INT,
                           & sVar,
                           NULL );
}
```

dbmBindParamByName

기능

dbmPrepareStmt에서 사용자가 지정한 parameter 이름을 기준으로 사용자 변수를 binding 한다.

- 사용자 변수는 input mode 로만 사용할 수 있다.
- Binding 할 변수의 pointer를 지정해야 하며, dbmPrepareStmt와 dbmExecuteStmt 사이에서 호출해야 한다.
- Binding 되는 data type이 CHAR인 경우 길이 정보가 지정되지 않으면, dbmExecuteStmt 호출 시점에 미리 바인딩 된 변수 값의 길이를 사용한다.
  이로 인해 null-terminated 되지 않은 변수가 사용될 경우, overflow로 인한 오류가 발생할 수 있다.

인자

```
int dbmBindParamByName( dbmHandle          * aHandle,
                        dbmStmt            * aStmt,
                        char               * aVarName,
                        dbmBindDataType      aBindDataType,
                        void               * aData,
                        long               * aSizePtr );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	Prepared 된 stmt 변수이다.
aVarName	char *	In	Prepare 시점에 나열된 parameter 이름이다.
aBindDataType	dbmBindDataType	In	Binding 변수의 데이터 타입이다. • DBM_BIND_DATA_TYPE_SHORT • DBM_BIND_DATA_TYPE_INT • DBM_BIND_DATA_TYPE_DOUBLE • DBM_BIND_DATA_TYPE_FLOAT • DBM_BIND_DATA_TYPE_LONG • DBM_BIND_DATA_TYPE_CHAR • DBM_BIND_DATA_TYPE_DATE • DBM_BIND_DATA_TYPE_TIMESTAMP
aData	void *	In	Binding할 사용자 변수 포인터이다.
aSizePtr	long *	In	Binding할 사용자 변수의 데이터 크기를 저장하는 8 byte 변수 포인터이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmStmt      * sStmt   = NULL;
    int            sVar;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareStmt( sHandle,
                         "select * from t1 where c1 = :v1",
                         & sStmt );
    rc = dbmBindParamByName( sHandle,
                             sStmt,
                             "v1",
                             DBM_BIND_DATA_TYPE_INT,
                             & sVar,
                             NULL );
}
```

dbmBindCol

기능

dbmPrepareStmt에 사용된 SQL 문이 SELECT 문인 경우, 조회 결과를 지정한 사용자 변수에 저장하기 위해 사용한다.
SQL문 방식에서 데이터를 조회할 때는 다음 순서로 수행해야 한다.
(dbmPrepareStmt → dbmBindCol → dbmExecuteStmt → dbmFetchStmt)
dbmBindCol은 dbmFetchStmt 의 실행 결과를 저장할 사용자 변수를 지정하는 API 이다.

- dbmPrepareStmt 와 dbmExecuteStmt 를 통해 수행되는 질의는 반드시 select 문이어야 한다.
- select target 절에 기술된 각 항목의 크기와 사용자 변수의 크기가 서로 다를 경우, 오류가 발생할 수 있다.

인자

```
int dbmBindCol( dbmHandle          * aHandle,
                dbmStmt            * aStmt,
                int                  aBindIdx,
                dbmBindDataType      aBindType,
                void               * aData,
                long                 aMaxSize,
                long               * aSizePtr )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	Prepared 된 stmt 변수이다.
aBindIdx	int	In	Target column이 나타나는 순서이다. (Base=1)
aBindType	dbmBindDataType	In	Target 변수의 데이터 타입이다.
aData	void *	In	반환되는 사용자 변수 포인터이다.
aMaxSize	long	In	반환되는 사용자 변수의 최대 크기이다
aSizePtr	long *	In	반환되는 데이터 크기를 저장하는 8 byte 변수 포인터이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmStmt      * sStmt   = NULL;
    int            sVar;
```

```
    int             sCol1;
    int             sCol2;
    int             rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareStmt( sHandle,
                         "select c1, c2 from t1 where c1 = :v1",
                         & sStmt );
    rc = dbmBindParamByName( sHandle,
                             sStmt,
                             "v1",
                             DBM_BIND_DATA_TYPE_INT,
                             & sVar,
                             NULL );
    rc = dbmBindCol( sHandle,
                     sStmt,
                     1,
                     DBM_BIND_DATA_TYPE_INT,
                     & sCol1,
                     sizeof(int),
                     NULL );

    rc = dbmBindCol( sHandle,
                     sStmt,
                     2,
                     DBM_BIND_DATA_TYPE_INT,
                     & sCol2,
                     sizeof(int),
                     NULL );
}
```

dbmBindColStruct

기능

dbmBindCol과 동일한 기능을 수행하며, 구조체 변수를 binding 할 때 사용한다.

- dbmPrepareStmt 와 dbmExecuteStmt 를 통해 수행되는 질의는 반드시 select 문이어야 한다.
- dbmBindCol과 혼용하여 사용할 수 없다.
- select target 절에 기술된 각 항목의 크기와 개수가 사용자 변수와 다를 경우, 오류가 발생할 수 있다.

인자

```
int dbmBindColStruct( dbmHandle          * aHandle,
                      dbmStmt            * aStmt,
                      void               * aData )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	Prepared 된 stmt 변수이다.
aData	void *	In	사용자 변수 포인터이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
    int c3;
} DATA;
int main( int argc, char *argv[] )
{
    dbmHandle    * sHandle = NULL;
    dbmStmt      * sStmt = NULL;
    DATA           sData;
    char           sErrMsg[1024];
    int c1;
    int rc;
    rc = dbmInitHandle( &sHandle, "demo" );
    TEST_ERR( sHandle, rc, "initHandle" );
```

```
    c1 = 10;
    rc = dbmPrepareStmt( sHandle,
                         "select * from t1 where c1 = ?",
                         & sStmt );
    TEST_ERR( sHandle, rc, "prepareStmt" );
    // 중략
    rc = dbmBindColStruct( sHandle,
                           sStmt,
                           &sData );
    TEST_ERR( sHandle, rc, "selectTargetBind" );
    rc = dbmExecuteStmt( sHandle, sStmt );
    if( rc )
    {
        dbmGetErrorData( sHandle, &rc, sErrMsg, sizeof(sErrMsg) );
        printf("ERR-%d] %s\n", rc, sErrMsg );
    }
    rc = dbmFetchStmt( sHandle, sStmt );
    TEST_ERR( sHandle, rc, "fetchStmt" );
    printf( "Out c1=%d, c2=%d, c3=%d\n", sData.c1, sData.c2, sData.c3 );
    rc = dbmFreeStmt( sHandle, &sStmt );
    TEST_ERR( sHandle, rc, "FreeStmtInsert" );
    return 0;
}
```

dbmExecuteStmt

기능

dbmPrepareStmt 를 통해 처리된 SQL문을 실행한다.

인자

```
int dbmExecuteStmt( dbmHandle           * aHandle,
                    dbmStmt             * aStmt )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	Prepared 된 stmt 변수이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmStmt      * sStmt   = NULL;
    int            sVar;
    int            sCol1;
    int            sCol2;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareStmt( sHandle,
                         "select c1, c2 from t1 where c1 = :v1",
                         & sStmt );
    rc = dbmBindParamByName( sHandle,
                             sStmt,
                             "v1",
                             DBM_BIND_DATA_TYPE_INT,
                             & sVar,
                             NULL );
    rc = dbmBindCol( sHandle,
                     sStmt,
                     1,
```

```
                    DBM_BIND_DATA_TYPE_INT,
                    & sCol1,
                    sizeof(int),
                    NULL );
    rc = dbmBindCol( sHandle,
                    sStmt,
                    2,
                    DBM_BIND_DATA_TYPE_INT,
                    & sCol2,
                    sizeof(int),
                    NULL );
    rc = dbmExecuteStmt( sHandle,
                         sStmt );
}
```

dbmExecDirect

기능

DDL 및 DML 구문을 SQL 형태로 실행할 때 사용한다.
(단, SELECT 구문은 지원하지 않는다.)

인자

```
int dbmExecDirect( dbmHandle           * aHandle,
                   char                * aSQL )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aSQL	char *	In	수행할 SQL 문장이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;

    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmExecDirect( sHandle,
                        "create table t1 (c1 int, c2 int)" );
}
```

dbmFetchStmt

기능

dbmExecuteStmt 에 의해 수행된 select 문의 조회 결과를 한 건씩 가져와, dbmBindCol을 통해 매핑된 사용자 변수에 저장한다.

- Select 문이 아닌 경우 오류가 발생한다.
- 또한 dbmBindCol 을 통해 사용자 변수가 미리 지정되어야 있어야 하며, 설정이 올바르지 않은 경우 오류가 발생한다.

인자

```
int dbmFetchStmt( dbmHandle          * aHandle,
                  dbmStmt            * aStmt )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	Executed 된 stmt 변수이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmStmt      * sStmt   = NULL;
    int            sVar;
    int            sCol1;
    int            sCol2;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareStmt( sHandle,
                         "select c1, c2 from t1 where c1 = :v1",
                         & sStmt );
    rc = dbmBindParamByName( sHandle,
                             sStmt,
                             "v1",
                             DBM_BIND_DATA_TYPE_INT,
```

```
                            & sVar,
                            NULL );
    rc = dbmBindCol( sHandle,
                     sStmt,
                     1,
                     DBM_BIND_DATA_TYPE_INT,
                     & sCol1,
                     sizeof(int),
                     NULL );
    rc = dbmBindCol( sHandle,
                     sStmt,
                     2,
                     DBM_BIND_DATA_TYPE_INT,
                     & sCol2,
                     sizeof(int),
                     NULL );
    rc = dbmExecuteStmt( sHandle,
                         sStmt );
    while(1)
    {
        rc = dbmFetchStmt( sHandle,
                           sStmt );

        if( rc != 0 )
        {
            break;
        }
    }
}
```

dbmFetchStmt2Json

기능

dbmExecuteStmt 로 수행한 select 문의 조회 결과를 한 건씩 JSON format text로 반환한다.

- Select 문이 아닌 경우 오류가 발생한다.
- json format string이 추가되므로 실제 반환되는 데이터의 크기는 레코드 크기보다 커질 수 있다. 따라서 사용자 변수의 크기를 충분히 고려하여 설정해야 한다.

인자

```
int dbmFetchStmt2Json( dbmHandle          * aHandle,
                       dbmStmt            * aStmt,
                       char               * aJsonPtr )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	Execute 된 stmt 변수이다.
aJsonPtr	char *	In/Out	JSON format 결과가 저장될 사용자 변수 포인터이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmStmt      * sStmt   = NULL;
    char           sText[1024];
    int            sVar;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareStmt( sHandle,
                         "select c1, c2 from t1 where c1 = :v1",
                         & sStmt );
    rc = dbmBindParamByName( sHandle,
                             sStmt,
                             "v1",
                             DBM_BIND_DATA_TYPE_INT,
```

```
                              & sVar,
                              NULL );
    while( dbmFetchStmt2Json( sHandle, sStmt, sText ) == 0 )
    {
        fprintf( stdout, "%s\n", sText );
    }
}
```

dbmInsertRow

기능

사용자 변수에 저장된 데이터를 지정한 테이블에 삽입한다.
사용자는 테이블의 형상에 맞는 offset 및 size 정보를 기준으로 데이터를 사용자 변수에 저장한 후 호출해야 한다.
(offset, size 등의 정보는 **create table**, **desc** 등의 구문을 참조한다.)

인자

```
int dbmInsertRow( dbmHandle          * aHandle,
                  const char         * aTableName,
                  void               * aUserData,
                  int                  aDataSize )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aUserData	void *	In	사용자 변수 포인터이다.
aDataSize	int	In	데이터 크기이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 1;
    sData.c2 = 100;
    rc = dbmInsertRow( sHandle,
                       "t1",
                       & sData,
```

```
                        sizeof(DATA) );
}
```

dbmInsert

기능

dbmInsertRow 와 동일한 기능을 수행하며, 다음과 같은 차이점이 있다.

- dbmPrepareTableHandle 을 통해 생성된 테이블 핸들 변수를 사용한다.
- 데이터 저장에 성공하면 테이블 내 레코드의 저장 위치를 반환받을 수 있다.

인자

```
int dbmInsert( dbmHandle       * aHandle,
               dbmTableHandle  * aTableHandle,
               void            * aUserData,
               int               aDataSize,
               long            * aSlotId )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 테이블 핸들이다.
aUserData	void *	In	사용자 변수 포인터이다.
aDataSize	int	In	데이터 크기이다.
aSlotId	long *	out	NULL이 아닌 경우 slot ID를 반환한다.

노트

Slot ID는 테이블 내에서 레코드가 저장된 고유한 위치 정보를 의미한다.
반환된 Slot ID를 활용하면 dbmUpdate/ dbmSelect/ dbmDelete 등의 함수 수행 시 index 검색 비용을 줄일 수 있다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
```

```
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    DATA             sData;
    long             sSlotId;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareTableHandle( sHandle,
                                "t1",
                                &sTableHandle );
    sData.c1 = 1;
    sData.c2 = 100;
    rc = dbmInsert( sHandle,
                    sTableHandle,
                    & sData,
                    sizeof(DATA),
                    & sSlotId );
}
```

dbmUpdateRow

기능

사용자 입력 변수에 저장된 key와 일치하는 하나 이상의 데이터를 사용자 변수의 값으로 갱신한다.
즉, 사용자 입력 변수에서 index key column에 해당하는 값을 기준으로 데이터를 검색한 후 변경 연산을 수행한다.

인자

```
int dbmUpdateRow( dbmHandle          * aHandle,
                  const char         * aTableName,
                  void               * aUserData,
                  int                * aRowCount )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aUserData	void *	In	사용자 변수 포인터이다.
aRowCount	int *	Out	Update 된 row 개수이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sRowCount = 0;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 1;
    sData.c2 = 100;
    rc = dbmUpdateRow( sHandle,
                       "t1",
                       & sData,
```

```
                    & sRowCount );
}
```

노트

- Update 동작은 테이블 내에서 일치하는 레코드 위치에 사용자 버퍼의 내용을 그대로 복사하는 과정이다. 따라서 사용자 버퍼에는 전체 데이터가 모두 포함되어 있어야 한다. 특정 column만 변경하려는 경우에는 **dbmUpdateRowByCols**를 사용해야 한다
- Key column update는 지원하지 않는다.

dbmUpdate

기능

dbmUpdateRow와 동일한 기능을 수행하며, 다음과 같은 차이점이 있다.

- dbmPrepareTableHandle에 의해 생성된 테이블 핸들 변수를 사용한다.
- 레코드 위치 정보를 이용할 경우 index 탐색 비용을 줄일 수 있다.
- 필요한 경우 변경 전 data를 반환받을 수 있다.

인자

```
int dbmUpdate( dbmHandle      * aHandle,
               dbmTableHandle * aTableHandle,
               void           * aUserData,
               long             aSlotId,
               int            * aRowCount,
               void           * aReturnOldData );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 테이블의 핸들이다.
aUserData	void *	In	사용자 변수 포인터이다.
aSlotId	long	in	-1 이 아닐 경우, 입력된 레코드 위치의 데이터를 변경한다.
aRowCount	int *	Out	Update 된 row 개수이다.
aReturnOldData	void *	Out	NULL이 아닌 경우 변경 전 data를 반환한다.

> 노트
> 이전 데이터 반환 기능은 unique index를 사용하는 경우에만 지원된다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
```

```
} DATA;
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    DATA             sData;
    DATA             sOldData;
    int              sRowCount = 0;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareTableHandle( sHandle,
                                "t1",
                                & sTableHandle );
    sData.c1 = 1;
    sData.c2 = 100;
    rc = dbmUpdate( sHandle,
                    sTableHandle,
                    & sData,
                    -1,
                    & sRowCount,
                    & sOldData );
}
```

dbmUpsertRow

기능

사용자 입력 변수에 저장된 key와 일치하는 데이터가 존재하면 변경 (update)하고, 존재하지 않으면 삽입 (insert)한다.

인자

```
int dbmUpsertRow( dbmHandle  * aHandle,
                  const char * aTableName,
                  void       * aUserData,
                  int          aDataSize );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aUserData	void *	In	사용자 변수 포인터이다.
aDataSize	int	in	aUserData의 크기이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle      * sHandle = NULL;
    DATA             sData;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 1;
    sData.c2 = 100;
    rc = dbmUpsertRow( sHandle,
                       "t1",
                       & sData,
                       sizeof(DATA) );
```

```
}
```

dbmUpsert

기능

dbmUpsertRow와 동일한 기능을 수행하며, 다음과 같은 차이점이 있다.

- dbmPrepareTableHandle에 의해 생성된 테이블 핸들 변수를 사용한다.
- 데이터가 변경된 경우, 필요에 따라 변경 전 데이터를 반환받을 수 있다.

인자

```
int dbmUpsert( dbmHandle        * aHandle,
               dbmTableHandle   * aTableHandle,
               void             * aUserData,
               int                aDataSize,
               void             * aReturnOldData,
               int              * aInsertCnt );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 테이블의 핸들이다.
aUserData	void *	In	사용자 변수 포인터이다.
aDataSize	int	in	aUserData의 크기
aReturnOldData	void *	Out	NULL이 아닌 경우 변경 전 data를 반환한다.
aInsertCnt	int *	Out	NULL이 아닌 경우 • Insert 성공 시, 1을 반환한다. • Update 성공 시, 0을 반환한다.

> 노트
>
> aReturnOldData 와 aInsertCnt 항목은 unique index 를 사용하는 경우에만 동작한다.

사용 예

```
typedef struct
{
    int c1;
```

```
    int c2;
} DATA;
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    DATA             sData;
    DATA             sOldData;
    int              sInsertCnt = 0;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareTableHandle( sHandle,
                                "t1",
                                & sTableHandle );
    sData.c1 = 1;
    sData.c2 = 100;
    rc = dbmUpsert( sHandle,
                    sTableHandle,
                    & sData,
                    sizeof(DATA),
                    & sOldData,
                    & sInsertCnt);
}
```

dbmDeleteRow

기능

사용자 입력 변수에 저장된 key와 일치하는 데이터를 삭제한다.

인자

```
int dbmDeleteRow( dbmHandle          * aHandle,
                  const char         * aTableName,
                  void               * aUserData,
                  int                * aRowCount )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aUserData	void *	In	사용자 변수 포인터이다.
aRowCount	int *	Out	Delete 된 row 개수이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sRowCount = 0;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 1;
    sData.c2 = 100;
    rc = dbmDeleteRow( sHandle,
                       "t1",
                       & sData,
                       & sRowCount );
```

```
}
```

dbmDelete

기능

dbmDeleteRow와 동일한 동작을 수행하며, 다음과 같은 차이점이 있다.

- dbmPrepareTableHandle에 의해 생성된 테이블 핸들 변수를 사용한다.
- 레코드 위치 정보를 이용할 경우 index 탐색 비용을 줄일 수 있다.
- 필요한 경우 삭제 전 데이터를 반환받을 수 있다.

인자

```
int dbmDelete( dbmHandle      * aHandle,
               dbmTableHandle * aTableHandle,
               void           * aUserData,
               long             aSlotId,
               int            * aRowCount,
               void           * aReturnOldData );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 테이블의 핸들이다.
aUserData	void *	In	사용자 변수 포인터이다.
aSlotId	long	in	-1 이 아닌 경우 해당 slot의 데이터를 삭제한다.
aRowCount	int *	Out	NULL이 아닌 경우 delete 된 row 개수이다.
aReturnOldData	void *	Out	NULL이 아닌 경우 delete 된 row의 이전 데이터이다.

> 노트
>
> aReturnOldData 항목은 unique index를 사용하는 경우에만 동작한다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
```

```
} DATA;
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    DATA             sData;
    DATA             sOldData;
    int              sRowCount = 0;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareTableHandle( sHandle,
                                "t1",
                                & sTableHandle );
    sData.c1 = 1;
    sData.c2 = 100;
    rc = dbmDelete( sHandle,
                    sTableHandle,
                    & sData,
                    -1,
                    & sRowCount,
                    & sOldData );
}
```

주의

Btree table의 경우 delete 연산이 수행되더라도 index에서의 key 삭제는 commit 시점에 반영된다. 반면 s play table type에서는 delete 수행 시 key가 즉시 삭제된다. 따라서 commit이 완료되지 않은 상태에서도, s play table에서 delete 된 데이터는 다른 세션에서 조회할 수 없다.
또한, delete를 포함한 트랜잭션이 rollback 되는 시점에, 이미 다른 세션에서 동일한 key가 삽입된 경우에는 delete rollback 처리가 수행되지 않고 skip 된다.

dbmBindColumn

기능

dbmUpdateRowByCols 호출 시 특정 column에 사용자 데이터를 binding 하기 위해 사용한다.
dbmBindColumn 수행 시 사용자 데이터는 내부 임시 버퍼에 복사된다. 따라서 실행 전에 반드시 dbmBindColumn을 호출해야 한다.
동일한 column에 대해 dbmBindColumn을 반복 호출할 경우, 마지막으로 호출한 값이 적용된다. 또한 존재하지 않는 column에 대해 호출할 경우 오류가 발생한다.

인자

```
int dbmBindColumn( dbmHandle     * aHandle,
                   const char    * aTableName,
                   const char    * aColumnName,
                   void          * aUserData,
                   int             aDataSize );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aColumnName	const char *	In	대상 column 이름이다.
aUserData	void *	In	사용자 데이터 포인터이다.
aDataSize	int	In	aUserData의 크기 (byte) 이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    char           sData[1024];
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmBindColumn( sHandle,
                        "t1",
                        "col1",
                        sData,
                        sizeof(sData) );
}
```

dbmClearBind

기능

dbmBindColumn으로 설정된 binding 정보를 삭제한다.
특정 column만 지정할 수 있으며 column을 명시하지 않고 NULL로 지정할 경우 기존에 binding 된 모든 정보를 삭제한다.

인자

```
int dbmClearBind( dbmHandle    * aHandle,
                  const char   * aTableName,
                  const char   * aColumnName );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aColumnName	const char *	In	대상 column 이름이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    char           sData[1024];
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmClearBind( sHandle,
                       "t1",
                       "col1" );
    rc = dbmClearBind( sHandle,
                       "t1",
                       NULL );
}
```

dbmUpdateRowByCols

기능

특정 column만 갱신하려는 경우에 사용한다. 호출 전에 반드시 dbmBindColumn API를 이용하여 key와 변경 대상 column 값을 미리 binding해야 한다.

인자

```
int dbmUpdateRowByCols( dbmHandle      * aHandle,
                        const char     * aTableName,
                        int            * aRowCount );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aRowCount	int *	Out	갱신된 row count를 반환한다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    char           sData[1024];
    int            sRowCount;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmBindColumn( sHandle,
                        "t1",
                        "col1",
                        sData,
                        sizeof(sData) );
    rc = dbmUpdateRowByCols( sHandle,
                             "t1",
                             &sRowCount );
}
```

dbmUpdateByCols

기능

dbmUpdateRowByCols와 동일한 기능을 수행하며, 다음과 같은 차이점이 있다.

- dbmPrepareTableHandle에 의해 생성된 테이블 핸들 변수를 사용한다.
- 레코드 위치 정보를 이용할 경우 index 탐색 비용을 줄일 수 있다.
- 필요한 경우 변경 전 data를 반환받을 수 있다. 단, 해당 기능은 unique index에서만 동작한다.

인자

```
int dbmUpdateByCols( dbmHandle        * aHandle,
                     dbmTableHandle   * aTableHandle,
                     dbmInt64           aSlotId,
                     dbmInt32         * aRowCount,
                     void             * aReturnOldData )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 테이블 핸들이다.
aSlotId	long	In	-1이 아닐 경우, 입력된 레코드 위치의 데이터를 변경한다.
aRowCount	int *	Out	Update 된 row 개수이다.
aReturnOldData	void *	Out	NULL이 아닐 경우 변경 전 data를 반환한다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
    int c3;
} DATA;
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    DATA             sData;
    long             slotId = 3;
```

```
    int              sRowCount = 0;
    int              rc;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareTableHandle( sHandle,
                                "t1",
                                & sTableHandle );
    sData.c2 = 1000;
    rc = dbmBindColumn( sHandle,
                        "t1",
                        "c2",
                        sData.c2,
                        sizeof(sData) );
    rc = dbmUpdateByCols( sHandle,
                          sTableHandle,
                          slotId,
                          &sRowCount,
                          NULL );
}
```

dbmSelectRow

기능

사용자 입력 변수에 저장된 key와 일치하는 데이터를 반환한다.
일치하는 데이터가 2건 이상 존재하는 경우에는 dbmFetchNext 계열의 API를 추가로 사용해야 한다.

인자

```
int dbmSelectRow( dbmHandle          * aHandle,
                  char               * aTableName,
                  void               * aUserData )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 테이블 이름이다.
aUserData	void *	In/ out	사용자 변수 포인터이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sRowCount = 0;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 1;
    rc = dbmSelectRow( sHandle,
                       "t1",
                       & sData );
}
```

> 노트
> dbmSelectRow, dbmUpdateRow, dbmDeleteRow 등은 사용자 데이터에 key 값이 포함된 상태여야 한다. dbmSelectRow는 key가 저장된 사용자 변수에 조회 결과를 반환한다.

또한 date 타입 값을 반환받아야 하는 경우에는, 사용자가 unsigned long long 형태의 8Byte 변수를 지정하여 값을 저장할 수 있다.

자세한 내용은 다음 예제를 참고한다.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>
#include <time.h>
#include <dbmUserAPI.h>
typedef struct
{
    int c1;
    long long c2;     // Date 타입을 반환받을 변수
} DATA;
main()
{
    dbmHandle *sHandle = NULL;
    struct timeval ss;
    time_t now;
    struct tm *nowtm;
    char buf[200];
    DATA sData;
    int sRowCount;
    int rc;
    rc = dbmInitHandle( &sHandle, "demo" );
    if( rc )
    {
        printf( "test fail1\n" );
    }
    sData.c1 = 1;
    rc = dbmSelectRow( sHandle, "t1", &sData );
    if( rc )
    {
```

```
        printf( "test fail2\n" );
    }
```

- Date 값을 string format으로 변환한다.

```
ss.tv_sec = sData.c2 / 1000000.0;
ss.tv_usec = sData.c2 % 1000000;
now = ss.tv_sec;
nowtm = localtime(&now);
strftime( buf, sizeof(buf), "%Y-%m-%d %H:%M:%S", nowtm);
printf( "c1=%d, c2=%s.%ld\n", sData.c1, buf, ss.tv_usec );
```

- 현재 시각으로 변경할 경우

```
gettimeofday( &ss, NULL );
sData.c2 = ss.tv_sec * 1000000LL + ss.tv_usec;
rc = dbmUpdateRow( sHandle, "t1", &sData, &sRowCount );
if( rc )
{
    printf( "test fail3\n" );
}
rc = dbmSelectRow( sHandle, "t1", &sData );
if( rc )
{
    printf( "test fail4\n" );
}
ss.tv_sec = sData.c2 / 1000000.0;
ss.tv_usec = sData.c2 % 1000000;
now = ss.tv_sec;
nowtm = localtime(&now);
strftime( buf, sizeof(buf), "%Y-%m-%d %H:%M:%S", nowtm);
printf( "c1=%d, c2=%s.%ld\n", sData.c1, buf, ss.tv_usec );
dbmCommit( sHandle );
}
================================
== Test
================================
shellPrompt> ./testPgm
c1=1, c2=2020-12-23 17:48:38.913314
c1=1, c2=2020-12-23 17:49:33.214220
```

dbmSelect

기능

dbmSelectRow와 동일한 기능을 수행하며, 다음과 같은 차이점이 있다.

- dbmPrepareTableHandle에 의해 생성된 테이블 핸들을 사용한다.
- 탐색 방향과 종료 조건을 지정할 수 있다.

인자

```
int dbmSelect( dbmHandle         * aHandle,
               dbmTableHandle    * aTableHandle,
               void              * aUserData,
               void              * aUntilData,
               dbmScanDirection    aScanDir,
               dbmScanType         aScanType );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 테이블의 핸들이다.
aUserData	void *	In/ out	시작 조건이 되는 사용자 변수 포인터이다. 검색 결과 데이터가 저장되는 버퍼 공간이기도 하다.
aUntilData	void *	In	종료 조건이 되는 사용자 변수 포인터 이다. 종료 조건이 없을 경우 NULL을 명시한다
aScanDir	dbmScanDirection	In	INDEX 탐색 방향을 지정한다. • DBM_SCAN_DIR_BACKWARD: 인덱스 역방향 탐색 (Equal 결과 제외) • DBM_SCAN_DIR_FORWARD: 인덱스 정방향 탐색 (Equal 결과 제외) • DBM_SCAN_DIR_EQUAL: 동일한 값을 가지는 key 탐색
aScanType	dbmScanType	in	레코드 lock 점유 여부를 설정한다. • DBM_SCAN_TYPE_RDONLY: 레코드에 lock을 설정하지 않음 • DBM_SCAN_TYPE_FOR_UPDATE: 레코드에 lock 을 설정

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    DATA             sData;
    int              sRowCount = 0;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareTableHandle( sHandle,
                                "t1",
                                & sTableHandle );
    sData.c1 = 1;
    rc = dbmSelect( sHandle,
                    sTableHandle,
                    & sData,
                    NULL,
                    DBM_SCAN_DIR_EQUAL,
                    DBM_SCAN_TYPE_RDONLY );
}
```

노트

- DBM_SCAN_DIR_FORWARD는 index 정렬 기준으로 aUserData 다음 위치부터 순방향으로 데이터를 반환한다.
- DBM_SCAN_DIR_BACKWARD는 index 정렬 기준으로 aUserData 이전 위치부터 역방향으로 데이터를 반환한다.
- DBM_SCAN_DIR_EQUAL은 aUserData와 key 값이 동일한 데이터를 반환한다.
- aUntilData의 조건은 index의 정렬 순서를 기준으로 판단하며, 지정한 범위를 초과할 경우 NOT_FOUND 에러를 반환한다.

주의

Auto commit mode에서는 SCAN_TYPE을 DBM_SCAN_TYPE_FOR_UPDATE로 지정할 수 없다.

dbmSetIndex

기능

지정한 테이블에서 사용할 index를 설정한다.

인자

```
int dbmSetIndex( dbmHandle          * aHandle,
                 const dbmChar      * aTableName,
                 const dbmChar      * aIndexName )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	테이블 이름을 입력한다.
aIndexName	const char *	In	사용할 index 이름을 입력한다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    DATA             sData;
    DATA             sUntilData;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareTableHandle( sHandle,
                                "t1",
                                & sTableHandle );
    rc = dbmSetIndex( sHandle, "T1", "IDX2_T1" );
}
```

dbmFetch

기능

dbmSelect 수행 이후, 지정된 탐색 방향에 따라 다음 데이터를 조회한다.

인자

```
int dbmFetch( dbmHandle         * aHandle,
              dbmTableHandle    * aTableHandle,
              void              * aUserData,
              void              * aUntilData,
              dbmScanDirection    aScanDir,
              dbmScanType         aScanType );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 테이블의 핸들이다.
aUserData	void *	out	검색된 결과 데이터가 저장되는 버퍼 공간이다.
aUntilData	void *	in	종료 조건이 되는 사용자 변수 포인터 이다. 종료 조건이 없을 경우 NULL을 명시한다 (aUntilData 값은 결과에 포함된다)
aScanDir	dbmScanDirection	in	INDEX 탐색 방향을 지정한다. • DBM_SCAN_DIR_BACKWARD: 인덱스 역방향 탐색 • DBM_SCAN_DIR_FORWARD: 인덱스 정방향 탐색 • DBM_SCAN_DIR_EQUAL: 동일한 값을 가지는 key 탐색
aScanType	dbmScanType	in	레코드 lock 설정 여부를 설정한다. • DBM_SCAN_TYPE_RDONLY: 레코드에 lock을 설정하지 않음 • DBM_SCAN_TYPE_FOR_UPDATE: 레코드에 lock을 설정

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
```

```
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    DATA             sData;
    DATA             sUntilData;
    rc = dbmInitHandle( &sHandle, "demo" );

    rc = dbmPrepareTableHandle( sHandle,
                                "t1",
                                & sTableHandle );
    sData.c1 = 10;
    sUntilData.c1 = 100;
    rc = dbmSelect( sHandle,
                    sTableHandle,
                    & sData,
                    & sUntilData,
                    DBM_SCAN_DIR_FORWARD,
                    DBM_SCAN_TYPE_RDONLY );
   while( 1 )
   {
       rc = dbmFetch( sHandle,
                      sTableHandle,
                      & sData,
                      & sUntilData,
                      DBM_SCAN_DIR_FORWARD,
                      DBM_SCAN_TYPE_RDONLY );
       if( rc != 0 ) break;
   }
}
```

노트

- DBM_SCAN_DIR_FORWARD는 index 정렬 기준으로 aUserData 다음 위치부터 순방향으로 데이터를 반환한다.
- DBM_SCAN_DIR_BACKWARD는 index 정렬 기준으로 aUserData 이전 위치부터 역방향으로 데이터를 반환한다.
- DBM_SCAN_DIR_EQUAL은 aUserData와 key 값이 동일한 데이터를 반환한다. (Non unique index 사용 시)
- aUntilData의 조건은 index의 정렬 순서를 기준으로 판단하며, 지정한 범위를 초과할 경우 NOT_FOUND 에러를 반환한다.

주의

Auto commit mode에서는 SCAN_TYPE을 DBM_SCAN_TYPE_FOR_UPDATE로 지정할 수 없다.

dbmSelectRowGT

기능

Index 정렬 기준으로 순방향 탐색 결과를 반환한다. 이 때 사용자 데이터에 해당하는 값은 결과에 포함되지 않는다.
예를 들어 index 정렬 순서가 (1, 2, 3, 4) 이고 사용자 버퍼에 저장된 key가 "2"인 경우 (3)을 반환한다.
또한 index 정렬 순서가 (4, 3, 2, 1)이고 사용자 버퍼에 저장된 key가 "2"인 경우에는 (1)을 반환한다.

인자

```
int dbmSelectRowGT( dbmHandle          * aHandle,
                    char               * aTableName,
                    void               * aUserData )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 테이블 이름이다.
aUserData	void *	In/ out	사용자 변수 포인터이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 1;
    rc = dbmSelectRowGT( sHandle,
                         "t1",
                         & sData );
}
```

dbmSelectRowLT

기능

Index 정렬 기준으로 역방향 탐색 결과를 반환한다. 이 때 사용자 데이터에 해당하는 값은 결과에 포함되지 않는다.
예를 들어 index 정렬 순서가 (1, 2, 3, 4) 이고 사용자 버퍼에 저장된 key가 "2"인 경우 (1)을 반환한다.
또한 index의 정렬 순서가 (4, 3, 2, 1)이고 사용자 버퍼에 저장된 key가 "2"인 경우에는 (3)을 반환한다.

인자

```
int dbmSelectRowLT( dbmHandle          * aHandle,
                    char               * aTableName,
                    void               * aUserData )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 테이블 이름이다.
aUserData	void *	In/ out	사용자 변수 포인터이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 10;
    rc = dbmSelectRowLT( sHandle,
                         "t1",
                         & sData );
}
```

dbmFetchNext

기능

입력된 데이터의 key 값과 동일한 다음 데이터를 조회한다.
Non-unique index에서 동일한 key 값을 가지는 레코드를 조회하기 위해 사용한다.
반드시 dbmSelectRow 호출 이후에 사용해야 한다.

인자

```
int dbmFetchNext( dbmHandle          * aHandle,
                  char               * aTableName,
                  void               * aUserData )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 테이블 이름이다.
aUserData	void *	In/ out	사용자 변수 포인터이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 10;
    rc = dbmSelectRow( sHandle,
                       "t1",
                       & sData );
   while( 1 )
   {
       rc = dbmFetchNext( sHandle,
```

```
                        "t1",
                        &sData );
        if( rc != 0 ) break;
    }
}
```

dbmFetchNextGT

기능

Index 정렬 기준으로 순방향 탐색 결과를 반환한다. 이 때 사용자 데이터에 해당하는 값은 결과에 포함되지 않는다.
예를 들어 index 정렬 순서가 (1, 2, 3, 4) 이고 사용자 버퍼에 저장된 key가 "2"인 경우 (3)을 반환한다.
또한 index 정렬 순서가 (4, 3, 2, 1)이고 사용자 버퍼에 저장된 key가 "2"인 경우 (1)을 반환한다.
반드시 dbmSelectRow 호출 이후에 사용해야 한다.

인자

```
int dbmFetchNextGT( dbmHandle           * aHandle,
                    char                * aTableName,
                    void                * aUserData )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 테이블 이름이다.
aUserData	void *	In/ out	사용자 변수 포인터이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 10;
    rc = dbmSelectRow( sHandle,
                       "t1",
                       & sData );
   while( 1 )
   {
```

```
        rc = dbmFetchNextGT( sHandle,
                             "t1",
                             &sData );
        if( rc != 0 ) break;
    }
}
```

dbmFetchNextLT

기능

Index 정렬 기준으로 역방향 탐색 결과를 반환한다. 이 때 사용자 데이터에 해당하는 값은 결과에 포함되지 않는다.
예를 들어 index 정렬 순서가 (1, 2, 3, 4) 이고 사용자 버퍼에 저장된 key가 "2"인 경우 (1)을 반환한다.
또한 index 정렬 순서가 (4, 3, 2, 1)이고 사용자 버퍼에 저장된 key가 "2"인 경우 (3)을 반환한다.
반드시 dbmSelectRow 호출 이후에 사용해야 한다.

인자

```
int dbmFetchNextLT( dbmHandle           * aHandle,
                    char                * aTableName,
                    void                * aUserData )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 테이블 이름이다.
aUserData	void *	In/ out	사용자 변수 포인터이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 10;
    rc = dbmSelectRow( sHandle,
                       "t1",
                       & sData );
   while( 1 )
   {
```

```
        rc = dbmFetchNextLT( sHandle,
                             "t1",
                             &sData );
        if( rc != 0 ) break;
    }
}
```

dbmSelectForUpdateRow

기능

dbmSelectRow와 동일하게 동작하며, 조회된 레코드에 대해 lock을 획득한다.

인자

```
int dbmSelectForUpdateRow( dbmHandle          * aHandle,
                           char               * aTableName,
                           void               * aUserData )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 테이블 이름이다.
aUserData	void *	In/ out	사용자 변수 포인터이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 10;
    rc = dbmSelectForUpdateRow( sHandle,
                                "t1",
                                & sData );
}
```

노트

다른 갱신 DML들과 마찬가지로 dbmSelectForUpdateRow를 수행한 경우에는 반드시 dbmCommit 또는 dbmRollback을 호출하여 트랜잭션을 완료해야 한다.

조회와 동시에 레코드 lock을 유지하는 API는 다음과 같다.

- dbmSelectForUpdateRowGT
- dbmSelectForUpdateRowLT
- dbmFetchNextUpdateRowGT
- dbmFetchNextUpdateRowLT

주의

Auto commit mode에서는 lock을 점유하지 않는다.

dbmInsertArray

기능

다수의 레코드를 삽입할 때 사용한다. Array 방식은 성능상의 장점은 없으나, 원격 노드에서 처리하는 경우 통신 횟수를 줄이기 위한 목적으로 사용할 수 있다.
사용자가 입력한 데이터를 모두 처리한 후 오류가 발생하면 함수는 "1"을 반환하며, 개별 에러 코드는 사용자 에러 코드 변수에 저장된다.

인자

```
int dbmInsertArray( dbmHandle          * aHandle,
                    char               * aTableName,
                    void               * aDataPtr,
                    int                  aDataSingleSize,
                    int                  aDataCount,
                    int                  aRetArr[] )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 테이블 이름이다.
aDataPtr	void *	In/ out	배열 데이터의 시작 주소를 가리키는 포인터이다.
aDataSingleSize	int	In	데이터 1 건의 크기를 지정한다.
aDataCount	int	In	aDataPtr 변수에 저장된 데이터 개수를 지정한다.
aRetArr	int	Out	각 operation의 에러 코드를 순서대로 저장한다.

노트

- aDataPtr은 (aDataSingleSize X aDataCount) bytes와 같은 크기여야 한다.
- aDataSingleSize는 table에 저장 가능한 레코드의 크기를 의미하며, **desc** 정보를 통해 확인할 수 있다.
- aRetArr 은 에러가 발생한 array index 위치에 에러 코드만 설정하여 반환한다.

사용 예

```
typedef struct
{
    int c1;
```

```
    int c2;
    int c3;
} DATA;
int main( int argc, char *argv[] )
{
    dbmHandle    * sHandle = NULL;
    DATA           sData[10];
    int            sErrCode[10];
    int            i;
    int            rc;
    rc = dbmInitHandle( &sHandle, "demo" );
    if( rc ) exit(-1);
    for( i = 0; i < 10; i ++ )
    {
        sData[i].c1 = i;
        sData[i].c2 = i;
        sData[i].c3 = i;
    }
```

- 정상 처리 예제

```
rc = dbmInsertArray( sHandle, "t1", sData, sizeof(DATA), 10, sErrCode );
if( rc ) exit(-1);
rc = dbmCommit( sHandle );
if( rc ) exit(-1);
```

- 오류 처리 예제

```
rc = dbmInsertArray( sHandle, "t1", sData, sizeof(DATA), 10, sErrCode );
for( i = 0 ; i < 10; i ++ )
{
    printf( "%d] retCode = %d\n", i, sErrCode[i] );
}
dbmFreeHandle( &sHandle );
return 0;
```

주의

- 사용자가 입력한 정보 (레코드 크기, 데이터 개수, 버퍼 크기)가 올바르지 않을 경우 잘못된 포인터 접근으로 인해 오동작이 발생할 수 있으므로 주의해야 한다.
- Auto commit mode에서는 처리에 성공한 데이터가 자동으로 commit 된다.

dbmUpdateArray

기능

다수의 레코드를 변경할 때 사용한다. Array 방식은 성능상의 이점은 없으나, 원격 노드에서 처리하는 경우 통신 횟수를 줄이기 위한 목적으로 사용할 수 있다.
사용자가 입력한 데이터를 모두 처리한 후 오류가 발생하면 함수는 "1"을 반환하며, 개별 에러 코드는 사용자 에러 코드 변수에 저장된다.

인자

```
int dbmUpdateArray( dbmHandle          * aHandle,
                    const char         * aTableName,
                    void               * aDataPtr,
                    int                  aDataCount,
                    int                * aRowCount,
                    int                  aRetArr[] )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aDataPtr	void *	In/ out	배열 데이터의 시작 주소를 가리키는 포인터이다.
aDataCount	int	In	aDataPtr 변수에 저장된 데이터 개수를 지정한다.
aRowCount	int	Out	전체 처리 건수를 반환한다.
aRetArr	int	Out	각 operation의 에러 코드를 순서대로 저장한다.

노트

- aDataPtr은 (Record 크기 X aDataCount) bytes와 같은 크기여야 한다.
- Record 크기는 table에 저장 가능한 레코드의 크기를 의미하며, **desc** 정보를 통해 확인할 수 있다.
- aRetArr 은 에러가 발생한 array index 위치에 에러 코드만 설정하여 반환한다.

사용 예

```
typedef struct
{
    int c1;
```

```
    int c2;
    int c3;
} DATA;
int main( int argc, char *argv[] )
{
    dbmHandle    * sHandle = NULL;
    DATA           sData[10];
    int sErrCode[10], sRowCount;
    int i;
    int rc;
    rc = dbmInitHandle( &sHandle, "demo" );
    if( rc ) exit(-1);
    for( i = 0; i < 10; i ++ )
    {
        sData[i].c1 = i;
        sData[i].c2 = i;
        sData[i].c3 = i;
    }
    rc = dbmInsertArray( sHandle, "t1", sData, sizeof(DATA), 10, sErrCode );
    if( rc ) exit(-1);
    rc = dbmCommit( sHandle );
    if( rc ) exit(-1);
    for( i = 0; i < 10; i ++ )
    {
        sData[i].c1 = i;
        sData[i].c2 = i*10;
        sData[i].c3 = i*20;
    }
    rc = dbmUpdateArray( sHandle, "t1", sData, 10, &sRowCount, sErrCode );
    if( rc ) exit(-1);
    printf( "AffectedRow = %d\n", sRowCount );
    rc = dbmCommit( sHandle );
    if( rc ) exit(-1);
    dbmFreeHandle( &sHandle );
    return 0;
}
```

주의

- 사용자가 입력한 정보 (버퍼 크기, 개수)가 올바르지 않을 경우 잘못된 포인터 접근으로 인해 오동작이 발생할 수 있으므로 주의해야 한다.

- Auto commit mode에서는 처리에 성공한 데이터가 자동으로 commit 된다.

dbmSelectArray

기능

다수의 레코드를 조회할 때 사용한다. Array 방식은 성능상의 이점은 없으나, 원격 노드에서 처리하는 경우 통신 횟수를 줄이기 위한 목적으로 사용할 수 있다.
사용자가 입력한 데이터를 모두 처리한 후 오류가 발생하면 함수는 "1"을 반환하며, 개별 에러 코드는 사용자 에러 코드 변수에 저장된다.

인자

```
int dbmSelectArray( dbmHandle          * aHandle,
                    const char         * aTableName,
                    void               * aDataPtr,
                    int                  aDataCount,
                    int                * aRowCount,
                    int                  aRetArr[] )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aDataPtr	void *	In/ out	배열 데이터의 시작 주소를 가리키는 포인터이다.
aDataCount	int	In	aDataPtr 변수에 저장된 데이터 개수를 지정한다.
aRowCount	int	Out	전체 처리 건수를 반환한다.
aRetArr	int	Out	각 operation의 에러 코드를 순서대로 저장한다.

노트

- aDataPtr은 (Record 크기 X aDataCount) bytes와 같은 크기여야 한다.
- Record 크기는 table에 저장 가능한 레코드의 크기를 의미하며, **desc** 정보를 통해 확인할 수 있다.
- aRetArr 은 에러가 발생한 array index 위치에 에러 코드만 설정하여 반환한다.

사용 예

```
typedef struct
{
    int c1;
```

```
    int c2;
    int c3;
} DATA;
int main( int argc, char *argv[] )
{
    dbmHandle    * sHandle = NULL;
    DATA           sData[10];
    int sErrCode[10], sRowCount;
    int i;
    int rc;
    rc = dbmInitHandle( &sHandle, "demo" );
    if( rc ) exit(-1);
    for( i = 0; i < 10; i ++ )
    {
        sData[i].c1 = i;
        sData[i].c2 = i;
        sData[i].c3 = i;
    }
    rc = dbmInsertArray( sHandle, "t1", sData, sizeof(DATA), 10, sErrCode );
    if( rc ) exit(-1);
    rc = dbmCommit( sHandle );
    if( rc ) exit(-1);
    for( i = 0; i < 10; i ++ )
    {
        sData[i].c1 = i;
    }
    rc = dbmSelectArray( sHandle, "t1", sData, 10, &sRowCount, sErrCode );
    if( rc ) exit(-1);
    printf( "AffectedRow = %d\n", sRowCount );
    dbmFreeHandle( &sHandle );
    return 0;
}
```

주의

사용자가 입력한 정보 (버퍼 크기, 개수)가 올바르지 않을 경우 잘못된 포인터 접근으로 인해 오동작이 발생할 수 있으므로 주의해야 한다.

dbmDeleteArray

기능

다수의 레코드를 삭제할 때 사용한다. Array 방식은 성능상의 장점은 없으나, 원격 노드에서 처리하는 경우 통신 횟수를 줄이기 위한 목적으로 사용할 수 있다.
사용자가 입력한 데이터를 모두 처리한 후 오류가 발생하면 함수는 "1"을 반환하며, 개별 에러 코드는 사용자 에러 코드 변수에 저장된다.

인자

```
int dbmDeleteArray( dbmHandle          * aHandle,
                    const char         * aTableName,
                    void               * aDataPtr,
                    int                  aDataCount,
                    int                * aRowCount,
                    int                  aRetArr[] )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aDataPtr	void *	In/ out	배열 데이터의 시작 주소를 가리키는 포인터이다.
aDataCount	int	In	aDataPtr 변수에 저장된 데이터 개수를 지정한다.
aRowCount	int	Out	전체 처리 건수를 반환한다.
aRetArr	int	Out	각 operation의 에러 코드를 순서대로 저장한다.

노트

- aDataPtr은 (Record 크기 X aDataCount) bytes와 같은 크기여야 한다.
- Record 크기는 table에 저장 가능한 레코드의 크기를 의미하며, **desc**를 통해 확인할 수 있다.
- aRetArr 은 에러가 발생한 array index 위치에 에러 코드만 설정하여 반환한다.

사용 예

```
typedef struct
{
    int c1;
```

```
    int c2;
    int c3;
} DATA;
int main( int argc, char *argv[] )
{
    dbmHandle    * sHandle = NULL;
    DATA           sData[10];
    int sErrCode[10], sRowCount;
    int i;
    int rc;
    rc = dbmInitHandle( &sHandle, "demo" );
    if( rc ) exit(-1);
    for( i = 0; i < 10; i ++ )
    {
        sData[i].c1 = i;
        sData[i].c2 = i;
        sData[i].c3 = i;
    }
    rc = dbmInsertArray( sHandle, "t1", sData, sizeof(DATA), 10, sErrCode );
    if( rc ) exit(-1);
    rc = dbmCommit( sHandle );
    if( rc ) exit(-1);
    for( i = 0; i < 10; i ++ )
    {
        sData[i].c1 = i;
    }
    rc = dbmDeleteArray( sHandle, "t1", sData, 10, &sRowCount, sErrCode );
    if( rc ) exit(-1);
    printf( "AffectedRow = %d\n", sRowCount );
    rc = dbmCommit( sHandle );
    if( rc ) exit(-1);
    dbmFreeHandle( &sHandle );
    return 0;
}
```

주의

- 사용자가 입력한 정보 (버퍼 크기, 개수)가 올바르지 않을 경우, 잘못된 포인터 접근으로 인해 오동작이 발생할 수 있으므로 주의해야 한다.
- Auto commit mode에서는 처리에 성공한 데이터가 자동으로 commit 된다.

dbmEnqueue

기능

사용자 데이터를 queue 형식의 테이블에 삽입한다.

인자

```
int dbmEnqueue( dbmHandle          * aHandle,
                const char         * aTableName,
                int                  aPriority,
                void               * aUserData,
                int                  aDataSize )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 queue 테이블 이름이다.
aPriority	int	In	0 이상의 사용자 정의 priority 값이다.
aUserData	void *	In	사용자 변수 포인터이다.
aDataSize	int	In	aUserData의 크기이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    rc = dbmInitHandle( &sHandle, "demo" );

    sData.c1 = 10;
    sData.c2 = 100;
    rc = dbmEnqueue( sHandle,
                     "que1",
                     1,
```

```
                    & sData,
                    sizeof(DATA) );

}
```

노트

- Commit이 완료된 후에 dequeue 할 수 있다.
- Priority 값이 낮을수록 우선 순위가 높다.

dbmDequeue

기능

Queue 형식의 테이블에서 사용자 데이터 1 건을 추출한다.

인자

```
int dbmDequeue( dbmHandle          * aHandle,
                const char         * aTableName,
                int                  aInPriority,
                int                * aOutPriority,
                void               * aUserData,
                int                * aDataSize,
                int                  aTimeoutMicroSecond )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 queue 테이블의 이름이다.
aInPriority	int	In	특정 priority에 대해 dequeue를 수행하려는 경우 해당 값을 입력한다. 제한 없이 조회하려는 경우에는 -1을 입력한다. 입력한 priority 숫자 이상의 데이터를 반환한다.
aOutPriority	int *	Out	해당 메시지의 priority 정보를 반환한다. (NULL 인 경우에는 반환하지 않는다.)
aUserData	void *	Out	사용자 변수 포인터이다.
aDataSize	int *	Out	aUserData의 크기이다.
aTimeoutMicroSecond	int	In	Queue에 데이터가 없을 경우 대기하는 시간을 us 단위로 지정한다.

aTimeoutMicroSecond의 설정 값에 따른 동작 방식은 다음과 같다.

aTimeout 설정값	동작 방식
-1	Queue에 데이터가 없을 경우 NOT_FOUND 에러를 반환한다.
0	Queue에 데이터가 없을 경우 무한 대기한다.
0 보다 큰 값	aTimeout 시간 동안 queue에 데이터가 없을 경우 TIMEOUT 에러를 반환한다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sDataSize = 0;
    int            sPriority;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmDequeue( sHandle,
                     "que1",
                     -1,
                     &sPriority,
                     & sData,
                     & sDataSize,
                     1000 );

}
```

노트

- MsgID는 dbmEnqueue 수행 시점에 부여되는 internal sequence 값이다.
- Queue에 저장된 데이터는 (Priority, MsgID) 조합 기준으로 정렬된다.
- Priority를 지정하지 않은 경우에는 MsgID 순서대로 출력된다.
- Priority 값이 동일한 경우에는 MsgID 순서대로 출력된다.
- Priority 값을 지정한 경우에는 해당 값 이상인 항목만 출력된다.

dbmGetStore

기능

Store 테이블에서 특정 key에 해당하는 value를 조회한다.

인자

```
dbmInt32 dbmGetStore( dbmHandle       * aHandle,
                      dbmTableHandle  * aTableHandle,
                      const char      * aKey,
                      char            * aValue )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 tableHandle 이다.
aKey	const char *	In	조회할 key 값이다.
aValue	char *	Out	aKey에 해당하는 value가 저장되는 포인터 변수이다.

사용 예

```
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    char             sKey[512];
    char             sValue[512];
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareTableHandle( sHandle, "S1", &sTableHandle );
    rc = dbmGetStore( sHandle,
                      sTableHandle,
                      sKey,
                      sValue );
    dbmFreeHandle( &sHandle );
}
```

노트

- Key와 value로 사용하는 사용자 변수는 NULL 일 수 없다.
- Key는 string 형식이어야 한다.

dbmSetStore

기능

Store 테이블에 특정 key와 value를 저장한다.

인자

```
dbmInt32 dbmSetStore( dbmHandle       * aHandle,
                      dbmTableHandle  * aTableHandle,
                      const char      * aKey,
                      const char      * aValue,
                      int               aCheckDup )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 tableHandle 이다.
aKey	const char *	In	저장할 데이터의 key 이다.
aValue	const char *	In/Out	저장할 데이터의 value 이다.
aCheckDup	int	In	중복 키 여부를 체크하는 변수이다. • 1: 중복 키가 존재할 경우 오류를 반환한다. • 0: 중복 키가 존재할 경우 기존 데이터를 update 한다.

사용 예

```
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    char             sKey[512];
    char             sValue[512];
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareTableHandle( sHandle, "S1", &sTableHandle );
    rc = dbmSetStore( sHandle,
                      sTableHandle,
                      sKey,
                      sValue,
                      0 );
    rc = dbmCommit( sHandle );
```

```
    rc = dbmGetStore( sHandle,
                      sTableHandle,
                      sKey,
                      sValue );
    dbmFreeHandle( &sHandle );
}
```

노트

- Key와 value로 사용하는 사용자 변수는 NULL 일 수 없다.
- Key는 string 형식이어야 한다.

dbmDelStore

기능

Store 테이블에서 특정 key에 해당하는 사용자 데이터를 삭제한다.

인자

```
dbmInt32 dbmDelStore( dbmHandle       * aHandle,
                      dbmTableHandle  * aTableHandle,
                      const dbmChar   * aKey )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 tableHandle 이다.
aKey	const char *	In	조회할 key 값이다.

사용 예

```
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    char             sKey[512];
    char             sValue[512];
    int              i;
    int              rc;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareTableHandle( sHandle, "S1", &sTableHandle );
    for( i = 0; i < 10; i++ )
    {
        sprintf( sKey, "k%d", i );
        rc = dbmDelStore( sHandle, sTableHandle, sKey );
        rc = dbmCommit( sHandle );
    }
    dbmFreeHandle( &sHandle );
}
```

노트

- Key와 value로 사용하는 사용자 변수는 NULL 일 수 없다.
- Key는 string 형식이어야 한다.

dbmGetCurrVal

기능

Sequence 객체의 현재값을 반환한다.

인자

```
int dbmGetCurrVal( dbmHandle          * aHandle,
                   const char         * aTableName,
                   long long          * aCurrVal )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 sequence 이름이다.
aCurrVal	long long int	Out	현재값을 반환받을 변수의 포인터이다. (8 byte 변수 필요)

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    long long      sCurrVal;

    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmGetNextVal( sHandle,
                        "seq1",
                        & sNextVal);
    rc = dbmGetCurrVal( sHandle,
                        "seq1",
                        & sCurrVal );
}
```

주의

Sequence 객체에 대해 nextval이 호출되지 않은 상태에서 currval을 호출할 경우 오류가 발생한다.

dbmGetNextVal

기능

sequence 객체의 다음 값을 반환한다.

인자

```
int dbmGetNextVal( dbmHandle          * aHandle,
                   const char         * aTableName,
                   long long          * aNextVal )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 sequence 이름이다.
aNextVal	long long int	Out	반환받을 변수의 포인터 (8 byte 변수 필요) 이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    long long      sNextVal;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmGetNextVal( sHandle,
                        "seq1",
                        & sNextVal);
}
```

dbmCommit

기능

사용자가 수행한 트랜잭션의 변경 사항을 영구적으로 반영한다.

인자

```
int dbmCommit( dbmHandle           * aHandle )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sRowCount = 0;
    rc = dbmInitHandle( &sHandle, "demo" );
    sData.c1 = 10;
    sData.c2 = 100;
    rc = dbmInsertRow( sHandle, "t1", &sData, sizeof(DATA) );
    sData.c1 = 10;
    sData.c2 = 200;
    rc = dbmUpdateRow( sHandle, "t1", &sData, &sRowCount );
    rc = dbmCommit( sHandle );
}
```

dbmRollback

기능

사용자가 수행한 트랜잭션을 rollback 한다.

인자

```
int dbmRollback( dbmHandle           * aHandle )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sRowCount = 0;
    rc = dbmInitHandle( &sHandle, "demo" );
    sData.c1 = 10;
    sData.c2 = 100;
    rc = dbmInsertRow( sHandle, "t1", &sData, sizeof(DATA) );
    sData.c1 = 10;
    sData.c2 = 200;
    rc = dbmUpdateRow( sHandle, "t1", &sData, &sRowCount );
    rc = dbmRollback( sHandle );
}
```

dbmDeferCommit

기능

디스크 모드에서만 호출할 수 있으며, 사용자가 수행한 트랜잭션의 변경 사항을 메모리에만 반영한다.
dbmDeferCommit 호출 이후에는 반드시 dbmDeferSync를 호출해야 트랜잭션이 정상적으로 완료된다.

인자

```
int dbmDeferCommit( dbmHandle           * aHandle )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sRowCount = 0;
    rc = dbmInitHandle( &sHandle, "demo" );
    sData.c1 = 10;
    sData.c2 = 100;
    rc = dbmInsertRow( sHandle, "t1", &sData, sizeof(DATA) );
    sData.c1 = 10;
    sData.c2 = 200;
    rc = dbmUpdateRow( sHandle, "t1", &sData, &sRowCount );
    rc = dbmDeferCommit( sHandle );
}
```

주의

- dbmDeferCommit 수행 후에는 해당 세션이 점유하고 있던 모든 lock이 해제된다.
- dbmDeferCommit 수행 이후에는 dbmRollback을 호출할 수 없다.
- dbmDeferCommit 수행 후부터 dbmDeferSync 호출 전까지는 새로운 트랜잭션을 수행할 수 없다.
- 프로세스가 비정상 종료되더라도 delayed recovery는 가능하다. 그러나 dbmDeferSync가 호출되지 않은 상태에서 OS fatal 오류 또는 H/W 장애가 발생하면 해당 트랜잭션은 유실될 수 있다.

dbmDeferSync

기능

dbmDeferCommit 수행으로 메모리에만 commit 되어 있던 트랜잭션 내역을 디스크에 기록한다.

인자

```
int dbmDeferSync( dbmHandle           * aHandle )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sRowCount = 0;
    rc = dbmInitHandle( &sHandle, "demo" );
    sData.c1 = 10;
    sData.c2 = 100;
    rc = dbmInsertRow( sHandle, "t1", &sData, sizeof(DATA) );
    sData.c1 = 10;
    sData.c2 = 200;
    rc = dbmUpdateRow( sHandle, "t1", &sData, &sRowCount );
    rc = dbmDeferCommit( sHandle );
    rc = dbmDeferSync( sHandle );
}
```

주의

- dbmDeferCommit 이후에만 사용할 수 있다.
- dbmDeferCommit 이후 dbmDeferSync가 호출되지 않은 상태에서는 새로운 트랜잭션을 시작할 수 없다.

dbmRefineSystem

기능

지정한 instance 내의 table/ index lock을 해제하거나 index를 재구축하는 복구 기능을 수행한다.
자세한 내용은 **alter system refine [TableList]**을 참조한다.

인자

```
int dbmRefineSystem( const char    * aInstName,
                     const char    * aTableName )
```

인자 항목	타입	In/ out	비고
aInstName	const char *	In	instance 이름을 입력한다.
aTableName	const char *	In	특정 table name을 입력한다.

- Instance 이름은 필수로 지정해야 한다.
- TableName을 지정하면 해당 테이블만 복구하고, NULL을 입력하면 instance와 관련된 전체 테이블 중에 문제 상태인 테이블을 탐색하여 복구를 수행한다.

사용 예

```
if( argc == 0 )
{
    if( dbmRefineSystem( "demo", NULL ) != 0 )   // Instance내 모든 테이블 대상
    {
        printf( "failed to refine all\n" );
        exit(-1);
    }
}
else
{
    if( dbmRefineSystem( "demo", argv[1] ) != 0 )  // 특정 테이블만 복구할 경우
    {
        printf( "failed to refine t1\n" );
        exit(-1);
    }
}
```

노트

dbmRefineSystem은 index segment를 재구축 (Drop → Create) 하는 방식으로 동작하며, 이 과정에서 이미 실행 중인 application 과의 동시성 제어는 보장하지 않는다.

dbmGetRowCount

기능

dbmExecuteStmt 수행 후 대상 레코드의 개수를 반환한다.

인자

```
int dbmGetRowCount( dbmHandle          * aHandle,
                    dbmStmt            * aStmt,
                    int                * aRowCount )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	dbmExecuteStmt으로 실행된 dbmStmt 변수이다.
aRowCount	int *	Out	반환받을 변수 포인터 (4 byte 크기의 변수) 이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmStmt      * sStmt = NULL;
    int            sRowCount;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareStmt( sHandle, "select * from t1", &sStmt );
    rc = dbmExecuteStmt( sHandle, sStmt );

    rc = dbmGetRowCount( sHandle, sStmt, &sRowCount );
}
```

dbmGetRowSize

기능

입력한 테이블의 레코드 크기 정보를 반환한다.

인자

```
int dbmGetRowSize( dbmHandle          * aHandle,
                   const char         * aTableName,
                   int                * aUserData )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	조회할 object name 이다.
aUserData	int *	Out	반환받을 변수 포인터 (4 byte 크기의 변수) 이다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    int            sRowSize;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmGetRowSize( sHandle, "Table1", &sRowSize );
}
```

dbmGetTableName

기능

테이블 핸들로부터 테이블 이름을 반환한다.

인자

```
const char * dbmGetTableName( dbmTableHandle          * aTableHandle )
```

인자 항목	타입	In/ out	비고
aTableHandle	dbmTableHandle *	In	dbmPrepareTableHandle 로 처리된 변수이다.

사용 예

```
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    const char     * sTableName;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareTableHandle( &sHandle, "t1", &sTableHandle );
    sTableName = dbmGetTableName ( sTableHandle );
}
```

노트

dbmPrepareTableHandle이 성공적으로 수행된 상태여야 한다.

dbmGetTableType

기능

테이블 핸들로부터 테이블 유형 값을 반환한다.

인자

```
dbmTableType  dbmGetTableType( dbmTableHandle          * aTableHandle )
```

인자 항목	타입	In/ out	비고
aTableHandle	dbmTableHandle *	In	dbmPrepareTableHandle 로 처리된 변수이다.

dbmTableType의 정의는 아래와 같다.

```
typedef enum
{
    DBM_TABLE_TYPE_INVALID = 0,
    DBM_TABLE_TYPE_TABLE,                // BTree Index Table
    DBM_TABLE_TYPE_QUEUE,                // Queue table
    DBM_TABLE_TYPE_STORE,                // Store table
    DBM_TABLE_TYPE_SEQUENCE,             // Sequence Object
    DBM_TABLE_TYPE_DIRECT_TABLE,         // Direct Table
    DBM_TABLE_TYPE_DIRECT_QUEUE,         // deprecated
    DBM_TABLE_TYPE_SPLAY_TABLE,          // Splay Index Table
    DBM_TABLE_TYPE_LIST_TABLE,           // deprecated
    DBM_TABLE_TYPE_PERF_VIEW,            // Performance view
    DBM_TABLE_TYPE_USER_TYPE,            // User-defined type
    DBM_TABLE_TYPE_MAX
} dbmTableType;
```

노트

사용자가 생성할 수 있는 유형은 다음과 같다.

- DBM_TABLE_TYPE_TABLE
- DBM_TABLE_TYPE_STORE
- DBM_TABLE_TYPE_QUEUE
- DBM_TABLE_TYPE_SEQUENCE

- DBM_TABLE_TYPE_DIRECT_TABLE
- DBM_TABLE_TYPE_SPLAY_TABLE

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    dbmTableType  sTableType;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareTableHandle( &sHandle, "t1", &sTableHandle );
    sTableType = dbmGetTableType ( sTableHandle );
}
```

dbmSetSplayMode4DML

기능

Splay table 삽입 연산 시 splay tree (마지막으로 접근한 데이터를 root로 지정하는 동작) 를 수행할지 여부를 설정한다. 기본값은 disable 이다.

인자

```
int dbmSetSplayMode4DML( dbmTableHandle          * aTableHandle,
                         int                       aMode )
```

인자 항목	타입	In/ out	비고
aTableHandle	dbmTableHandle *	In	dbmPrepareTableHandle 로 처리된 변수이다.
aMode	int	In	0: No splay (default) 1: Key 삽입 시 splay 동작 수행

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareTableHandle( &sHandle, "t1", &sTableHandle );
    dbmSetSplayMode4DML( sTableHandle, 1 );
}
```

dbmGetErrorData

기능

사용자 핸들에는 API 처리 과정에서 발생한 오류가 순차적으로 최대 4 개까지 적재된다.
해당 핸들에 저장된 에러를 확인할 때 사용한다.

인자

```
int dbmGetErrorData( dbmHandle          * aHandle,
                     int                * aErrorCode,
                     char               * aErrorMsg,
                     int                  aErrorMsgSize )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aErrorCode	int *	Out	에러 코드가 담길 변수 포인터이다.
aErrorMsg	char *	Out	에러 메시지가 저장될 변수 포인터이다.
aErrorMsgSize	int	In	에러 메시지를 반환받을 버퍼의 크기이다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sErrCode;
    char           sErrMsg[1024];
    rc = dbmInitHandle( &sHandle, "demo" );
    sData.c1 = 10;
    sData.c2 = 100;
    rc = dbmInsertRow( sHandle, "t1", &sData, sizeof(DATA) );
    if( rc )
    {
```

```
        while( dbmGetErrorData( sHandle, &sErrCode, sErrMsg, 1024 ) == 0 )
        {
             fprintf( stdout, "ERR-%d] %s\n", sErrCode, sErrMsg );
        }
    }
}
```

노트

에러 메시지를 저장할 변수의 크기는 최소 512 byte 이상이어야 한다.

dbmGetErrorMsg

기능

API 호출 과정에서 발생한 각 오류 코드에 대한 상세 에러 메시지를 확인한다.

인자

```
void dbmGetErrorMsg( int         aErrorCode,
                     char      * aErrorMsg,
                     int         aErrorMsgSize )
```

인자 항목	타입	In/ out	비고
aErrorCode	int	In	조회할 에러 코드이다.
aErrorMsg	char *	Out	에러 코드가 담길 변수 포인터이다.
aErrorMsgSize	int	In	에러 메시지가 저장될 변수의 크기를 지정한다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sErrCode;
    char           sErrMsg[1024];
    rc = dbmInitHandle( &sHandle, "demo" );
    sData.c1 = 10;
    sData.c2 = 100;
    rc = dbmInsertRow( sHandle, "t1", &sData, sizeof(DATA) );
    if( rc )
    {
        dbmGetErrorMsg( rc, sErrMsg, sizeof(sErrMsg) );
    }
}
```

노트

에러 메시지를 저장할 변수의 크기는 최소 512 byte 이상이어야 한다.

dbmGetTableUsage

기능

지정한 테이블의 사용량 정보를 반환한다.

인자

```
int dbmGetTableUsage( dbmHandle       * aHandle,
                      const char      * aTableName,
                      long            * aMaxSize,
                      long            * aTotalSize,
                      long            * aUsedSize,
                      long            * aFreeSize )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 테이블 이름이다.
aMaxSize	long *	Out	테이블을 생성할 때 지정한 최대 row 개수이다.
aTotalSize	long *	Out	테이블 확장 상태를 포함하여 현재 기준 최대 저장 가능 row 개수이다.
aUsedSize	long *	Out	현재 테이블에 사용 중인 row 개수이다. (Commit 되지 않은 row를 포함할 수 있음)
aFreeSize	long *	Out	테이블의 가용 공간 개수이다. (Commit 되지 않은 row를 포함할 수 있음)

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    long           sTotal;
    long           sMax;
    long           sUsed;
    long           sFree;
    int            rc;
    rc = dbmInitHandle( &sHandle,
                        "demo" );
    rc = dbmGetTableUsage( sHandle,
                           "t1",
                           &sMax,
```

```
                        &sTotal,
                        &sUsed,
                        &sFree );
}
```

노트

Slot 관리를 별도로 수행하는 테이블 (Normal, Queue, Store, Splay) 에 대해서만 유효한 정보를 반환한다.

dbmGetTableUsageByHandle

기능

테이블 핸들에 해당하는 테이블의 사용량 정보를 반환한다.

인자

```
int dbmGetTableUsageByHandle( dbmHandle       * aHandle,
                              dbmTableHandle  * aTableHandle,
                              long            * aMaxSize,
                              long            * aTotalSize,
                              long            * aUsedSize,
                              long            * aFreeSize );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 테이블 핸들이다.
aMaxSize	long *	Out	테이블을 생성할 때 지정한 최대 row 개수이다.
aTotalSize	long *	Out	테이블 확장 상태를 포함하여 현재 기준 최대 저장 가능 row 개수이다.
aUsedSize	long *	Out	현재 테이블에 사용 중인 row 개수이다. (Commit 되지 않은 row를 포함할 수 있음)
aFreeSize	long *	Out	테이블의 가용 공간 개수이다. (Commit 되지 않은 row를 포함할 수 있음)

사용 예

```
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    long             sTotal;
    long             sMax;
    long             sUsed;
    long             sFree;
    int              rc;
    rc = dbmInitHandle( &sHandle,
                        "demo" );
    rc = dbmPrepareTableHandle( sHandle,
```

```
                                "t1",
                                & sTableHandle );
    rc = dbmPrepareTableHandle( sHandle, "t1", &sTableHandle );
    rc = dbmGetTableUsageByHandle( sHandle,
                                   sTableHandle,
                                   &sMax,
                                   &sTotal,
                                   &sUsed,
                                   &sFree );
}
```

노트

Slot 관리를 별도로 수행하는 테이블 (Normal, Queue, Store, Splay) 에 대해서만 유효한 정보를 반환한다.

dbmExtendTable

기능

테이블 핸들로 지정된 테이블에 segment를 추가한다.
추가되는 segment의 크기는 테이블 생성 시 설정된 extend 크기와 같다.

인자

```
int dbmExtendTable( dbmHandle       * aHandle,
                    dbmTableHandle  * aTableHandle );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 테이블의 핸들이다.

사용 예

```
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    rc = dbmInitHandle( &sHandle,
                        "demo" );
    rc = dbmPrepareTableHandle( sHandle,
                                "t1",
                                & sTableHandle );
    rc = dbmExtendTable( sHandle,
                         sTableHandle );
}
```

노트

- Extend에 의해 확장 가능한 segment의 최대 개수는 999 개이다.
- Extend가 빈번하게 발생할 경우 삽입 성능이 저하될 수 있으므로, table 생성 시 init 크기를 적절하게 설정해야 한다.

dbmExistDataInQue

기능

Queue table에 데이터가 존재하는지 여부를 반환한다. 단, 이 값은 다음 dbmDequeue 호출에서 데이터가 반드시 반환됨을 보장하지 않으므로, 힌트 정보로만 활용해야 한다.

인자

```
int dbmExistDataInQue( dbmHandle       * aHandle,
                       const char      * aTableName,
                       int             * aExists );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	const char *	In	대상 queue table 이름이다.
aExists	int *	Out	• 0: 존재하지 않는다. • 1: 한 개 이상의 데이터가 존재한다.

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    int            i;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmExistDataInQue( sHandle, "que1", &i );
}
```

노트

Queue type 테이블만 지원한다.

dbmSetAutoCommit

기능

변경 연산을 자동으로 커밋할지 여부를 지정한다.

인자

```
int dbmSetAutoCommit( dbmHandle       * aHandle,
                      int               aMode )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aMode	int	In	• 0: Non auto commit mode • 1: Auto commit mode

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmSetAutoCommit( sHandle, 1 );
}
```

주의

- Auto commit mode에서는 dbmSelectForUpdateRow 와 같은 API가 lock을 점유하지 않는다.
- 진행 중인 트랜잭션이 있는 경우 dbmCommit 또는 dbmRollback으로 먼저 종료한 후 수행해야 한다.

dbmSetLoggingMode

기능

In-memory logging 모드를 지정한다. 변경/삭제 연산 시 레코드 이미지를 undo 영역에 복제한 후 변경을 수행한다. 만일 사용자가 logging이 필요하지 않다고 판단하는 경우, API를 통해 logging mode를 disable 로 설정할 수 있다.

인자

```
int dbmSetLoggingMode( dbmHandle       * aHandle,
                       int               aMode )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aMode	int	In	• 0: Logging을 수행하지 않음 • 1: Logging을 수행함

사용 예

```
main()
{
    dbmHandle    * sHandle = NULL;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmSetAutoCommit( sHandle, 1 );
}
```

주의

In-memory logging을 disable 할 경우 주의사항은 다음과 같다.

- 트랜잭션 단위의 commit/ rollback 이 지원되지 않으며, auto commit 으로 동작한다.
- Delayed recovery 기능은 지원되지 않는다.

2.3 JAVA

Java 환경에서 GOLDILOCKS LITE 에 접근할 수 있도록 type-2 형태의 JDBC driver를 제공한다.
Type-2 방식은 Java 22에서 지원하는 FFI (Foreign Function Interface)를 사용하여 libdbmCore.so에 구현된 C API를 호출하는 방식이다.

노트

- Java 22 이상 버전에서 사용해야 한다.
- JDBC spec 중 필수적인 최소 함수만 제공한다.
- Lite 환경에서는 Java의 byte 타입을 별도로 지원하지 않으며 CHAR column에는 string 또는 byte[] 형태로 데이터를 저장한다. 조회 시에는 상황에 따라 getString 또는 getBytes 를 사용해야 한다.
- DATE column은 java.sql.TIMESTAMP 타입으로 처리되며, 값은 마이크로초 단위로 저장된다.

아래 표에서는 JAR 파일, driver class, URL 정보에 대해 설명한다.

항목	설명
Jar	${DBM_HOME}/lib/dbmJdbc.jar
Driver Class	com.dbm.jdbc.dbmDriver
URL	jdbc:dbm://<IP>:<Port>/demo • TCP 방식으로 접속하려면 dbmListener가 실행 중이어야 한다. • IP와 PORT를 생략할 경우 DA 방식으로 접속한다.

예제 수행을 위해 먼저 다음과 같이 테이블을 생성한다.

```
create table T1
(
    C1                          int,
    C2                          short,
    C3                          long,
    C4                          double,
    C5                          float,
    C6                          char(100),
    C7                          date
) init 1024 extend 102400 max 4096000;
create unique index IDX_T1 on T1 (C1 asc);
```

다음은 GOLDILOCKS LITE JDBC를 이용한 예제이다.

```
import java.sql.*;
import java.util.*;
import java.util.concurrent.*;
public class LiteJdbcTest {
    static final int TOTAL_COUNT = 1_000_000;
    static final String VALUE_1200B;
    static {
        char[] buf = new char[1200];
        Arrays.fill(buf, 'X');
        VALUE_1200B = new String(buf);
    }
    static class Worker implements Runnable {
        int start, end;
        String phase;
        long[] phaseTimes;
        int index;
        String jdbcUrl;
        Worker(int start, int end, String phase, long[] phaseTimes, int index, String jdbcUrl)
{
            this.start = start;
            this.end = end;
            this.phase = phase;
            this.phaseTimes = phaseTimes;
            this.index = index;
            this.jdbcUrl = jdbcUrl;
        }
        @Override
        public void run() {
            long t0 = System.nanoTime();
            try (Connection conn = DriverManager.getConnection(jdbcUrl)) {
                conn.setAutoCommit(true);
                switch (phase) {
                    case "INSERT":
                        try (PreparedStatement pstmt = conn.prepareStatement(
                                "INSERT INTO t_test(id, payload) VALUES(?, ?)")) {
                            for (int i = start; i <= end; i++) {
                                pstmt.setInt(1, i);
                                pstmt.setString(2, VALUE_1200B);
                                pstmt.execute();
                                //conn.commit();
                            }
```

```
                    }
                    break;
                case "UPDATE":
                    try (PreparedStatement pstmt = conn.prepareStatement(
                            "UPDATE t_test SET payload=? WHERE id=?")) {
                        for (int i = start; i <= end; i++) {
                            pstmt.setString(1, VALUE_1200B);
                            pstmt.setInt(2, i);
                            pstmt.execute();
                            //conn.commit();
                        }
                    }
                    break;
                case "SELECT":
                    try (PreparedStatement pstmt = conn.prepareStatement(
                            "SELECT payload FROM t_test WHERE id=?")) {
                        for (int i = start; i <= end; i++) {
                            pstmt.setInt(1, i);
                            try (ResultSet rs = pstmt.executeQuery()) {
                                if (rs.next()) {
                                    rs.getString(1);
                                }
                            }
                        }
                    }
                    break;
                case "DELETE":
                    try (PreparedStatement pstmt = conn.prepareStatement(
                            "DELETE FROM t_test WHERE id=?")) {
                        for (int i = start; i <= end; i++) {
                            pstmt.setInt(1, i);
                            pstmt.execute();
                            //conn.commit();
                        }
                    }
                    break;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        long t1 = System.nanoTime();
```

```
            phaseTimes[index] = t1 - t0;
        }
    }
    private static void runPhase(String phase, int threadCount, String jdbcUrl) throws
InterruptedException {
        System.out.println("===== " + phase + " Phase 시작 =====");
        int perThread = TOTAL_COUNT / threadCount;
        long[] phaseTimes = new long[threadCount];
        Thread[] threads = new Thread[threadCount];
        for (int i = 0; i < threadCount; i++) {
            int start = i * perThread + 1;
            int end = (i == threadCount - 1) ? TOTAL_COUNT : (i + 1) * perThread;
            threads[i] = new Thread(new Worker(start, end, phase, phaseTimes, i, jdbcUrl), "
worker-" + i);
            threads[i].start();
        }
        for (Thread t : threads) t.join();
        for (int i = 0; i < threadCount; i++) {
            System.out.printf("Thread-%d : %.3f sec\n", i, phaseTimes[i] / 1_000_000_000.0);
        }
        long maxNs = 0;
        for (long t : phaseTimes) maxNs = Math.max(maxNs, t);
        double secTotal = maxNs / 1_000_000_000.0;
        double tps = TOTAL_COUNT / secTotal;
        System.out.printf("Phase Total (max of threads): %.3f sec, TPS=%.2f\n", secTotal, tps
);
        System.out.println("===== " + phase + " Phase 완료 =====\n");
    }
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.out.println("Usage: java LiteJdbcTestMultiConn <numThreads>");
            return;
        }
        int threadCount = Integer.parseInt(args[0]);
        System.out.println("NumThreads: " + threadCount);
        // SQLite in-memory DB URL
        //String jdbcUrl = "jdbc:dbm://127.0.0.1:27584/demo";
        String jdbcUrl = "jdbc:dbm:///demo";
        Class.forName("com.dbm.jdbc.dbmDriver");
        // 초기 테이블 생성
        try (Connection conn = DriverManager.getConnection(jdbcUrl)) {
```

```
            conn.setAutoCommit(false);
            try (Statement stmt = conn.createStatement()) {
                stmt.execute("DROP TABLE t_test");
            } catch(Exception e) {}
        }
        try (Connection conn = DriverManager.getConnection(jdbcUrl)) {
            try (Statement stmt = conn.createStatement()) {
                stmt.execute("CREATE TABLE t_test(id int , payload CHAR(1200))");
                stmt.execute("CREATE unique index idx1 on t_test(id)");
            }
            conn.commit();
        }
        runPhase("INSERT", threadCount, jdbcUrl);
        runPhase("UPDATE", threadCount, jdbcUrl);
        runPhase("SELECT", threadCount, jdbcUrl);
        runPhase("DELETE", threadCount, jdbcUrl);
    }
}
```

다음은 예제를 컴파일하고 실행하는 명령어 예이다.

```
shell> javac -cp ${DBM_HOME}/lib/dbmJdbc.jar:. LiteJdbcTest.java
shell> java -cp ${DBM_HOME}/lib/dbmJdbc.jar:. LiteJdbcTest
```

Class & Method

GOLDILOCKS LITE에서 제공하는 JDBC 구현 class 와 method를 설명한다.
여기에 기술되지 않은 method는 모두 SQLFeatureNotSupportedException 을 반환한다.

> 노트
> 일부 구현된 method는 연동 호환성을 위해 의미 없는 고정값을 반환할 수 있다.
> (자세한 내용은 각 method 별 상세 설명을 참조한다.)

Class	설명
dbmConnection	java.sql.Connection의 implementation
dbmStatement	java.sql.Statement의 implementation

Class	설명
dbmPreparedStatement	java.sql.PreparedStatement의 implementation
dbmResultSet	java.sql.ResultSet의 implementation
dbmResultSetMeta	java.sql.ResultSetMeta의 implementation

dbmConnection

Method	관련 C API	비고
PreparedStatement prepareStatement(String sql)	dbmPrepareStmt	입력된 SQL을 실행할 수 있도록 준비한다.
Statement createStatement()	해당 사항 없음	-
Statement createStatement(int resultSetType, int resultSetConcurrency)	해당 사항 없음	TYPE_FORWARD_ONLY CONCUR_READ_ONLY CLOSE_CURSOR_AT_COMMIT 속성으로 고정된다. (다른 값을 지정할 경우 오류가 발생한다.)
Statement createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)	해당 사항 없음	
void commit()	dbmCommit	트랜잭션을 commit 한다.
void rollback()	dbmRollback	트랜잭션을 rollback 한다.
void close()	dbmFreeHandle	현재 연결을 종료한다.
boolean isClosed()	해당 사항 없음	연결 종료 여부를 반환한다. (정상 상태에서는 false 반환)
boolean getAutoCommit()	해당 사항 없음	현재 auto commit mode를 반환한다. (Default: false)
void setAutoCommit(boolean autoCommit)	dbmSetAutoCommit	Auto commit mode를 설정한다.

dbmStatement

Method	관련 C API	비고
boolean execute(String sql)	dbmExecuteStmt	SQL 문을 수행한다.
int executeUpdate(String sql)	dbmExecuteStmt	SQL 문을 수행한다.
ResultSet executeQuery(String sql)	dbmExecuteStmt	SQL 문을 수행한다.
void close()	dbmFreeStmt	Statement 를 종료한다.
boolean isClosed()	해당 사항 없음	Statement가 닫힌 상태인지 반환한다. (닫힌 경우 true를 반환한다.)
Connection getConnection()	해당 사항 없음	Statement를 생성한 connection 객체를 반환한다.
getUpdateCount	dbmGetRowCount	INSERT/ UPDATE/ DELETE에 의해 영향을 받은 레코드 개수를 반환한다. (SELECT의 경우에는 보장하지 않는다.)

dbmPreparedStatement

Method	관련 C API	비고
int executeUpdate()	dbmExecuteStmt	preparedStatement 를 실행한다. INSERT/ UPDATE/DELETE 수행 시, 영향 받은 레코드 개수를 반환한다.
boolean execute()	dbmExecuteStmt	preparedStatement 를 실행한다.
void close()	-	preparedStatement 객체를 해제한다.
Connection getConnection()	-	preparedStatement를 생성한 connection 객체를 반환한다.
void setShort(int index, short value)	dbmBindParamById	SHORT type column에 데이터를 바인딩 할 때 사용한다.
void setFloat(int index, float value)	dbmBindParamById	FLOAT type column에 데이터를 바인딩 할 때 사용한다.
void setDouble(int index, double value)	dbmBindParamById	DOUBLE type column에 데이터를 바인딩한다.
void setInt(int index, int value)	dbmBindParamById	INT type column에 데이터를 바인딩한다.
void setLong(int index, long value)	dbmBindParamById	LONG type column에 데이터를 바인딩한다.
void setString(int index, String value)	dbmBindParamById	CHAR type column에 데이터를 바인딩한다.
void setTimestamp(int index, java.sql.Timestamp value)	dbmBindParamById	DATE type column에 데이터를 바인딩한다.
void setNull(int index, int sqlType)	해당 사항 없음	지정된 index의 bind parameter 값을 NULL로 설정한다.
void setByte(int index, Byte value)	dbmBindParamById	CHAR type column에 byte 값을 바인딩한다.
void setBytes(int index, byte[] value)	dbmBindParamById	CHAR type column에 byte[] 값을 바인딩한다.
ResultSet executeQuery()	dbmExecuteStmt	preparedStatement 를 실행한다.
ResultSet getResultSet()	해당 사항 없음	마지막으로 수행된 결과를 반환한다.
int getUpdateCount()	dbmGetRowCount	INSERT/ UPDATE/ DELETE 수행 시 영향 받은 레코드 개수를 반환한다.
boolean getMoreResults()	해당 사항 없음	항상 false를 반환한다.

dbmResultSet

Class	관련 C API	비고
int findColumn(String columnLabel)	해당 사항 없음	주어진 columnLabel 에 해당하는 column index를 반환한다.
int getColumnCount()	-	ResultSet의 column 개수를 반환한다.
String getColumnTypeName(int idx)	dbmGetTargetInfo	주어진 columnIndex에 해당하는 column의 Java 데이터 타입 이름을 반환한다.

Class	관련 C API	비고
boolean next()	dbmFetchStmt	ResultSet에서 다음 한 건을 반환한다.
short getShort(int columnIndex)	해당 사항 없음	입력된 columnIndex에 해당하는 column 값을 반환한다.
long getLong(int columnIndex)	해당 사항 없음	입력된 columnIndex에 해당하는 column 값을 반환한다.
float getFloat(int columnIndex)	해당 사항 없음	입력된 columnIndex에 해당하는 column 값을 반환한다.
double getDouble(int columnIndex)	해당 사항 없음	입력된 columnIndex에 해당하는 column 값을 반환한다.
int getInt(int columnIndex)	해당 사항 없음	입력된 columnIndex에 해당하는 column 값을 반환한다.
String getString(int columnIndex)	해당 사항 없음	입력된 columnIndex에 해당하는 column 값을 반환한다.
Timestamp getTimestamp(int columnIndex)	해당 사항 없음	입력된 columnIndex에 해당하는 column 값을 반환한다.
byte getByte(int columnIndex)	해당 사항 없음	입력된 columnIndex에 해당하는 column 값을 반환한다.
byte[] getByte(int columnIndex)	해당 사항 없음	입력된 columnIndex에 해당하는 column 값을 반환한다.
Object getObject(int columnIndex)	해당 사항 없음	입력된 columnIndex에 해당하는 column 값을 반환한다. (CHAR 타입 column은 byte 반환을 지원하지 않으므로 getByte, getBytes 함수를 사용해야 한다.)
void close()	해당 사항 없음	ResultSet을 해제한다.
boolean isClosed()	해당 사항 없음	ResultSet이 닫혔는지 여부를 반환한다.
ResultSetMetaData getMetaData()	해당 사항 없음	ResultSetMeta를 반환한다.
int getType()	해당 사항 없음	ResultSet.TYPE_FORWARD_ONLY만 반환한다.
String getString(String columnLabel)	해당 사항 없음	입력된 columnLabel에 해당하는 column 값을 반환한다.
getShort(String columnLabel)	해당 사항 없음	입력된 columnLabel에 해당하는 column 값을 반환한다.
float getFloat(String columnLabel)	해당 사항 없음	입력된 columnLabel에 해당하는 column 값을 반환한다.
getInt(String columnLabel)	해당 사항 없음	입력된 columnLabel에 해당하는 column 값을 반환한다.
getLong(String columnLabel)	해당 사항 없음	입력된 columnLabel에 해당하는 column 값을 반환한다.
double getDouble(String columnLabel)	해당 사항 없음	입력된 columnLabel에 해당하는 column 값을 반환한다.
Timestamp getTimestamp(String columnLabel)	해당 사항 없음	입력된 columnLabel에 해당하는 column 값을 반환한다.
byte getByte(String columnLabel)	해당 사항 없음	입력된 columnLabel에 해당하는 column 값을 반환한다.
byte[] getBytes(String columnLabel)	해당 사항 없음	입력된 columnLabel에 해당하는 column 값을 반환한다..

Class	관련 C API	비고
Object getObject(String columnLabel)	해당 사항 없음	입력된 columnLabel에 해당하는 column 값을 반환한다. (CHAR 타입 column은 byte 반환을 지원하지 않으므로 getByte, getBytes 함수를 사용해야 한다.)
boolean wasNull()	해당 사항 없음	항상 false 를 반환한다. (NULL 미지원)

노트

getXXX() 계열 메서드를 사용할 경우 ColumnIndex 사용을 권장한다.
ColumnName을 사용할 경우 SQL 문에서 alias를 지정해야 하고, 해당 alias는 대문자를 사용해야 한다.
예: SELECT payload as pp1 from table;
rs.getString("PP1");

dbmResultSetMeta

Method	관련 C API	비고
int getColumnCount()	dbmGetRowCount	ResultSet의 column 개수를 반환한다.
int getColumnType(int column)	해당 사항 없음	입력된 ColumnIndex에 해당하는 column 의 Java type을 반환한다.
String getColumnClassName(int column)	해당 사항 없음	입력된 ColumnIndex에 해당하는 column 의 Java type name을 반환한다.
String getColumnLabel(int column)	해당 사항 없음	입력된 ColumnIndex에 해당하는 column 의 이름을 반환한다.
String getColumnTypeName(int column)	해당 사항 없음	입력된 ColumnIndex에 해당하는 column 의 native data type을 반환한다.
String getColumnName(int column)	해당 사항 없음	입력된 ColumnIndex에 해당하는 column 의 이름을 반환한다.
int isNullable(int column)	해당 사항 없음	항상 true를 반환한다.
boolean isAutoIncrement(int column)	해당 사항 없음	항상 false를 반환한다.
boolean isCaseSensitive(int column)	해당 사항 없음	항상 true를 반환한다.
boolean isSigned(int column)	해당 사항 없음	항상 true를 반환한다.
boolean isReadOnly(int column)	해당 사항 없음	항상 true를 반환한다.
boolean isWritable(int column)	해당 사항 없음	항상 false를 반환한다.
boolean isDefinitelyWritable(int column)	해당 사항 없음	항상 false를 반환한다.

Connection Pool

GOLDILOCKS LITE에서는 dbmJdbc.jar의 dbmDataSource를 이용하여 connection pool을 구성할 수 있다. dbmConnection이 제공하는 method만 지원한다.

다음은 LITE JDBC에서 제공하는 DataSource를 사용하여 connection pool을 활용하는 예이다.

```
import javax.sql.DataSource;
import java.sql.*;
import java.util.List;
import java.util.ArrayList;
public class DataSourceTest {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = Class.forName("com.dbm.jdbc.dbmDataSource");
        DataSource ds = (DataSource) clazz
                .getConstructor(String.class, String.class, String.class, int.class)
                .newInstance(
                        "jdbc:dbm:///demo",
                        "",   // USER는 지원하지 않음
                        "",   // PASSWORD는 지원하지 않음
                        20    // MaxPoolSize
                );
        List<Connection> connections = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            Connection conn = ds.getConnection();
            connections.add(conn);
            System.out.println("connection " + (i + 1) + " acquired");
        }
        Connection con1 = ds.getConnection();
        Statement stmt1 = con1.createStatement();
        String sql = "select * from dic_table limit 1";
        int i = 1;
        for( Connection conn : connections ) {
            PreparedStatement ps = conn.prepareStatement(sql);
            try (ResultSet rs = ps.executeQuery()) {
                while (rs.next()) {
                    System.out.println(rs.getObject(1) + ", c2=" + rs.getObject(2));
                }
            }
            i++;
            conn.close();
```

```
        }
        ResultSet rs = stmt1.executeQuery( "select count(*) from v$session" );
        rs.next();
        System.out.println( "Connection Count = " + rs.getInt(1) );
    }
}
```

생성자 유형은 다음과 같으며, 각 매개변수에 대한 설명은 아래와 같다.

```
dbmDataSource(String url,
              String user,
              String password,
              int maxPoolSize)
dbmDataSource(String url,
              String user, String password,
              int maxPoolSize,
              long poolMaximumCheckoutTime,
              long poolTimeToWait)
```

- url: 접속 URL을 지정한다.
- user: 미지원 항목이다.
- password: 미지원 항목이다.
- maxPoolSize: Connection pool에 저장할 수 있는 connection의 최대 개수이다.
- poolMaximumCheckoutTime: 미지원 항목이다.
- poolTimeToWait: Connection pool에 가용 자원이 없을 경우 대기하는 시간이다.

> 노트
> 연결 개수는 dbmMetaManager를 통해 다음 질의를 실행하여 확인할 수 있다.
> `SELECT count(*) FROM v$session`

MyBatis 연동

SQLMapper인 MyBatis와 연동하는 예제를 설명한다.

노트

- MyBatis에서 제공하는 full-spec 을 모두 지원하지는 않으며, DDL/DML 과 같은 기본적인 기능만 제공한다.
- 성능 향상을 위해 ExecutorType.REUSE를 사용하여 PreparedStatement를 재사용할 것을 권장한다.

연결 정보 설정

mybatis-config.xml 의 DataSource 항목에 driver와 URL을 기술한다.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "https://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <environments default="dev">
    <environment id="dev">
      <transactionManager type="JDBC"/>
      <dataSource type="UNPOOLED">
        <property name="driver" value="com.dbm.jdbc.dbmDriver"/>
        <property name="url" value="jdbc:dbm:///demo"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="LiteMapper.xml"/>
  </mappers>
</configuration>
```

Mapper.xml 설정 예

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "https://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="LiteMapper">
    <insert id="insert">
        INSERT INTO t_test(id, payload) VALUES(#{id}, #{payload})
    </insert>
```

```
    <update id="update">
        UPDATE t_test SET payload=#{payload} WHERE id=#{id}
    </update>
    <select id="select" resultType="string">
        SELECT payload FROM t_test WHERE id=#{id}
    </select>
    <delete id="delete">
        DELETE FROM t_test WHERE id=#{id}
    </delete>
</mapper>
```

예제 코드

예제 코드는 다음과 같다.

```
import java.io.InputStream;
import java.util.*;
import java.util.concurrent.*;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.*;
public class LiteMybatisTest {
    static final int TOTAL_COUNT = 1_000_000;
    static final String VALUE_1200B;
    static {
        char[] buf = new char[1200];
        Arrays.fill(buf, 'X');
        VALUE_1200B = new String(buf);
    }
    static class Worker implements Runnable {
        int start, end;
        String phase;
        long[] phaseTimes;
        int index;
        SqlSessionFactory sqlSessionFactory;
        Worker(int start, int end, String phase, long[] phaseTimes, int index,
SqlSessionFactory sqlSessionFactory) {
            this.start = start;
            this.end = end;
            this.phase = phase;
            this.phaseTimes = phaseTimes;
            this.index = index;
```

```
            this.sqlSessionFactory = sqlSessionFactory;
        }
        @Override
        public void run() {
            long t0 = System.nanoTime();
            try (SqlSession session = sqlSessionFactory.openSession(ExecutorType.REUSE, true))
{ // autoCommit=true
                switch (phase) {
                    case "INSERT":
                        for (int i = start; i <= end; i++) {
                            session.insert("LiteMapper.insert", Map.of("id", i, "payload",
VALUE_1200B));
                        }
                        break;
                    case "UPDATE":
                        for (int i = start; i <= end; i++) {
                            session.update("LiteMapper.update", Map.of("id", i, "payload",
VALUE_1200B));
                        }
                        break;
                    case "SELECT":
                        for (int i = start; i <= end; i++) {
                            session.selectOne("LiteMapper.select", Map.of("id", i));
                        }
                        break;
                    case "DELETE":
                        for (int i = start; i <= end; i++) {
                            session.delete("LiteMapper.delete", Map.of("id", i));
                        }
                        break;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            long t1 = System.nanoTime();
            phaseTimes[index] = t1 - t0;
        }
    }
    private static void runPhase(String phase, int threadCount, SqlSessionFactory
sqlSessionFactory) throws InterruptedException {
        System.out.println("===== " + phase + " Phase 시작 =====");
```

```
        int perThread = TOTAL_COUNT / threadCount;
        long[] phaseTimes = new long[threadCount];
        Thread[] threads = new Thread[threadCount];
        for (int i = 0; i < threadCount; i++) {
            int start = i * perThread + 1;
            int end = (i == threadCount - 1) ? TOTAL_COUNT : (i + 1) * perThread;
            threads[i] = new Thread(new Worker(start, end, phase, phaseTimes, i,
sqlSessionFactory), "worker-" + i);
            threads[i].start();
        }
        for (Thread t : threads) t.join();
        for (int i = 0; i < threadCount; i++) {
            System.out.printf("Thread-%d : %.3f sec\n", i, phaseTimes[i] / 1_000_000_000.0);
        }
        long maxNs = Arrays.stream(phaseTimes).max().orElse(0L);
        double secTotal = maxNs / 1_000_000_000.0;
        double tps = TOTAL_COUNT / secTotal;
        System.out.printf("Phase Total (max of threads): %.3f sec, TPS=%.2f\n", secTotal, tps
);
        System.out.println("===== " + phase + " Phase 완료 =====\n");
    }
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.out.println("Usage: java LiteMybatisTest <numThreads>");
            return;
        }
        int threadCount = Integer.parseInt(args[0]);
        System.out.println("NumThreads: " + threadCount);
        Class.forName("com.dbm.jdbc.dbmDriver");
        // SqlSessionFactory 생성
        InputStream configStream = Resources.getResourceAsStream("mybatis-config.xml");
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(
configStream);
        // 초기 테이블 생성
        try (var session = sqlSessionFactory.openSession()) {
            try {
                session.getConnection().createStatement().execute("DROP TABLE t_test");
            } catch (Exception ignored) {}
            session.getConnection().createStatement().execute("CREATE TABLE t_test(id int,
payload CHAR(1200))");
            session.getConnection().createStatement().execute("CREATE UNIQUE INDEX idx1 ON t_
```

```
test(id)");
            session.commit();
        }
        runPhase("INSERT", threadCount, sqlSessionFactory);
        runPhase("UPDATE", threadCount, sqlSessionFactory);
        runPhase("SELECT", threadCount, sqlSessionFactory);
        runPhase("DELETE", threadCount, sqlSessionFactory);
    }
}
```

Mybatis Connection Pool 설정

Mybatis config에 다음과 같이 설정한다.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "https://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <environments default="dbm">
    <environment id="dbm">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="com.dbm.jdbc.dbmDriver"/>
        <property name="url" value="jdbc:dbm:///demo"/>
        <property name="username" value=""/>
        <property name="password" value=""/>
        <property name="poolMaximumActiveConnections" value="10"/>
        <property name="poolMaximumIdleConnections" value="10"/>
        <property name="poolMaximumCheckoutTime" value="20000"/>
        <property name="poolTimeToWait" value="20000"/>
        <property name="poolPingEnabled" value="true"/>
        <property name="poolPingQuery" value="select 1 from dual"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper class="PoolTestMapper"/>
  </mappers>
</configuration>
```

노트

GOLDILOCKS LITE에서는 connection pool의 일부 속성과 동작을 지원하지 않는다.

Hibernate 6 연동

ORM/JPA 구현체인 Hibernate와의 연동 예를 설명한다.
Hibernate는 객체 상태를 중심으로 데이터를 관리하며, entity의 상태 변화와 트랜잭션을 자동으로 처리하는 기능을 제공한다. 그러나 내부적으로 트랜잭션 레이어와 세션 관리 등 다양한 오버헤드가 존재하므로, 고성능이 요구되는 경우에는 최적의 선택이 아닐 수 있다. 이러한 경우, Hibernate의 session API에서 제공하는 doWork 메서드를 활용하여 JDBC API를 직접 호출할 수 있다.
이를 통해 Hibernate의 트랜잭션 레이어를 최소화하면서도 JDBC 기반의 높은 성능을 유지할 수 있다.

노트

- Hibernate의 방식은 사용할 수 있으나, 자동 DDL 기능은 지원하지 않으므로 사전에 객체를 생성해야 한다.
  - Temporary table, sequence 객체는 지원하지 않는다.
  - Procedure 및 trigger 는 지원하지 않는다.
  - Data type은 제한적으로 지원된다.
- 처리 가능한 SQL 이 제한되므로, 동적 SQL 조립 방식은 사용할 수 없다.
  - Join 및 aggregation 은 지원하지 않는다.
  - Batch 기능은 지원하지 않으며, commit 단위를 조정하여 처리해야 한다.

Session API 예

다음은 session API를 사용하는 예이다.

```
import org.hibernate.*;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.*;
import org.hibernate.cfg.Environment;
import java.sql.*;
import java.util.*;
import java.util.concurrent.*;
public class LiteHibernateSessionAPITest {
    static final int TOTAL_COUNT = 1_000_000;
```

```
    static final String VALUE_1200B;
    static {
        char[] buf = new char[1200];
        Arrays.fill(buf, 'X');
        VALUE_1200B = new String(buf);
    }
    static class Worker implements Runnable {
        int start, end;
        String phase;
        long[] phaseTimes;
        int index;
        SessionFactory sessionFactory;
        Worker(int start, int end, String phase, long[] phaseTimes, int index, SessionFactory
sessionFactory) {
            this.start = start;
            this.end = end;
            this.phase = phase;
            this.phaseTimes = phaseTimes;
            this.index = index;
            this.sessionFactory = sessionFactory;
        }
        @Override
        public void run() {
            long t0 = System.nanoTime();
            try (Session session = sessionFactory.openSession()) {
                session.setHibernateFlushMode(FlushMode.MANUAL);
                session.setJdbcBatchSize(0); // batch off
                Transaction tx = session.beginTransaction();
                int cnt = 0;
                switch (phase) {
                    case "INSERT":
                        for (int i = start; i <= end; i++) {
                            session.createNativeQuery(
                                    "insert into t_test (id, payload) values(:id, :payload)")
                                    .setParameter("id", i)
                                    .setParameter("payload", VALUE_1200B)
                                    .executeUpdate();
                            if( cnt % 100 == 0 ) {
                                tx.commit();
                                tx = session.beginTransaction();
                            }
```

```
                            cnt++;
                        }
                        tx.commit();
                        tx = session.beginTransaction();
/*
                        session.doWork(conn -> {
                            conn.setAutoCommit(false);
                            PreparedStatement ps = conn.prepareStatement("insert into t_test(
id, payload) values(?, ?)");
                            for (int i = start; i <= end; i++) {
                               ps.setInt(1, i);
                               ps.setString(2, VALUE_1200B);
                               ps.executeUpdate();
                               conn.commit();
                            }
                        });
*/
                        break;
                    case "UPDATE":
                        for (int i = start; i <= end; i++) {
                            tx = session.beginTransaction();
                            session.createNativeQuery(
                                    "UPDATE t_test SET payload=:payload WHERE id=:id")
                                    .setParameter("id", i)
                                    .setParameter("payload", VALUE_1200B)
                                    .executeUpdate();
                            tx.commit();
                        }
                        break;
                    case "SELECT":
                        for (int i = start; i <= end; i++) {
                            String result = session.createQuery(
                                    "SELECT payload FROM t_test WHERE id=:id", String.class)
                                    .setParameter("id", i)
                                    .getSingleResult();
                        }
                        break;
                    case "DELETE":
                        for (int i = start; i <= end; i++) {
                            tx = session.beginTransaction();
                            session.createNativeQuery(
```

```
                                    "DELETE FROM t_test WHERE id=:id")
                                    .setParameter("id", i)
                                    .executeUpdate();
                            tx.commit();
                        }
                        break;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            long t1 = System.nanoTime();
            phaseTimes[index] = t1 - t0;
        }
    }
    private static void runPhase(String phase, int threadCount, SessionFactory sessionFactory)
throws InterruptedException {
        System.out.println("===== " + phase + " Phase 시작 =====");
        int perThread = TOTAL_COUNT / threadCount;
        long[] phaseTimes = new long[threadCount];
        Thread[] threads = new Thread[threadCount];
        for (int i = 0; i < threadCount; i++) {
            int start = i * perThread + 1;
            int end = (i == threadCount - 1) ? TOTAL_COUNT : (i + 1) * perThread;
            threads[i] = new Thread(new Worker(start, end, phase, phaseTimes, i,
sessionFactory), "worker-" + i);
            threads[i].start();
        }
        for (Thread t : threads) t.join();
        for (int i = 0; i < threadCount; i++) {
            System.out.printf("Thread-%d : %.3f sec\n", i, phaseTimes[i] / 1_000_000_000.0);
        }
        long maxNs = 0;
        for (long t : phaseTimes) maxNs = Math.max(maxNs, t);
        double secTotal = maxNs / 1_000_000_000.0;
        double tps = TOTAL_COUNT / secTotal;
        System.out.printf("Phase Total (max of threads): %.3f sec, TPS=%.2f\n", secTotal, tps
);
        System.out.println("===== " + phase + " Phase 완료 =====\n");
    }
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
```

```
            System.out.println("Usage: java LiteHibernateTest <numThreads>");
            return;
        }
        int threadCount = Integer.parseInt(args[0]);
        System.out.println("NumThreads: " + threadCount);
        // Hibernate 설정
        Map<String, Object> settings = new HashMap<>();
        settings.put(Environment.DRIVER, "com.dbm.jdbc.dbmDriver");
        settings.put(Environment.URL, "jdbc:dbm:///demo");
        settings.put(Environment.DIALECT, "dbmLiteDialect"); // 제공하는 Lite Dialect
        settings.put(Environment.SHOW_SQL, "false");
        settings.put(Environment.HBM2DDL_AUTO, "none");
        settings.put("hibernate.jdbc.batch_size", 0); // batch off
        settings.put("hibernate.statement_cache.size", 100);
        settings.put(Environment.ORDER_UPDATES, false);
        settings.put(Environment.ORDER_INSERTS, false);
        settings.put("hibernate.query.plan_cache_max_size", 2048);
        settings.put("hibernate.query.plan_parameter_metadata_max_size", 128);
        settings.put("hibernate.connection.autocommit", "false");
        settings.put("hibernate.connection.provider_disables_autocommit", "true");
        StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
                .applySettings(settings)
                .build();
        MetadataSources sources = new MetadataSources(registry);
        SessionFactory sessionFactory = sources.buildMetadata().buildSessionFactory();
        // 테이블 초기화
        try (Session session = sessionFactory.openSession()) {
            Transaction tx = session.beginTransaction();
            session.createNativeQuery("DROP TABLE t_test").executeUpdate();
            tx.commit();
        } catch( Exception ignored){}
        try (Session session = sessionFactory.openSession()) {
            Transaction tx = session.beginTransaction();
            session.createNativeQuery("CREATE TABLE t_test(id int, payload CHAR(1200))").
executeUpdate();
            session.createNativeQuery("CREATE UNIQUE INDEX idx1 ON t_test(id)").executeUpdate
();
            tx.commit();
        }
        runPhase("INSERT", threadCount, sessionFactory);
        runPhase("UPDATE", threadCount, sessionFactory);
```

```
        runPhase("SELECT", threadCount, sessionFactory);
        runPhase("DELETE", threadCount, sessionFactory);
        sessionFactory.close();
        StandardServiceRegistryBuilder.destroy(registry);
    }
}
```

- 빌드 예

```
shell> javac -cp ${DBM_HOME}/lib/dbmJdbc.jar:jlib/*:. LiteHibernateSessionAPITest.java
dbmLiteDialect.java
** dbmJdbc.jar : LITE JDBC driver
** jlib/* : hibernate가 요구하는 jar경로
** dbmDialect.java : hibernate용 LITE Dialect class
```

Entity 방식

다음은 entity 방식의 예제 코드이다.

```
import jakarta.persistence.*;
import org.hibernate.*;
import org.hibernate.cfg.Environment;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ThreadLocalRandom;
public class LiteHibernateORMTest {
    static final int TOTAL_COUNT = 1_000_000;
    static final String VALUE_1200B;
    static {
        char[] buf = new char[1200];
        Arrays.fill(buf, 'X');
        VALUE_1200B = new String(buf);
    }
    // ===== Entity 매핑 =====
    @Entity(name = "TTest")
    @Table(name = "t_test")
    public static class TTest {
```

```
        @Id
        private int id;
        @Column(length = 1200)
        private String payload;
        public TTest() {}
        public TTest(int id, String payload) { this.id = id; this.payload = payload; }
        public int getId() { return id; }
        public void setId(int id) { this.id = id; }
        public String getPayload() { return payload; }
        public void setPayload(String payload) { this.payload = payload; }
    }
    static class Worker implements Runnable {
        int start, end;
        String phase;
        long[] phaseTimes;
        int index;
        EntityManagerFactory emf;
        Worker(int start, int end, String phase, long[] phaseTimes, int index,
EntityManagerFactory emf) {
            this.start = start;
            this.end = end;
            this.phase = phase;
            this.phaseTimes = phaseTimes;
            this.index = index;
            this.emf = emf;
        }
        @Override
        public void run() {
            long t0 = System.nanoTime();
            EntityManager em = emf.createEntityManager();
            try {
                switch (phase) {
                    case "INSERT":
                        for (int i = start; i <= end; i++) {
                            EntityTransaction tx = em.getTransaction();
                            tx.begin();
                            em.persist(new TTest(i, VALUE_1200B));
                            tx.commit();
                        }
                        break;
                    case "UPDATE":
```

```
                    for (int i = start; i <= end; i++) {
                        EntityTransaction tx = em.getTransaction();
                        tx.begin();
                        TTest entity = em.find(TTest.class, i);
                        if (entity != null) {
                            entity.setPayload(VALUE_1200B);
                        }
                        tx.commit();
                    }
                    break;
                case "SELECT":
                    for (int i = start; i <= end; i++) {
                        TTest entity = em.find(TTest.class, i);
                    }
                    break;
                case "DELETE":
                    for (int i = start; i <= end; i++) {
                        EntityTransaction tx = em.getTransaction();
                        tx.begin();
                        TTest entity = em.find(TTest.class, i);
                        if (entity != null) em.remove(entity);
                        tx.commit();
                    }
                    break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            em.close();
        }
        long t1 = System.nanoTime();
        phaseTimes[index] = t1 - t0;
    }
}
    private static void runPhase(String phase, int threadCount, EntityManagerFactory emf)
throws InterruptedException {
        System.out.println("===== " + phase + " Phase 시작 =====");
        int perThread = TOTAL_COUNT / threadCount;
        long[] phaseTimes = new long[threadCount];
        Thread[] threads = new Thread[threadCount];
        for (int i = 0; i < threadCount; i++) {
```

```
            int start = i * perThread + 1;
            int end = (i == threadCount - 1) ? TOTAL_COUNT : (i + 1) * perThread;
            threads[i] = new Thread(new Worker(start, end, phase, phaseTimes, i, emf), "worker
-" + i);
            threads[i].start();
        }
        for (Thread t : threads) t.join();
        for (int i = 0; i < threadCount; i++) {
            System.out.printf("Thread-%d : %.3f sec\n", i, phaseTimes[i] / 1_000_000_000.0);
        }
        long maxNs = 0;
        for (long t : phaseTimes) maxNs = Math.max(maxNs, t);
        double secTotal = maxNs / 1_000_000_000.0;
        double tps = TOTAL_COUNT / secTotal;
        System.out.printf("Phase Total (max of threads): %.3f sec, TPS=%.2f\n", secTotal, tps
);
        System.out.println("===== " + phase + " Phase 완료 =====\n");
    }
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.out.println("Usage: java LiteHibernateORMTest <numThreads>");
            return;
        }
        int threadCount = Integer.parseInt(args[0]);
        System.out.println("NumThreads: " + threadCount);
        // Hibernate + JPA 설정
        Map<String, Object> settings = new HashMap<>();
        settings.put(Environment.DRIVER, "com.dbm.jdbc.dbmDriver");
        settings.put(Environment.URL, "jdbc:dbm:///demo");
        settings.put(Environment.DIALECT, "dbmLiteDialect");
        settings.put(Environment.SHOW_SQL, "false");
        settings.put(Environment.HBM2DDL_AUTO, "none");
        settings.put("hibernate.statement_cache.size", 100);
        settings.put("hibernate.connection.autocommit", "false");
        StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
                .applySettings(settings)
                .build();
        MetadataSources sources = new MetadataSources(registry);
        SessionFactory sessionFactory = sources.buildMetadata().buildSessionFactory();
        sources.addAnnotatedClass(TTest.class);
        EntityManagerFactory emf = sources.buildMetadata().buildSessionFactory();
```

```
        try (Session session = sessionFactory.openSession()) {
            Transaction tx = session.beginTransaction();
            try {
                // 기존 테이블을 삭제하되, 없을 경우 발생하는 예외는 무시한다.
                session.createNativeQuery("DROP TABLE t_test").executeUpdate();
            } catch (Exception ignored) { }
            tx.commit();
        }
        try (Session session = sessionFactory.openSession()) {
            Transaction tx = session.beginTransaction();
            // 테이블 생성
            session.createNativeQuery("CREATE TABLE t_test (id int, payload CHAR(1200))").
executeUpdate();
            // 인덱스 생성
            session.createNativeQuery("CREATE UNIQUE INDEX idx1 ON t_test(id)").executeUpdate
();
            tx.commit();
        }
        runPhase("INSERT", threadCount, emf);
        runPhase("UPDATE", threadCount, emf);
        runPhase("SELECT", threadCount, emf);
        runPhase("DELETE", threadCount, emf);
        emf.close();
        StandardServiceRegistryBuilder.destroy(registry);
    }
}
```

2.4 Python 연동

Python에서는 unixODBC를 이용하여 연동할 수 있다.
unixODBC 설치 후 DSN 방식으로 접속할 수 있으며, 설정 예시는 다음과 같다.

```
# $ cat ~/.odbc.ini
[LITE]
HOST = 127.0.0.1  # DA일 경우 영향 없음
PORT = 27584      # DA일 경우 영향 없음
INSTANCE = demo   # 접근할 Instance Name
```

```
DA_MODE = true    # DA=true, TCP=false (TCP로 접속할 경우 dbmListener 필요)
DRIVER = /mnt/md1/ssd_home/lim272/new_lite/pkg/lib/libdbmCore.so  # 설치 경로
[ODBC]
trace = yes                # 오류 발생 시 추적용
tracefile = /tmp/odbc.log  # 추적 로그 파일 경로
```

다음은 연동 예제 코드이다.

```
import pyodbc
from datetime import datetime
def main():
    conn_str = "DSN=LITE"
    conn = None
    cur = None
    try:
        conn = pyodbc.connect(conn_str, autocommit=False)
        cur = conn.cursor()
        # ------------------------------------------------
        # DROP TABLE
        # ------------------------------------------------
        try:
            cur.execute("DROP TABLE t1")
            conn.commit()
        except Exception:
            conn.rollback()  # Table이 존재하지 않을 수 있음
        # ------------------------------------------------
        # CREATE TABLE
        # (short -> SMALLINT, long -> BIGINT)
        # ------------------------------------------------
        cur.execute("""
            CREATE TABLE t1 (
                c1 INT,
                c2 SHORT,
                c3 LONG,
                c4 DOUBLE,
                c5 FLOAT,
                c6 CHAR(1000),
                c7 DATE
            )
        """)
        # ------------------------------------------------
```

```
# CREATE UNIQUE INDEX
# ------------------------------------------------
cur.execute("CREATE UNIQUE INDEX idx_t1 ON t1 (c1)")
conn.commit()
# ------------------------------------------------
# INSERT 10 ROWS
# ------------------------------------------------
insert_sql = """
    INSERT INTO t1 (c1, c2, c3, c4, c5, c6, c7)
    VALUES (?, ?, ?, ?, ?, ?, ?)
"""
for i in range(1, 11):
    cur.execute(
        insert_sql,
        i,                  # c1
        i % 100,             # c2
        i * 1000,            # c3
        i * 1.1,             # c4
        i * 2.2,             # c5
        f"row-{i}".ljust(1000),
        datetime.now()         # c7
    )
conn.commit()
# ------------------------------------------------
# UPDATE 10 TIMES
# ------------------------------------------------
update_sql = """
    UPDATE t1
       SET c4 = c4 + 1.0,
           c5 = c5 + 1.0
     WHERE c1 = ?
"""
for i in range(1, 11):
    cur.execute(update_sql, i)
conn.commit()
# ------------------------------------------------
# SELECT 10 ROWS
# ------------------------------------------------
cur.execute("""
    SELECT c1, c2, c3, c4, c5, c6, c7
      FROM t1
```

```
            ORDER BY 1
        """)
        rows = cur.fetchall()
        for r in rows:
            print(
                f"c1={r.C1}, c2={r.C2}, c3={r.C3}, "
                f"c4={r.C4}, c5={r.C5}, c7={r.C7}"
            )
        # ------------------------------------------------
        # DELETE 10 ROWS
        # ------------------------------------------------
        cur.execute("DELETE FROM t1")
        conn.commit()
    except Exception as e:
        if conn:
            conn.rollback()
        print("ERROR:", e)
    finally:
        if cur:
            cur.close()
        if conn:
            conn.close()
if __name__ == "__main__":
    main()
```

노트

- DATE column은 datetime을 사용한다.
- unixODBC를 통해 수행되므로 성능에는 제한이 있다.

2.5 GO Lang

다음은 GoLang에서 LITE를 사용하는 예이다.

```
mkdir gotest
cd gotest
go mod init gotest
go get github.com/alexbrainman/odbc
go run gotest.go
```

다음은 gotest.go 예제이다.

```
package main
import (
        "database/sql"
        "fmt"
        "log"
        "time"
        _ "github.com/alexbrainman/odbc"
)
func main() {
        // ------------------------------------------------
        // CONNECT
        // ------------------------------------------------
        db, err := sql.Open("odbc", "DSN=LITE")
        if err != nil {
                log.Fatal(err)
        }
        defer db.Close()
        // ------------------------------------------------
        // DROP TABLE (ignore error)
        // ------------------------------------------------
        func() {
                tx, err := db.Begin()
                if err != nil {
                        log.Fatal(err)
                }
                defer tx.Rollback()
                _, _ = tx.Exec("DROP TABLE t1")
                tx.Commit()
        }()
        // ------------------------------------------------
        // CREATE TABLE + INDEX
        // ------------------------------------------------
        tx, err := db.Begin()
```

```
	if err != nil {
		log.Fatal(err)
	}
	_, err = tx.Exec(`
		CREATE TABLE t1 (
			c1 INT,
			c2 SHORT,
			c3 LONG,
			c4 DOUBLE,
			c5 FLOAT,
			c6 CHAR(1000),
			c7 DATE
		)
	`)
	if err != nil {
		tx.Rollback()
		log.Fatal(err)
	}
	_, err = tx.Exec("CREATE UNIQUE INDEX idx_t1 ON t1 (c1)")
	if err != nil {
		tx.Rollback()
		log.Fatal(err)
	}
	if err = tx.Commit(); err != nil {
		log.Fatal(err)
	}
	// ------------------------------------------------
	// INSERT 10 ROWS
	// ------------------------------------------------
	tx, err = db.Begin()
	if err != nil {
		log.Fatal(err)
	}
	insertSQL := `
		INSERT INTO t1 (c1, c2, c3, c4, c5, c6, c7)
		VALUES (?, ?, ?, ?, ?, ?, ?)
	`
	stmt, err := tx.Prepare(insertSQL)
	if err != nil {
		tx.Rollback()
		log.Fatal(err)
```

```
}
defer stmt.Close()
for i := 1; i <= 10; i++ {
        _, err = stmt.Exec(
                i,                  // c1
                i%100,              // c2
                i*1000,             // c3
                float64(i)*1.1,     // c4
                float64(i)*2.2,     // c5
                fmt.Sprintf("row-%d", i), // CHAR(1000)
                time.Now(),         // DATE (TIMESTAMP로 바인딩됨)
        )
        if err != nil {
                tx.Rollback()
                log.Fatal(err)
        }
}
if err = tx.Commit(); err != nil {
        log.Fatal(err)
}
// ------------------------------------------------
// UPDATE 10 TIMES
// ------------------------------------------------
tx, err = db.Begin()
if err != nil {
        log.Fatal(err)
}
updateSQL := `
        UPDATE t1
           SET c4 = c4 + 1.0,
               c5 = c5 + 1.0
         WHERE c1 = ?
`
stmt, err = tx.Prepare(updateSQL)
if err != nil {
        tx.Rollback()
        log.Fatal(err)
}
defer stmt.Close()
for i := 1; i <= 10; i++ {
        if _, err = stmt.Exec(i); err != nil {
```

```
                tx.Rollback()
                log.Fatal(err)
        }
}
if err = tx.Commit(); err != nil {
        log.Fatal(err)
}
// ------------------------------------------------
// SELECT 10 ROWS
// ------------------------------------------------
rows, err := db.Query(`
        SELECT c1, c2, c3, c4, c5, c6, c7
          FROM t1
         ORDER BY 1
`)
if err != nil {
        log.Fatal(err)
}
defer rows.Close()
for rows.Next() {
        var (
                c1 int
                c2 int16
                c3 int64
                c4 float64
                c5 float64
                c6 string
                c7 time.Time
        )
        if err := rows.Scan(&c1, &c2, &c3, &c4, &c5, &c6, &c7); err != nil {
                log.Fatal(err)
        }
        fmt.Printf(
                "c1=%d, c2=%d, c3=%d, c4=%f, c5=%f, c7=%s\n",
                c1, c2, c3, c4, c5, c7.Format(time.RFC3339),
        )
}
// ------------------------------------------------
// DELETE
// ------------------------------------------------
tx, err = db.Begin()
```

```
        if err != nil {
                log.Fatal(err)
        }
        if _, err = tx.Exec("DELETE FROM t1"); err != nil {
                tx.Rollback()
                log.Fatal(err)
        }
        if err = tx.Commit(); err != nil {
                log.Fatal(err)
        }
}
```

2.6 Error Message

GOLDILOCKS LITE에서 발생하는 오류 코드의 원인 및 조치 방법을 설명한다.

ERR-22001: invalid parameters or usage at internal processing

Cause	올바르지 않은 사용자 변수 입력 또는 내부 처리 과정에서 인자 검증에 오류가 발생한 경우이다.
Action	전달한 인자 값과 타입이 올바른지 확인한다.

ERR-22002: fail to alloc memory from OS (errno=%d)

Cause	내부 처리 과정에서 메모리 할당에 실패한 경우 발생한다.
Action	시스템 혹은 프로세스의 가용 메모리를 확인한다. (메모리 사용량, kernel 및 ulimit 설정등을 체크한다.)

ERR-22003: fail to alloc memory from dbmAllocator

Cause	내부 Memory Allocator의 확장이 필요한 상황에서 메모리가 부족한 경우 발생한다.
Action	시스템 혹은 프로세스의 가용 메모리를 확인한다. (메모리 사용량, kernel 및 ulimit 설정등을 체크한다.)

ERR-22004: not implemented

Cause	지원되지 않는 SQL/명령 등이 사용될 경우 발생한다.
Action	해당 기능은 사용할 수 없다.

ERR-22005: a shared memory already exists

Cause	동일한 이름의 공유 메모리 세그먼트가 이미 존재하는 경우에 발생한다.
Action	**딕셔너리 테이블(DIC_TABLE, DIC_INDEX, DIC_INST)**에서 해당 객체 존재 유무를 확인한다. 만일, 딕셔너리 테이블에서 조회되지 않는 경우 딕셔너리 테이블에서 삭제가 되었지만 세그먼트는 삭제가 되지 않은 상황이다. /dev/shm 에서 직접 세그먼트를 삭제한다.

ERR-22006: fail to create a shared memory segment

Cause	디스크 공간 또는 권한 문제로 세그먼트 생성에 실패한 경우에 발생한다.
Action	사용자에게 읽기, 쓰기 권한을 부여한다.

ERR-22007: fail to initialize a shared memory segment

Cause	공유 메모리 세그먼트를 생성하는 과정에서 공간 부족 등으로 발생한다.
Action	/dev/shm의 마운트 크기 및 사용 가능한 공간을 확인한다.

ERR-22008: fail to attach a shared memory segment

Cause	공유 메모리 세그먼트가 존재하지 않거나 접근 불가로 세그먼트 attach에 실패할 경우 발생한다.
Action	대상 세그먼트 존재 여부 및 권한 등을 확인한다.

ERR-22009: fail to open a shared memory

Cause	/dev/shm 에 존재하는 공유 메모리 세그먼트를 열지 못했을 경우에 발생한다.
Action	대상 세그먼트 존재 여부 및 권한 등을 확인한다.

ERR-22010: fail to get a information about shm

Cause	/dev/shm의 공유메모리 세그먼트 파일 정보 획득에 실패한 경우 발생한다.
Action	파일 시스템 오류를 점검한다.

ERR-22011: invalid segment size to attach a shm

Cause	공유 메모리 세그먼트의 실제 크기가 필요한 크기보다 작을 때 발생한다.
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22012: fail to call a mmap to attach a shared memory segment

Cause	공유 메모리 세그먼트 attach를 위해 호출한 system call 호출에 실패한 경우에 발생한다.
Action	Trace log에서 system call에 관한 errno를 확인하여 조치한다. 1) ENOMEM - 메모리 공간 부족 2) EACCES - 권한 문제 3) EBADF - 파일 손상등의 문제

ERR-22013: fail to detach a shared memory segment

Cause	해당 공유 메모리 세그먼트가 이미 해제되어, detach에 실패할 경우에 발생한다.
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22014: fail to drop a shared segment memory

Cause	Deprecated
Action	

ERR-22015: fail to create a directory for shared-memory

Cause	Deprecated
Action	

ERR-22016: invalid slot number (SlotId=%ld)

Cause	내부 처리 중 유효하지 않은 메모리에 접근하는 경우에 발생한다.
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22017: fail to attach dictionary (execute initdb)

Cause	Deprecated
Action	

ERR-22018: a operation not allowed without an instance

Cause	내부 처리 과정에서 인스턴스 이름 검증에 실패한 경우 발생한다.
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22019: object does not exists

Cause	테이블이 존재하지 않는 경우에 발생한다.
Action	딕셔너리 테이블을 조회하여 테이블 존재 여부를 확인한다.

ERR-22020: Column does not exists

Cause	컬럼이 존재하지 않는 경우에 발생한다.
Action	dbmMetaManager에서 **desc 명령어** 혹은 딕셔너리 테이블을 통해 컬럼 존재 여부를 확인한다.

ERR-22021: a segment has no space to extend because of reached max_segment

Cause	Deprecated
Action	

ERR-22022: a segment has no space to extend because of reached max_size

Cause	공유 메모리 세그먼트가 확장 가능한 최대 사이즈를 초과한 경우에 발생한다.
Action	해당 상황은 더 이상 확장 불가능하며 긴급한 경우 아래와 같이 조치한다. 1) 테이블의 레코드 삭제가 가능한 경우, DELETE 구문을 이용해 삭제한다. 2) 테이블의 레코드 삭제가 불가능한 경우 dbmExp로 데이터를 백업한 후 새로운 테이블을 생성하고 dbmImp를 통해 데이터를 복원한다.

ERR-22023: fail to connect target server

Cause	원격 서버의 dbmListener에 접속할 수 없는 경우에 발생한다. 1) 연결정보에 IP/포트 오류 2) dbmListener 구동되지 않은 경우 3) network, 방화벽 설정에 의해 연결되지 못하는 경우
Action	1) 연결정보가 올바른지 확인한다. 2) dbmListener 프로세스 상태 및 네트워크 상태를 확인한다.

ERR-22024: fail to send a packet

Cause	통신에 사용하던 소켓이 유효하지 않을 경우 발생한다.
Action	네트워크를 점검하고 dbmListener의 상태를 확인한다.

ERR-22025: fail to receive a packet

Cause	원격 서버로부터 설정된 Timeout 을 초과하여 응답 수신에 실패한 경우에 발생한다.
Action	1) 서버(dbmListener)의 소켓 상태를 확인한다. 2) 상황에 따라 클라이언트의 설정 파일에서 Timeout 시간을 조정한다. (기본 설정: 10초)

ERR-22026: fail to send or receive a packet for HB

Cause	Deprecated
Action	

ERR-22027: fail to initialize a handle

Cause	Deprecated
Action	

ERR-22028: fail to alloc a memory for handle

Cause	Deprecated
Action	

ERR-22029: a pointer has to be set to null to initialize a handle

Cause	dbmInitHandle의 핸들 인자가 NULL로 초기화 되지 않은 경우에 발생한다.
Action	사용자 Handle변수를 NULL로 초기화 한다.

ERR-22030: fail to free a handle

Cause	Deprecated
Action	

ERR-22031: fail to alloc a memory for statement

Cause	SQL 처리 과정에 메모리가 부족한 경우에 발생한다.
Action	시스템 가용 메모리 및 환경설정등을 체크한다.

ERR-22032: fail to alloc a memory for parser-context

Cause	SQL 파싱 과정에서 메모리가 부족한 경우에 발생한다.
Action	시스템 가용 메모리 및 환경설정등을 체크한다.

ERR-22033: fail to execute a statement

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22034: invalid stmt type

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22035: invalid plan type

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22036: invalid data type

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22037: invalid table size option

Cause	테이블 생성 시 입력한 init, extend, max의 설정값이 올바르지 않은 경우에 발생한다.
Action	생성할 테이블 세그먼트의 초기값과 확장값의 합이 최대값보다 크면 안된다. 다음 조건에 맞게 값을 설정한다. [init 값] + [extend 값] <= [max 값]

ERR-22038: fail to prepare a statement

Cause	Goldilocks lite에서 지원하지 않는 구문이거나 잘못된 문법을 사용할 경우 발생한다.
Action	매뉴얼을 참고하여 구문의 지원 여부를 확인하고 알맞은 문법으로 사용한다.

ERR-22039: fail to finalize a stmt

Cause	dbmFreeStmt 실행에 실패하는 경우에 발생한다.
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22040: invalid expr type

Cause	사용자의 SQL에 사용된 표현식이 올바르지 않거나 지원되지 않는 경우에 발생한다.
Action	매뉴얼을 참고하여 지원 여부를 확인하고 알맞은 표현식을 사용한다.

ERR-22041: fail to alloc a memory for something

Cause	SQL 내부처리 과정에서 메모리가 부족할 경우 발생한다.
Action	시스템 가용 메모리 및 환경설정등을 체크한다.

ERR-22042: invalid built-in function

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22043: invalid segment

Cause	Process가 세그먼트에 lock을 점유하고 비정상적으로 종료한 경우 발생한다.
Action	dbmDump로 세그먼트 상태를 확인(Lock)하고 **ALTER SYSTEM REFINE 구문**을 사용해 세그먼트를 유효한 상태로 복구한다.

ERR-22044: fail to alloc a trans for current-session

Cause	동시 접속 가능한 세션의 최대 개수(1024)를 초과하는 경우에 발생한다.
Action	dbmMetaManager에 **admin으로 접속**하여 접속 중인 세션 수(**v$session 테이블**)를 확인한다. 필요 시, 일부 세션을 종료한다.

ERR-22045: the number of binding data does not match the target list

Cause	리턴 받을 컬럼의 개수와 바인딩한 호스트 변수의 개수가 다른 경우 발생한다.
Action	리턴 받을 컬럼의 개수 및 바인딩 호스트 변수의 개수가 일치하는지 확인한다.

ERR-22046: column does not exists

Cause	Deprecated
Action	

ERR-22047: invalid expression type

Cause	사용자 SQL에 사용된 표현식이 올바르지 않거나 지원되지 않는 경우에 발생한다.
Action	매뉴얼을 참고하여 지원 여부를 확인하고 올바른 표현식을 사용한다.

ERR-22048: fail to convert a data as invalid data-type or value-size or origin-value etc.

Cause	표현식 처리 과정에서 데이터 타입이나 크기 등이 호환되지 않을 때 발생한다.
Action	매뉴얼을 참고하여 올바른 표현식을 사용한다.

ERR-22049: fail to bind a column (%s)

Cause	데이터 삽입 시, 특정 컬럼에 대한 binding이 실패하는 경우에 발생한다.
Action	dbmMetaManager에서 **struct out** 명령어를 사용하여 대상 테이블의 컬럼 타입을 확인하여 재시도한다.

ERR-22050: fail to convert data as overflow

Cause	문자열을 Int, short 등의 숫자형으로 변환하는 과정에서 overflow가 발생한 경우이다.
Action	조회한 값이 Bind할 변수의 데이터 타입 범위를 초과하지 않는지 확인한다.

ERR-22051: no more data to fetch

Cause	Deprecated
Action	

ERR-22052: a operation can not be executed because of division by zero

Cause	표현식에서 나눗셈 연산의 분모가 0인 경우 발생한다.
Action	0이 아닌 값으로 변경한 후 재시도한다.

ERR-22053: invalid group-by or target-list to execute group-by

Cause	Deprecated
Action	

ERR-22054: index does not exist

Cause	지정한 인덱스가 존재하지 않는 경우 발생한다. 1) 존재하지 않는 인덱스에 대한 drop을 시도한 경우 2) 존재하지 않는 인덱스에 대한 rebuild를 시도한 경우 3) 존재하지 않는 인덱스를 setIndex로 지정한 경우
Action	**DIC_INDEX** 테이블에서 인덱스 존재 여부를 확인하고 인덱스를 사용한다.

ERR-22055: index key value duplicated

Cause	중복이 허용되지 않는 인덱스(unique index)에 중복되는 key를 삽입하는 경우 발생한다.
Action	중복되지 않도록 key 값을 확인하여 삽입한다.

ERR-22056: index key not found

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22057: invalid log type

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22058: column duplicated

Cause	컬럼 이름을 수정하거나 새로운 컬럼을 추가할 때, 기존 컬럼과 중복된 이름을 사용할 경우 발생한다.
Action	dbmMetaManager에서 **desc 명령어**로 컬럼을 확인한다.

ERR-22059: invalid data size (Limit=%d : InputSize=%d)

Cause	테이블에 데이터 삽입 시 API의 인자로 테이블 Row 크기보다 큰 값이 입력된 경우에 발생한다.
Action	dbmMetaManager에서 **struct out 명령어**를 사용해 테이블 구조체를 확인한다. 테이블에 저장 가능한 Row 크기를 넘지 않도록 처리한다.

ERR-22060: fail to change SCN of row (Segment=%s, SlotId=%ld)

Cause	Deprecated
Action	

ERR-22061: invalid scn (SCN=%ld)

Cause	Deprecated
Action	

ERR-22062: fail to process a function to commit (log=%s)

Cause	커밋을 수행하는 동안 다른 트랜잭션에 의해 테이블에 DDL이 수행되거나 인덱스가 Invalid 상태로 변경되어 커밋에 실패한 경우다.
Action	Trace log를 추적하여 원인을 분석하고 조치한다.

ERR-22063: fail to process a function to rollback (log=%s)

Cause	롤백을 수행하는 동안 다른 트랜잭션에 의해 테이블에 DDL이 수행되어 롤백에 실패한 경우다.
Action	Trace log를 추적하여 원인을 분석하고 조치한다.

ERR-22064: an index with the same ordering key already created (%s)

Cause	인덱스를 생성할 때 기존 인덱스와 동일한 Key 구성 및 정렬 순서를 가진 인덱스가 이미 존재하는 경우 발생한다.
Action	dbmMetaManager에서 **desc 명령어**로 기존 인덱스를 확인한다.

ERR-22065: a column definition duplicated (%s)

Cause	테이블 생성 시 컬럼 이름이 중복된 경우 발생한다.
Action	컬럼명을 확인하여 재시도 한다.

ERR-22066: fail to open a disk logfile (%s) (errno=%d)

Cause	디스크 로그 모드에서 로깅을 위한 파일을 열지 못한 경우 발생한다. 로깅을 위해서는 파일/디렉토리에 대한 읽기, 쓰기 권한이 필요하다.
Action	디스크 로그를 기록하는 설정(**DBM_DISK_LOG_DIR**)을 참고하여 해당 디렉토리 및 로그 파일에 필요한 권한이 있는지 확인한다.

ERR-22067: fail to locate a position of disk logfile (%s) (errno=%d)

Cause	Deprecated
Action	

ERR-22068: fail to switch a disk logfile

Cause	새로운 로그 파일을 생성하는데 실패한 경우 발생한다.
Action	로그를 기록하는 디렉토리의 설정 및 권한이나 저장공간을 확인한다.

ERR-22069: fail to write a disk logfile (errno=%d)

Cause	트랜잭션 로그 파일 기록에 실패한 경우 발생한다.
Action	디스크 공간이 충분한지 확인한다.

ERR-22070: fail to sync a disk logfile (errno=%d)

Cause	트랜잭션 로그 파일을 디스크로 sync 하는데 실패할 경우에 발생한다.
Action	파일 시스템 오류를 점검한다.

ERR-22071: fail to read from a disk logfile (errno=%d)

Cause	트랜잭션 로그 파일을 읽지 못하는 경우에 발생한다.
Action	파일 시스템 오류를 점검한다.

ERR-22072: invalid disk log block

Cause	Deprecated
Action	

ERR-22073: an operation cannot be executed on the target table (check table type)

Cause	대상 테이블이 지원하지 않는 기능을 실행하려 할 경우에 발생한다.
Action	dbmMetaManager의 **desc 명령어**를 통해 테이블 타입을 확인하고, 해당 테이블 타입에 맞는 사용 방법을 매뉴얼에서 확인한다.

ERR-22074: column(ID, INTIME) can not be specified at the target list when the target table is queue-type

Cause	**Queue 타입 테이블**에 enqueue 할 때 ID 및 INTIME 컬럼의 값을 지정하여 삽입을 시도하는 경우 발생한다. Queue 타입 테이블은 priority, msg_size, message 컬럼에만 값을 지정하여 삽입할 수 있다.
Action	Queue 타입 테이블에 데이터를 삽입할 때는 priority, msg_size, message 컬럼의 값만 지정한다.

ERR-22075: an index (%s) invalid stat, need to rebuild index

Cause	인덱스의 Lock을 점유한 세션이 Lock를 해제하지 못한 상태로 비정상 종료되어 복구가 필요한 경우 발생한다.
Action	dbmDump 로 인덱스 상태를 확인하고(Node LockTx) dbmMetaManager에서 refine명령을 수행한다.

ERR-22076: not supported transaction

Cause	지원하지 않는 기능을 실행할 경우에 발생한다.
Action	매뉴얼을 참고하여 올바르게 사용한다.

ERR-22077: an instance already exists

Cause	생성하려는 인스턴스가 이미 존재하는 경우에 발생한다.
Action	dbmMetaManager에서 **DIC_INST 테이블**을 조회하여 존재하는 인스턴스 목록을 확인한다. (기존 인스턴스 삭제 조치를 하는 경우 딕셔너리 테이블에서 조회 가능하지만, 세그먼트가 존재하지 않는다면 삭제가 동작하지 않는다. 이때 **force 옵션**을 사용하여 Dictionary정보를 정리할 수 있다.)

ERR-22078: an index already exists

Cause	생성하려는 인덱스 이름이 대상 테이블에 이미 존재하는 경우에 발생한다.
Action	dbmMetaManager에서 **DIC_INDEX 테이블**을 조회해 테이블 존재 유무를 확인한다.

ERR-22079: a table already exists

Cause	생성하려는 테이블이 해당 인스턴스에 이미 존재하는 경우에 발생한다.
Action	dbmMetaManager에서 **DIC_TABLE 테이블**을 조회해 테이블 존재 유무를 확인한다.

ERR-22080: a dead-lock detection

Cause	Deadlock이 감지되어 victim으로 선택된 세션에 발생한다.
Action	에러가 발생한 세션에서 Rollback을 수행하여 Deadlock 상태를 해소한다.

ERR-22081: a length of object too long (max %d bytes)

Cause	Object 이름의 길이가 64byte를 초과하는 경우 발생한다.
Action	Object 이름을 64byte 이하로 지정한다.

ERR-22082: invalid binding parameters (index or name not exist)

Cause	사용자가 바인딩하려는 id 또는 이름이 Prepare된 SQL에 존재하지 않는 경우에 발생한다.
Action	dbmPrepareStmt한 SQL를 확인하여 올바른 id 또는 이름을 사용한다.

ERR-22083: invalid binding column count

Cause	결과셋을 위해 바인딩할 컬럼의 개수가 실제 Prepare된 SQL의 바인딩 개수와 다른 경우 발생한다.
Action	Prepare된 SQL의 결과셋 컬럼 개수에 맞춰 변수를 바인딩한다.

ERR-22084: this operation can be executed by a dictionary handle.

Cause	DICT 인스턴스에서만 수행해야 하는 기능을 사용자 인스턴스에서 수행하는 경우 발생한다.
Action	사용자 인스턴스에 대한 CREATE, DROP, ALTER 동작은 DICT 인스턴스에서 수행한다.

ERR-22085: an instance does not exists

Cause	DICT 인스턴스의 dict_inst 테이블에 존재하지 않는 인스턴스를 삭제(drop) 시도한 경우 발생한다.
Action	인스턴스 세그먼트가 존재 여부 따라 조치한다. 1) 인스턴스 세그먼트가 존재하지 않는 경우, 별도 조치가 필요하지 않다. 2) 인스턴스 세그먼트가 존재하는 경우, 관련 세그먼트들을 직접 삭제한다.

ERR-22086: a index key column must have a data type as (long, char, int, short)

Cause	인덱스 생성 시, 인덱스 키로 사용할 수 없는 데이터 타입의 컬럼을 키로 지정하는 경우 발생한다. 각 테이블은 다음 데이터 타입을 인덱스 키로 지정할 수 있다. 1) Direct 테이블: int, long, short 2) 그 외 테이블: int, long, float, double, short, char
Action	dbmMetaManager에서 **desc 명령어**로 테이블 타입과 컬럼별 데이터 타입을 확인하여 인덱스 키를 설정한다.

ERR-22087: a timeout raised on this operation

Cause	1) 원격 접속 및 replication 기능을 사용 시, 일정 시간 내에 응답을 받지 못한 경우 발생한다. 2) Dequeue 호출에서 사용자가 지정한 Timeout내에 데이터가 없는 경우 발생한다.
Action	1) 네트워크 상태를 체크한다. 2) 원격접속인 경우 dbmListener의 정상 동작 여부를 확인한다. 3) Replication 환경일 경우 dbmReplica의 정상 동작 여부를 확인한다. 4) 응답 TIMEOUT과 관련된 속성을 변경한다.

ERR-22088: this operation not allowed at current-instance

Cause	DICT 인스턴스에서 사용자의 객체에 대한 DDL 작업을 수행하는 경우 발생한다.
Action	사용자 인스턴스로 전환한 후 객체에 대한 작업을 수행한다.

ERR-22089: a total size of columns is too big to create

Cause	객체에 저장 가능한 최대 크기를 초과할 경우 발생한다.
Action	**다음 지원 범위**를 참고하여 테이블 및 인덱스 키를 생성한다. (1) Table record: 1,048,247byte (2) Queue message: 1,048,208byte (3) Index key column 전체 길이 합: 1,024byte

ERR-22090: fail to free a statement variable as transaction not completed

Cause	Deprecated
Action	

ERR-22091: a log-size is too big to write a transaction log

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22092: fail to parse syntax

Cause	SQL 문이 문법에 맞지 않아 파싱에 실패한 경우 발생한다.
Action	매뉴얼을 참고하여 SQL 문을 확인한다.

ERR-22093: a operation via API need a index

Cause	API를 사용하려는 대상 테이블에 인덱스가 없는 경우 발생한다.
Action	대상 테이블에 하나 이상의 인덱스를 생성한 후 API를 시도한다.

ERR-22094: a invalid number or range for sequence

Cause	시퀀스 생성 시, START WITH 에 설정된 값이 MAXVALUE를 초과한 경우 발생한다.
Action	START WITH 값을 MAXVALUE 이하로 지정한다.

ERR-22095: a sequence reached at max-value

Cause	No cycle 시퀀스가 최대값에 도달한 경우에 발생한다.
Action	**ALTER SEQUENCE** 구문을 이용하며 current value를 변경하거나 최대값을 고려하여 시퀀스를 재생성한다.

ERR-22096: a currval of sequence not yet defined (need to call nextval)

Cause	시퀀스 객체에 대해 nextval이 호출하지 않은 상태에서 currval을 호출할 경우 발생한다.
Action	nextval을 먼저 호출한 후 currval을 호출한다.

ERR-22097: not enough buffer size

Cause	Deprecated
Action	

ERR-22098: invalid license

Cause	라이선스 검증에 실패한 경우에 발생한다. 1) DBM_HOME 환경 변수 설정이 잘못되어 라이선스 파일을 찾을 수 없는 경우 2) 라이선스 키가 유효하지 않은 경우
Action	DBM_HOME 환경 변수가 올바르게 설정되었는지 확인하고 라이선스 키가 유효하지 않다면 기술지원팀에 문의한다.

ERR-22099: invalid offset

Cause	Internal error code
Action	해당 에러 발생 시 고객센터에 문의한다.

ERR-22100: too many rows

Cause	Deprecated
Action	

ERR-22101: fail to check dictionary

Cause	Deprecated
Action	

ERR-22102: fail to invoke a thread

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22103: fail to read

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22104: fail to write

Cause	데이터파일, 로그 등에 대한 파일 쓰기를 실패한 경우에 발생한다.
Action	권한 설정 및 디스크 공간을 확인한다.

ERR-22105: a instance not active-mode

Cause	INACTIVE 상태의 인스턴스에서 DDL, DML을 시도하는 경우에 발생한다.
Action	ALTER INSTANCE 구문을 사용하여 인스턴스를 ACTIVE 상태로 변경한다.

ERR-22106: a index key column must have a data type as (long, int, short)

Cause	Direct 테이블에 Index key 로 사용할 수 없는 데이터 타입의 컬럼을 지정하는 경우 발생한다.
Action	int, short, long 타입의 컬럼을 Index Key로 지정한다.

ERR-22107: at first, need to create a index to use a direct table

Cause	Internal error code
Action	해당 에러 발생 시 고객센터에 문의한다.

ERR-22108: fail to prepare a disk logfile

Cause	디스크 로깅모드에서 로그파일을 준비하는 과정에서 오류가 난 경우에 발생한다.
Action	Trace log를 추적하여 원인을 분석하고 조치한다.

ERR-22109: fail to prepare replication

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22110: fail to prepare a table

Cause	테이블 준비 과정에서 실패한 경우 발생한다.
Action	다음 사항을 순차적으로 확인하여 원인을 추적한다. 1) 딕셔너리 테이블(DIC_TABLE)에서 테이블 존재 유무 확인 2) 테이블 세그먼트 존재 유무 확인 3) 테이블 세그먼트 권한 확인 4) 테이블 객체 락 상태 확인 (**dbmDump**)

ERR-22111: no data found

Cause	조건과 일치하는 데이터가 검색되지 않는 경우에 발생한다.
Action	

ERR-22112: a replication-session not connected

Cause	Deprecated
Action	

ERR-22113: a result-set has too many rows to process

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22114: a procedure not found

Cause	Deprecated
Action	

ERR-22115: a procedure already exists

Cause	Deprecated
Action	

ERR-22116: invalid identifier

Cause	Deprecated
Action	

ERR-22117: case not found

Cause	Deprecated
Action	

ERR-22118: a cursor already opened

Cause	Deprecated
Action	

ERR-22119: a cursor not opened

Cause	Deprecated
Action	

ERR-22120: a exception duplicated(Line=%d,Column=%d)

Cause	Deprecated
Action	

ERR-22121: a user exception raised

Cause	Deprecated
Action	

ERR-22122: unhandled exceptions

Cause	Deprecated
Action	

ERR-22123: fail to prepare a object/statement of procedure (Line=%d, Column=%d)

Cause	Deprecated
Action	

ERR-22124: a exit/continue statement is able to be used in loop-statement

Cause	Deprecated
Action	

ERR-22125: fail to execute a procedure statement (Line=%d)

Cause	Deprecated
Action	

ERR-22126: changed index after dbmPrepareStmt

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22127: current thread-id already attached at (Trans=%d)

Cause	Deprecated
Action	

ERR-22128: a count of segment expected too many chunk. (need less than 999)

Cause	테이블 생성 시 확장 가능한 최대 세그먼트 개수가 999개 넘을 경우 발생한다.
Action	테이블 생성 시, 최대값과 확장값을 다음 조건에 맞게 설정한다. [max 값] / [extend 값] < 999

ERR-22129: error get semaphore (id=%ld)

Cause	Deprecated
Action	

ERR-22130: open cursor api already executed

Cause	Deprecated
Action	

ERR-22131: close cursor api already executed

Cause	Deprecated
Action	

ERR-22132: a handle of table re-prepared as ddl executed

Cause	트랜잭션 진행 과정에서 대상 테이블에 DDL이 발생한 경우이다.
Action	해당 트랜잭션은 더 진행할 수 없으므로 프로세스를 종료한다. Goldilocks Lite는 트랜잭션 개시 시점에 DDL 감지는 자동으로 Re-prepare를 수행하여 진행 가능하지만 트랜잭션 중간에 발생한 DDL은 오류로 처리한다. (복구 과정에서 모두 롤백처리 된다.)

ERR-22133: an operation can not be executed because another transaction (transId=%d) has already begun

Cause	Deprecated
Action	

ERR-22134: an operation can not be executed because the previous transaction needs to be committed or rolled back

Cause	트랜잭션이 종료되지 않은 세션에서 DDL을 수행하는 경우에 발생한다.
Action	트랜잭션 Commit 또는 Rollback을 수행한 후 DDL을 수행한다.

ERR-22135: a library already exists

Cause	Deprecated
Action	

ERR-22136: a function not exists

Cause	Deprecated
Action	

ERR-22137: fail to execute a user function (%s:RetCode=%d)

Cause	Deprecated
Action	

ERR-22138: invalid time option

Cause	Deprecated
Action	

ERR-22139: port out of range

Cause	Deprecated
Action	

ERR-22140: client max out of range

Cause	Deprecated
Action	

ERR-22141: process max out of range

Cause	Deprecated
Action	

ERR-22142: process min out of range

Cause	Deprecated
Action	

ERR-22143: process count out of range

Cause	Deprecated
Action	

ERR-22144:

Cause	Deprecated
Action	

ERR-22145: create gsb failed

Cause	Deprecated
Action	

ERR-22146: drop gsb failed

Cause	Deprecated
Action	

ERR-22147: a key value must be a json object or an array

Cause	Deprecated
Action	

ERR-22148: invalid json key-string or valueOrType

Cause	Deprecated
Action	

ERR-22149: a replication already exists

Cause	동일한 이름의 이중화 객체가 이미 존재하는 경우에 발생한다.
Action	dbmMetaManager에서 **DIC_REPL_TABLE** 을 조회하여 이중화 대상 객체 목록을 확인한다.

ERR-22150: a column is not valid as index in direct-table

Cause	Internal Error Code
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22151: a value of unsent_dir property is not matched between anchor-file and property-file

Cause	Deprecated
Action	

ERR-22152: a property(%s) is not found or invalid value

Cause	Deprecated
Action	

ERR-22153: a join table needs an index

Cause	Deprecated
Action	

ERR-22154: DDL is not allowed on a table involved in replication

Cause	이중화 대상 테이블에 DDL이 발생할 경우에 발생한다.
Action	**ALTER REPLICATION 구문**을 사용하여 DDL을 수행하려는 테이블을 이중화 목록에서 제거한 후 DDL을 수행한다.

ERR-22155: an operation can not be executed because some table needs a unique index

Cause	이중화 동작을 위해서는 대상 테이블에 unique 인덱스가 필요하다. 이중화 대상으로 등록을 시도한 테이블에 unique 인덱스가 없는 경우 발생한다.
Action	테이블에 unique index를 생성한 후 재시도한다.

ERR-22156: fail to call ODBC_LIB (Detail:%s)

Cause	Deprecated
Action	

ERR-22157: a dsn not exists

Cause	Deprecated
Action	

ERR-22158: fail to open odbc-library

Cause	Deprecated
Action	

ERR-22159: fail to get a function symbol of mapping ODBC API

Cause	Deprecated
Action	

ERR-22160: a json key is too long

Cause	Deprecated
Action	

ERR-22161: http failed

Cause	Deprecated
Action	

ERR-22162: invalid limit option

Cause	Deprecated
Action	

ERR-22163: an update operation is not allowed on a record with expired time

Cause	Deprecated
Action	

ERR-22164: a creation time of segment is invalid

Cause	현재의 인스턴스 생성 전에 만들어진 세그먼트에 접근할 경우 발생한다.
Action	해당 에러 발생 시 기술지원팀에 문의한다.

ERR-22165: cannot update key column value

Cause	인덱스 키 컬럼의 값을 UPDATE하는 경우 발생한다.
Action	테이블의 인덱스 키 컬럼은 UPDATE할 수 없다.

ERR-22166: invalid store key size

Cause	Store key size를 초과하는 길이의 key가 입력될 경우에 발생한다.
Action	dbmMetaManager에서 **desc 명령어**로 Key 값의 크기를 확인하고, 허용 범위 내의 값으로 지정하여 재시도한다.

ERR-22167: invalid store value size

Cause	Store value size를 초과하는 길이의 value가 입력될 경우에 발생한다.
Action	dbmMetaManager에서 **desc 명령어**로 Value 값의 크기를 확인하고, 허용 범위 내의 값을 사용한다.

ERR-22168: an operation not applicable

Cause	Deprecated
Action	

ERR-22169: invalid password or session need to be authorized

Cause	인스턴스 암호화가 설정된 상태에서 인증되지 않은 세션이 작업을 수행하는 경우 발생한다.
Action	dbmInitHandle 수행 후, dbmAuthorize 함수를 사용하여 인증한다.

ERR-22170: some files exist in DBM_DISK_LOG_DIR

Cause	인스턴스 생성 시점에 기존의 트랜잭션 로그 파일이나 유사한 이름의 파일이 존재할 경우에 발생한다.
Action	디스크 로그가 저장되는 디렉토리에 생성된 파일을 확인한다.

ERR-22171: a lock timeout raised

Cause	지정된 시간 동안 lock을 획득하지 못한 경우 발생한다. (프로퍼티: **DBM_LOCK_TIMEOUT** / 함수: **dbmSetLockTimeout**)
Action	**dbmDump 기능**을 사용하여 Lock을 점유한 세션을 확인하고 조치한다.

ERR-22172: too many columns (max=1024)

Cause	컬럼 개수가 1024개를 초과하는 테이블 생성을 시도한 경우 발생한다.
Action	컬럼 개수가 1024개를 초과하지 않도록 테이블을 생성한다.