GOLDILOCKS LITE 3.1 Manual (ko)

SUNJE SOFT
(주)선재소프트

07217 서울시 영등포구 당산로 171 금강펜테리움IT타워 604호
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

차례 v

1. Getting Started 1
   - 1.1 구조 2
   - 1.2 Quick Start 8
   - 1.3 구문 14
   - 1.4 복구 가이드 91
   - 1.5 Utility 94
   - 1.6 Sizing 106
   - 1.7 Monitoring 108
2. API Reference 115
   - 2.1 API 공통사항 116
   - 2.2 dbmInitHandle 117
   - 2.3 dbmFreeHandle 118
   - 2.4 dbmPrepareTable 119
   - 2.5 dbmPrepareTableHandle 120
   - 2.6 dbmPrepareStmt 121
   - 2.7 dbmFreeStmt 123
   - 2.8 dbmBindParamById 124
   - 2.9 dbmBindParamByName 128
   - 2.10 dbmBindCol 130
   - 2.11 dbmBindColStruct 132
   - 2.12 dbmExecuteStmt 135
   - 2.13 dbmFetchStmt 137
   - 2.14 dbmFetchStmt2Json 139
   - 2.15 dbmInsertRow 141
   - 2.16 dbmInsertRowExpired 143
   - 2.17 dbmInsert 145
   - 2.18 dbmUpdateRow 147

2.19 dbmUpdate 149
2.20 dbmUpsert 151
2.21 dbmDeleteRow 153
2.22 dbmDelete 155
2.23 dbmBindColumn 157
2.24 dbmClearBind 159
2.25 dbmUpdateRowByCols 160
2.26 dbmSelectRow 161
2.27 dbmSelect 165
2.28 dbmFetch 167
2.29 dbmSelectMax 169
2.30 dbmSelectCount 171
2.31 dbmSelectRowGT 173
2.32 dbmSelectRowLT 175
2.33 dbmFetchNext 177
2.34 dbmFetchNextGT 179
2.35 dbmFetchNextLT 181
2.36 dbmSelectForUpdateRow 183
2.37 dbmInsertArray 185
2.38 dbmUpdateArray 187
2.39 dbmSelectArray 189
2.40 dbmDeleteArray 191
2.41 dbmEnqueue 193
2.42 dbmDequeue 195
2.43 dbmGetCurrVal 197
2.44 dbmGetNextVal 198
2.45 dbmCommit 199
2.46 dbmRollback 200
2.47 dbmRefineSystem 201
2.48 dbmGetRowCount 203
2.49 dbmGetRowSize 204
2.50 dbmGetTableName 205
2.51 dbmGetTableType 206
2.52 dbmSetSplayMode4DML 208
2.53 dbmGetErrorData 209
2.54 dbmGetErrorMsg 211
2.55 dbmGetTableUsage 213
2.56 dbmGetTableUsageByHandle 215
2.57 dbmExtendTable 217

2.58 dbmExistDataInQue 219
2.59 dbmNow 220
2.60 Error Message 222

3. Stored Procedure 231

3.1 개요 232
3.2 PROCEDURE DDL 234
3.3 PROCEDURE Language Elements 237

4. V2_V3 변환 가이드 265

4.1 V3 신규 기능 266
4.2 Package 변경사항 266
4.3 DB Object Conversion Guide 266
4.4 API Conversion Guide 270

5. High Availability (HA) 구성 가이드 275

5.1 High Availability (HA) 구성 가이드 276
5.2 디스크 로깅 방식 277
5.3 REPLICATION 방식 281

6. 연동가이드 283

6.1 연동가이드 283
6.2 ODBC 설정 285
6.3 JDBC 설정 287
6.4 PYTHON3 연동가이드 288
6.5 JDBC 연동가이드 290
6.6 DBEAVER 연동가이드 292
6.7 Grafana 연동가이드 297

1.

Getting Started

1.1 구조

GOLDILOCKS LITE는 트랜잭션 기능을 탑재한 고성능의 shared memory 기반 데이터 관리 library로서의 성격을 가지고 있다.

GOLDILOCKS LITE의 특징은 다음과 같다.

- 별도의 관리 프로세스가 존재하지 않는 C/C++ library 형태의 direct attached 방식 데이터 관리
- Shared memory 기반의 저장 공간 활용
- Shared memory 공간 자동 확장과 최대 크기 제한 가능
- 하드웨어 장애 복구 및 안정성을 위해 disk logging 기법과 복구시간 단축을 위한 체크포인트 기능 제공
- C 언어 기반의 직관적인 User-Interface API 제공
- 관리자를 위한 SQL syntax 제공
- B+ tree 기반의 탐색 기반 제공
- 동시성 제어를 위한 row level locking 제공
- Automatic deadlock detection 기능 제공

C 언어를 기준으로 설명하면, 다음과 같은 사용자 정의 struct 변수를 그대로 저장/ 조회/ 변경할 수 있도록 직관적인 동작 구조를 가지고 있다.

- 다음은 예제이다.

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

- 위와 같은 C 구조체를 다음과 같은 테이블로 만들 수 있다.

```
create table node
(
    empNo       int,
    empName     char(20),
    deptNo      int,
    birth       date
)
```

- 또한 쉬운 탐색을 위해 (empNo)에 index를 생성한다.

```
create unique index idx_node on node (empNo)
```

사용자는 application source code 내에서 다음과 같이 조작할 수 있다.

```
struct node xData;
...
dbmInsertRow( Handle, "Node", &xData, sizeof(struct Node));
...
dbmSelectRow( Handle, "Node", &xData );
...
```

GOLDILOCKS LITE는 위와 같은 C 언어의 structured data를 그대로 저장/ 조회하기 위해 제공되는 라이브러리이다. 다만, 이 데이터를 빠르게 탐색하기 위해 사용자가 구조체를 대신할 table/ index만 생성하고 그 외는 제공되는 API를 통해 데이터를 관리하는 내장 library로 정의할 수 있다.

Object

GOLDILOCKS LITE는 system 용도의 dictionary 및 사용자 명령에 의한 instance, table, index, queue, sequence 형태의 object를 생성할 수 있다. 각 object들은 트랜잭션을 처리하고 데이터를 저장하기 위한 물리적 공간을 갖는 단위 저장 공간이다.

모든 object 이름의 크기는 32 byte 이내로 제한된다.

DICTIONARY

Dictionary는 내부적으로 object의 정보를 관리하기 위한 공간으로 활용된다.
자세한 내용은 **DICTIONARY**를 참조한다.

INSTANCE

Instance는 다음과 같은 세 가지 저장 공간으로 사용된다. 한 개의 system 안에 N 개의 instance를 생성할 수 있는데 한 개의 instance는 다시 N 개의 하위 테이블을 가질 수 있는 구조로 동작한다.

저장 속성	설명
Session status	Instance 에 접근하는 모든 세션의 PID 등의 정보를 저장하는 공간이다.
Transaction status	Session이 발생시키는 모든 트랜잭션의 상태 정보를 저장하는 공간이다.
Rollback image	트랜잭션에 의해 변경되기 전의 record image를 저장하는 공간이다.

- 동시에 접속할 수 있는 세션의 최대 개수는 1024 개이다.
- Instance는 그 특성상 rollback image를 저장하기 때문에 공간이 부족해질 수 있는데 이를 방지하기 위해 사용자가 1 M 단위로 instance 자동 확장 크기나 최대 크기를 지정하여 생성할 수 있다.

TABLE

- 테이블 이름은 한 개의 instance 안에서 고유해야 하며 테이블의 생성 개수에는 제한이 없다.
- 사용자의 데이터가 저장되는 공간이다.
- 테이블 공간은 생성 시점의 옵션에 따라 레코드 개수 단위로 자동 확장할 수 있고 최대 개수도 지정할 수 있다.
- 저장 가능한 레코드의 최대 크기는 999K 이다.

GOLDILOCKS LITE의 TABLE 개념은 C 기반의 structured data를 저장하기 위한 공간으로서 이는 다른 DBMS의 개념과 다르다. 즉, 사용자가 저장하는 C struct 변수를 테이블 형태로 저장하는 개념이다.

GOLDILOCK LITE는 다음과 같은 유형의 테이블들을 지원한다.

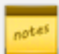
테이블 유형	설명
Normal table	B tree indexing을 가지는 테이블 형태이다.
Direct table	한 개의 column value 기반의 array table 형태이다.
Splay table	Splay index로 구성되는 테이블 형태이다.
Queue table	B tree 기반의 FIFO 방식 테이블이다.
Sequence	Sequence object 이다. (Atomic 보장, non-transactional)

Column 타입으로 지정할 수 있는 데이터 유형은 다음 표와 같다.

Column type	설명
int	sizeof(int)
short	sizeof(short)
float	sizeof(float) (별도의 정밀도를 제공하지 않고 C type과 동일한 형태로 In/out)
long	sizeof(long long)
char (size)	Size
double	sizeof(double) (별도의 정밀도를 제공하지 않고 C type과 동일한 형태로 In/out)
date	sizeof(unsigned long) 8 bytes
USER_TYPE	USER TYPE으로 생성한 타입을 column으로 지정

INDEX

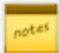
데이터는 테이블에 순차적으로 저장되지만 사용자가 검색에 필요한 index를 생성할 수도 있다.
Index는 B tree를 기반으로 구축되며 별도로 크기를 지정하는 것이 아니라 index가 속한 테이블 크기에 따라 계산된 크기로 생성된다.


Index key column으로 지정할 수 있는 데이터 유형은 short, int, long, char type이며 한 개 이상의 com posite key를 구성하거나 동일한 테이블에 대해 N개의 index를 생성할 수 있다.

DIRECT TABLE

사용자의 특정 column 한 개를 key로 지정하여 그 값을 테이블의 레코드 위치로 변환하여 저장한다. 즉, 일종의 arra y 타입 저장구조를 가지고 있다.
Key로 지정할 수 있는 column 타입은 long, short, int 뿐이다.
만일 key 값이 테이블의 전체 개수보다 클 경우에는 modular 연산을 통해 나머지 값을 key로 지정하고 그 위치에 해당하는 테이블스페이스에 저장한다.

예를 들어, 최대 크기를 100 만개로 지정하고 key 값을 1049로 입력하면 1049 번째 tablespace 위치에 사용자 데이터가 저장된다. 또는 값이 1049000 일 경우, modular 연산을 통해 나온 나머지인 49000 번째 위치에 저장된다. 이 때, 해당 위치에 어떤 값이 이미 저장되어 있는 상태일 경우 duplicated error가 발생한다.


Direct table은 한 개의 key만 지정할 수 있기 때문에 index는 반드시 unique 타입으로 생성해야하며 테이블 하나당 한 개의 unique index만 생성할 수 있다.

SPLAY TABLE

Splay table은 key를 기준으로 별도의 index segment 없이 table segment 내에서 splay indexing으로 정렬되는 테이블이다.
해당 테이블은 최근 조회/ 조작한 데이터 근처의 데이터를 빠르게 탐색해야 하는 업무에 특화된 테이블 기능이다.

QUEUE

사용자의 데이터를 First-In/First-Out (FIFO) 방식으로 저장하고 조회하는 저장소 역할을 수행한다.
Queue table은 B tree 기반의 index를 자동으로 설정하여 MsgType나 priority 등을 사용자가 조작할 수 있도록 지원한다.

Dequeue 동작은 트랜잭션 개념에서 차이가 있다. 예를 들어, 데이터가 (1, 2, 3) 순으로 queue에 저장되어 있을 때 A 세션이 1을 dequeue 하고 아직 커밋하기 전이라도 다른 B 세션은 2라는 데이터를 dequeue 할 수 있다. 즉, A가 커밋할 때까지 기다리지 않는다. Dequeue 동작은 delete와 달리 어떠한 세션에서도 접근하지 않은 데이터를 가져오는 방식으로 동작한다.

동일한 테이블의 다양한 MsgType과 priority를 지원하지만 경합이 빈번할 경우 성능이 저하될 수 있다. 따라서 가능한 응용 프로그램에서 N개의 다양한 queue table을 생성하여 목적에 맞게 분산하여 사용할 것을 권장한다.

SEQUENCE

사용자가 sequence 객체로 지정한 start 값부터 increment 값에 지정된 크기만큼 atomic 하게 증가된 값을 반환한다.

Sequence 객체는 트랜잭션을 지원하지 않기 때문에 standby로 전환될 경우, 각 sequence 객체의 current value를 변경해야 데이터 중복 등을 방지할 수 있다.

동시성 제어

Row Level Locking

GOLDILOCKS LITE는 row level locking을 제공한다. 즉, 최소 lock 대기 지점이 항상 row level에서 발생한다. 따라서 동일한 테이블에서 다수의 session이 서로 다른 row에 접근하더라도 배타적으로 상호 보호할 수 있다.
또한, dead lock을 auto detection 하기 때문에 교착 상태에 빠지는 것을 방지한다.

Undo Space

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

Auto Dead Lock Detection

갱신 연산끼리 서로 대기하는 상황이 발생할 수 있다. 예를 들어 T1 (A, B) 레코드가 존재할 때 session 1은 A를 갱신한 후에 B를 갱신하려고 시도하는 반면에 session 2는 B를 갱신한 후에 A를 변경하려고 시도할 경우, session 1과 session 2는 서로 교착상태에 빠지게 된다.
이런 상황을 자동으로 감지하고 session ID가 큰 쪽에서 오류가 발생하도록 하여 교착 상태를 해제한다.

Delayed Recovery Concept

Delayed recovery는 GOLDILOCKS LITE에 존재하는 실시간 비정상 종료 트랜잭션에 대한 복구 매커니즘이다.

GOLDILOCKS LITE의 in-memory mode 환경에는 별도의 daemon process가 존재하지 않는다. 따라서 사용자 응용 프로그램이 트랜잭션을 수행하는 도중에 commit/ rollback 과정 없이 비정상적으로 종료될 경우에 이를 감지하고 복구하는 일련의 과정을 수행해야 하는데 이를 delayed recovery 라고 한다.

Delayed recovery는 다음과 같이 수행된다.

- 모든 세션은 접속 과정에서 instance transaction 공간에 자신의 고유공간을 할당받는다
- 여기에는 자신의 PID, TID와 같은 process 정보와 해당 공간이 점유되었음을 알리는 정보를 남긴다.
- 모든 세션은 record에 lock을 점유할 때 자신의 고유 ID (Transaction ID)를 기록한다.

이후 트랜잭션 과정에서 lock을 점유해야 할 경우, 나중에 진입한 세션은 위의 방식을 통해 다음의 두 가지 정보에 접근할 수 있다.

- Lock을 점유한 세션 (transaction ID)
- Instance transaction 공간을 통해 lock을 점유한 세션이 alive 한지 여부

위의 방식을 통해 비정상 종료가 감지된 세션은 다음 과정을 통해 복구된다.

- 비정상 종료된 세션의 instance transaction 공간에 대한 lock (복구 작업이 동시에 개시되는 것을 방지)
- 비정상 종료된 세션이 기록한 transaction log를 바탕으로 데이터 복구, lock 해제, 자원 해제
- Instance transaction 공간 해제 및 재사용 공간으로 전환
- 자신의 트랜잭션 개시

복구가 불가능한 경우 사용자에게 에러 메세지를 보여주고 대상 테이블을 refine 처리하여 복구하는 방안을 제공한다.

1.2 Quick Start

GOLDILOCKS LITE의 설치와 간단한 사용법에 대해 기술한다.

설치 전 작업

- Posix 방식의 shared memory를 관리하므로 /dev/shm에 충분한 공간을 설정해야 한다.
- Shared memory 커널 파라미터를 설정해야 한다.
- Semaphore 커널 파라미터를 설정해야 한다.

Shared Memory 커널 파라미터

```
[lim272@localhost test]$ ipcs -lm
------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398442373116
min seg size (bytes) = 1
```

다음과 같이 커널 속성을 변경한 후에 reboot 또는 sysctl -p 하여 반영할 수 있다.

```
[lim272@localhost test]$ vi /etc/sysctl.conf
########################################
kernel.shmmax = (실제 메모리 크기: byte 단위)
kernel.shmall = (실제 메모리 크기: byte 단위)
kernel.shmmni = (segment의 총 개수)
########################################
```

하나의 프로세스 내에서 multi threads로 많은 세션을 가져가는 구조에서는 vm.map_max_count가 65536이면 부족하기 때문에 필요할 경우 262144 정도로 늘려야 한다.

기타

Redhat, Ubuntu와 같은 리눅스 기반 시스템의 systemd가 관리하는 세션 목록에서 IPC 자원 생성 사용자가 로그아웃하면 해당 IPC 자원이 자동 삭제되는 옵션이 추가되었다. 따라서 다음과 같이 시스템을 설정해 주어야 세마포어가 삭제되는 것을 방지할 수 있다. (커널 3.0.0 이상)

```
# cp -i /etc/systemd/logind.conf /etc/systemd/logind.conf_prev
# cat /etc/systemd/logind.conf
[Login]
#NAutoVTs=6
#ReserveVT=6
...
RemoveIPC=no
```

- 수정한 후에 수행한다.

```
# systemctl restart systemd-login
```

환경 변수

GOLDILOCKS LITE를 사용하려면 사용자가 $DBM_HOME/conf/dbm.cfg를 설정하거나 터미널 환경변수를 설정해야 한다. 환경 변수는 각각 다음과 같이 설정한다.

환경변수 이름	설명
DBM_HOME	DBM이 설치된 경로이다.
DBM_INSTANCE	사용할 Default Instance Name을 지정한다.
PATH	$DBM_HOME/bin 을 추가한다.
LD_LIBRARY_PATH	$DBM_HOME/lib 를 추가한다.
DBM_DISK_LOG_ENABLE	DISK mode를 사용할 경우 1로 설정한다.
DBM_DISK_LOG_DIR	DISK mode를 사용할 경우 logfile이 저장될 경로를 지정한다.
DBM_DISK_LOG_FILE_SIZE	기본 100 M로 설정되며 로그 파일의 크기를 지정한다.
DBM_DISK_DATA_FILE_DIR	DISK mode에서 CheckPoint를 수행할 때 생성될 데이터 파일이 저장될 경로를 지정한다.
DBM_LISTEN_PORT	dbmListener를 이용하여 원격으로 접속할 때 사용할 port를 지정한다.
DBM_LISTEN_CONN_TIMEOUT	원격 접속 연결 시점의 timeout 시간을 설정한다. (millisecond 단위)
DBM_LISTEN_RECV_TIMEOUT	원격 접속 전송 후에 ack 수신을 기다리는 시간을 설정한다. (millisecond 단위)
DBM_LOG_CACHE_MODE	Log cache 모드를 설정한다. • 0: 사용 안함 • 1: NVDIMM 사용 • 2: Shared memory 사용
DBM_LOG_CACHE_SIZE	Log cache 크기를 지정한다. (하나의 NVDIMM 또는 shared memory 크기) DBM_DISK_LOG_FILE_SIZE보다 두 배 이상 커야 한다. (Log cache에 DBM_DISK_LOG_FILE_SIZE 이상의 데이터가 있을 때 파일로 저장됨)
DBM_LOG_CACHE_FLUSH_INTERVAL	Log cache가 log file로 flush 되는 interval 이다. (millisecond 단위) (Log cache가 log file 크기에 도달하면 flush 되는데 그 크기가 되기 전이라도 여기에 설정된 interval이 지나면 flush 된다.)

환경변수 이름	설명
DBM_LOG_CACHE_EMPTY_INTERVAL	Log cache를 skip 하는 interval 이다. (millisecond 단위) (Log cache를 사용하는 도중에 죽은 경우를 interval로 판단하여 log를 skip 한다.)
DBM_DISK_COMMIT_WAIT	Commit 할 때 redo 파일이 손실되지 않도록 파일에 완전히 쓰여질 때까지 기다리는 모드를 설정한다.
DBM_ARCHIVE_ENABLE	Redo 로그를 archive로 저장하도록 설정한다.
DBM_ARCHIVE_PATH	Redo 로그를 archive로 저장하는 디렉토리를 설정한다.
DBM_REPL_ENABLE	이중화 동작 여부를 설정한다. (1: 이중화 모드)
DBM_REPL_ASYNC_DML	Async 이중화 동작 여부를 설정한다 (1: async)
DBM_REPL_TARGET_PRIMARY_IP	Sender 측에서 slave에 대한 IP를 설정한다. (예: 127.0.0.1)
DBM_REPL_TARGET_PORT	Sender 측에서 slave의 dbmReplica의 listen port를 설정한다.
DBM_REPL_LISTEN_PORT	Slave 측 dbmReplica의 listen port를 설정한다.
DBM_REPL_CONN_TIMEOUT	세션을 초기 구동하면서 이중화 연결을 시도할 때 timeout을 지정한다. (단위: millisecond)
DBM_REPL_RECV_TIMEOUT	이중화 동작 중에 여기 설정된 시간 동안 Recv packet이 수신되지 않으면 오류로 처리한다. (단위: millisecond)
DBM_REPL_UNSENT_LOG_DIR	이중화 라인이 단절될 경우 미전송 로그를 쌓을 디렉토리를 지정한다.
DBM_REPL_UNSENT_LOGFILE_SIZE	미전송 로그를 쌓을 경우 1개 파일의 최대 크기를 지정한다.
DBM_ODBC_DRIVER_PATH	load/ sync 구문을 사용하기 위해 설치한 unix odbc driver manager의 library 경로를 지정한다. (이 환경변수가 설정되지 않으면 load/ sync 구문이 작동하지 않는다.)
DBM_SHM_PREFIX	이 속성에 설정된대로 /dev/shm의 하위 디렉토리에 shared memory segment를 생성할 경우 그 이름을 고유하게 설정한다.
DBM_SHM_DIR	/dev/shm 하위에 디렉토리를 생성할지 여부이다. 최신 linux 커널은 디렉토리를 생성할 수 없다.

DBM_LOG_CACHE 설정, DIRECTORY 경로 설정, FILE-SIZE와 관련된 사항은 create instance 작업이 수행된 이후에는 프로퍼티를 통해 변경할 수 없으므로 신중하게 설정해야 한다.

다음은 bash 환경에서 환경 변수를 설정하는 예이다.

- GOLDILOCKS LITE 기본 설정

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

- DBM DISK LOG

```
#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 설치 방법과 라이센스 발급에 관한 내용을 기술한다.

설치

설치파일은 보통 다음과 같은 압축파일로 제공되기 때문에 사용자가 설정한 $DBM_HOME 경로에서 압축을 해제하면 설치가 완료된다.

```
goldilocks_A.B.C.tar.gz
<A> : Major Version
<B,C> : Patch Version
```

```
shell> gzip -dc goldilocks_3.1.1.tar.gz | tar xvf -
```

압축을 해제한 후 설치된 각 경로에 대한 설명은 다음과 같다.

경로 이름	설명
arch	Archive log가 저장되는 경로이다.
bin	dbmMetaManager를 포함한 각종 유틸리티 바이너리의 집합이다.
lib	API shared library 위치이다.
include	API header 파일 위치이다.
trc	Trace log가 저장되는 경로이다.
sample	API를 이용한 demo code 이다.
conf	dbm.cfg와 dbm.license 파일이 위치하는 곳이다.
dbf	Data file이 저장되는 경로이다.
repl	not used (reserved)
wal	Redo log가 저장되는 경로이다.

bin 디렉토리의 각 binary는 각각 다음과 같은 기능을 수행한다.

Binary 이름	설명
dbmMetaManager	DDL/ DML을 수행하는 interactive tool 이다.
dbmListener	원격 접속을 가능하게 하는 데몬 tool 이다.
dbmLogFlusher	LogCache를 사용하는 경우 LogCache를 파일로 flush하는 tool이다.
dbmMonitor	현재 DB 상태를 모니터링 하는 tool 이다.

Binary 이름	설명
dbmExp	DDL script와 데이터를 추출하는 tool 이다.
dbmImp	저장된 파일의 데이터를 구분자를 가진 plain text로 load하는 tool 이다.
dbmErrorMsg	Error code를 출력하는 tool 이다.
dbmCkpt	DISK mode에서 logfile을 이용하여 데이터 파일을 생성하고 불필요한 logfile을 삭제하는 tool 이다.
dbmReplica	이중화 운영모드인 경우 Slave에서 데이터를 수신/처리하는 프로세스이다.
dbmDump	세그먼트와 파일을 추적하는 tool이다. 트랜잭션, 테이블, 인덱스, 데이터 파일, 로그 파일, anchor 파일 등의 이상 유무를 추적한다.

라이센스

별도의 테스트 라이센스를 포함하고 있지 않기 때문에 technet@sunjesoft.com 메일로 테스트 장비의 hostid를 기록하여 라이센스를 요청해야 한다.

시작하기

설치가 완료되고 환경변수 설정도 끝났다면 dbmMetaManager를 구동하여 사용을 시작할 수 있다.
다음과 같이 dictionary instance를 생성하여 사용을 준비한다.

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

다음과 같이 사용자 instance를 생성한다. Dictionary instance에 접속한 상태에서만 instance를 생성할 수 있다. Dictionary instance 이름은 dict로 고정되어 있다.

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

다음과 같이 사용자의 table과 index를 생성하여 실제로 데이터를 조작할 수 있는 상태로 만든다.

```
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
```

1.3 구문

API와 dbmMetaManager에서 사용할 각 구문들을 기술한다.

DDL

Instance, table, index 등과 같은 각 object 들을 생성하고 제거하는 구문들이다.
DDL에 사용되는 각 object의 이름의 크기는 최대 32 byte로 제한되며 반드시 문자로 시작해야 한다.

Table (또는 direct table)을 생성할 때 column의 최대 크기는 1048332 byte이다.
Window type table을 생성할 때 data의 최대 크기는 1048328 byte이다.
Queue를 생성할 때 message의 최대 크기는 1048296 byte이다.
Index를 생성할 때 column의 최대 크기는 1024 byte 이다.

DDL이 수행될 때면 항상 instance lock이 잡히는데 이 때문에 이 시점에 시작되는 트랜잭션들은 대기하게 된다. 따라서 모든 세션들을 종료한 후에 DDL을 수행할 것을 권장한다.
DDL을 수행할 때 이전에 수행했던 트랜잭션이 존재할 경우, 오류가 발생하므로 이전에 수행한 트랜잭션을 commit/ rollback 한 후에 실행해야 한다.

initdb

기능

최초로 설치한 후에 dictionary instance를 생성하기 위해 수행한다. 수행하지 않을 경우 사용할 수 없다.
생성된 dictionary instance의 이름은 DICT 이고 set instance 구문을 통해 접근할 수 있다.

구문

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

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(unknown)> initdb;
success
```

```
dbmMetaManager(unknown)>
```

create instance

기능

사용자가 session/ undo space로 사용할 instance object를 생성한다.
Dictionary instance (DICT)에 접속한 상태에서만 instance를 생성하거나 제거할 수 있다.

구문

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

- instance_name: 사용자 지정 이름
- init <size>: 최초로 할당할 공간 크기를 지정 (한 개당 size 단위는 32K 이다.)
- extend <size>: init 공간이 모두 사용되어 max까지 확장될 때의 크기
- max <size>: 최대로 확장할 수 있는 instance 크기

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
```

create table

기능

사용자가 데이터를 저장할 테이블을 생성한다. 테이블은 instance의 하위 개념이기 때문에 생성될 테이블의 instanc e에 먼저 접근해야 한다.
Dictionary instance에는 사용자가 테이블을 생성할 수 없다.

구문

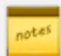
```
<create table> ::= CREATE [TABLE_TYPE] table_name [ SIZE int_val ]
                     (
                        column_definition [, ...]
                     )
                     [ init <size> ]
                     [ extend <size> ]
                     [ max <size> ]
                     ;
```

- TABLE_TYPE ::= TABLE
  | DIRECT
  | SPAY
- table_name: 사용자 지정 테이블 이름이다.
- SIZE int_val ::= WINDOW TYPE 테이블을 생성할 때만 최대 저장길이를 명시한다.
- column_definition ::= column_name data_type_definition
- column_name: 사용자 지정 column 이름이다.
- data_type_definition ::= short
  | int
  | long
  | float
  | double
  | char (size)
  | date
  | USER_TYPE <TypeName>
- init <size>: 최초로 할당될 공간 크기를 지정한다. (Size 단위 하나당 row 하나를 갖는다.)
- extend <size>: init 공간이 모두 사용되어 max까지 확장될 때의 크기이다.
- max <size>: 최대로 확장할 수 있는 row 개수이다.
- USER_TYPE <TypeName> : TypeName은 미리 생성한 사용자 지정 타입의 이름이다.

GOLDILOCKS LITE의 data type은 C 언어의 변수 개념과 유사하다. 다음 크기를 고려하여 사용자 정의 변수를 사용할 경우, offset 위치를 동일하게 사용할 수 있다.

Data type	C type과 크기 (64 bit OS 기준)
short	short과 같으며 2 byte 이다.
int	int와 같으며 4 byte 이다.
long	long을 의미하며 8 byte 이다.
float	4 byte 이다.
double	8 byte 이다.
char	char와 같으며 사용자가 지정한 크기만큼 설정된다. (실제 크기는 c의 struct 구조와 동일하게 할당된다.)

Data type	C type과 크기 (64 bit OS 기준)
date	8 byte이며 사용자 구조체의 멤버 변수는 unsigned long으로 선언해야 한다.
USER_TYPE	사용자가 생성한 타입으로 column 타입을 지정

 각 column의 offset은 c 언어의 struct와 동일하게 alignment 된다.

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
```

다음은 사용자 타입을 지정하는 예이다.

```
dbmMetaManager(DEMO)> CREATE USER_TYPE u2 ( c1 INT, c2 LONG );
success
dbmMetaManager(DEMO)> CREATE TABKE t2 (c1 LONG, c2 INT, c3 USER_TYPE u2 )
success
dbmMetaManager(DEMO)> DESC t2
--------------------------------------------------------------------------------
Instance=(DEMO) Table=(T2) Type=(TABLE) RowSize=(32) LockMode(1)
--------------------------------------------------------------------------------
C1                              long                8           0
C2                              int                 4           8
C3                              U2                  16          16
--------------------------------------------------------------------------------
Any index not created
--------------------------------------------------------------------------------
success
```

create index

기능

사용자가 지정한 테이블에 속한 B-tree index를 생성한다.
최대로 생성할 수 있는 index의 개수에는 제한이 없지만 index key column이 중복될 경우에는 생성할 수 없고 key column size의 합이 1024 byte를 넘는 경우에도 생성할 수 없다.
Index key column으로는 int, short, long, char 네 가지 데이터 타입 column만 허용된다.

구문

```
<create index> ::= CREATE [UNIQUE] INDEX index_name ON table_name
                   ( column_name [, ... ] )
                   ;
```

- index_name: 사용자 지정 index 이름
- table_name: Index를 생성할 대상 table 이름
- column_name: Key column으로 사용될 column 이름

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)
IDX2_T1                                    (C1, C2)
--------------------------------------------------------------------------------------
success
```

USER_TYPE으로 지정된 column은 index column으로 지정할 수 없다.

create queue

기능

First-In/ First-Out (FIFO) 동작을 하는 queue 형태 테이블을 생성한다.

구문

```
<create queue> ::= CREATE QUEUE queue_name SIZE msg_size
                      [ init <size> ]
                      [ extend <size> ]
                      [ max <size> ]
                   ;
```

- queue_name: 대상 queue table 이름
- msg_size: 테이블에 저장될 메시지의 최대 크기
- init <size>: 최초로 할당될 공간 크기를 지정한다. (Size 단위 하나당 row 하나를 갖는다.)

- extend <size>: init 공간이 모두 사용되어 max까지 확장될 때의 크기이다
- max <size>: 최대로 확장할 수 있는 row의 개수이다.

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)
-------------------------------------------------------------------------------------
MSG_TYPE                         int                  4          0
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     (MSG_TYPE, PRIORITY, ID)
-------------------------------------------------------------------------------------
success
```

- Enqueue로 삽입된 데이터는 커밋된 이후 조회하거나 dequeue 할 수 있고 enqueue한 세션 역시 commit 한 후에 dequeue 할 수 있다.
- Dequeue로 한 건을 가져온 이후 commit 하지 않는 세션이 존재하더라도 다른 dequeue를 수행하는 세션이 이를 기다리지 않고 다른 데이터를 dequeue하도록 동작한다.
- Dequeue한 데이터를 rollback 할 경우, 가장 앞쪽으로 다시 재삽입된다.

- Queue를 생성할 때 메시지 크기는 사용자 입력값을 8 byte로 align 하여 생성한다. 따라서 사용자 버퍼를 설정할 때는 테이블 생성 후의 크기를 확인하여 주의해서 할당해야 한다.
- Queue 테이블은 내부적으로 생성 시점에 자동으로 index를 생성하는데 이는 사용자가 변경할 수 없다.

create sequence

기능

Sequence 객체를 생성한다. 다른 DBMS와는 달리 별도로 로깅을 수행하지 않으므로 필요한 경우 사용자가 dbmExp 등을 활용하여 직접 백업해야 한다.
nextval, currval 함수를 제공하므로 해당 값을 이용하여 alter sequence 구문을 통해 변경할 수도 있다.

구문

```
<create sequence> ::= CREATE SEQUENCE sequence_name [options]
                    ;
<options> ::= START WITH <value>
            | INCREMENT BY <value>
            | MAXVALUE <value>
            | MINVALUE <value>
            | CYCLE | NOCYCLE
```

sequence_name: Sequence 이름

- Sequence 생성 시점의 각 옵션들은 생략할 수 있다.
- <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 넘어서면 MinValue로 설정된다.

사용 예

```
***********************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
***********************************************************
dbmMetaManager(DEMO)> create sequence seq1 start with 10 increment by 1 maxvalue 100 cycle;
success
dbmMetaManager(DEMO)> select seq1.nextval from dual;
---------------------------------------------------------------------------------
NEXTVAL              : 11
---------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select seq1.currval from dual;
---------------------------------------------------------------------------------
CURRVAL              : 11
---------------------------------------------------------------------------------
1 row selected
```

Sequence는 로깅 및 이중화 대상이 아니므로 복구 후에 사용자가 해당 값을 새롭게 세팅해야 한다.

drop index

기능

지정된 index를 제거한다.

구문

```
<drop index> ::= DROP INDEX index_name
                 ;
```

index_name: 제거 대상 index의 이름

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
```

Direct type과 queue table에 생성된 index는 사용자가 삭제할 수 없으며 삭제하려고 할 경우 다음과 같은 오류가 발생한다.

```
***********************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
***********************************************************
dbmMetaManager(DEMO)> drop index IDX_QUE1;
Command] <drop index IDX_QUE1>
ERR-102012] fail to prepare a statement
ERR-102047] a operation can not be executed on target-table (check table type)
dbmMetaManager(DEMO)>
```

drop table (queue)

기능

사용자가 지정한 table (queue)과 해당 object에 생성된 하위 index object를 모두 제거한다.

구문

```
<drop table> ::= DROP TABLE table_name
                 |
                 DROP QUEUE table_name
                 ;
```

table_name: 제거 대상 table (queue)의 이름

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)>
```

drop sequence

기능

사용자가 지정한 sequence object를 제거한다.

구문

```
<drop table> ::= DROP SEQUENCE <sequence_name>
                 ;
```

sequence_name: 제거 대상 sequence의 이름

사용 예

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

drop instance

기능

사용자가 지정한 instance와 모든 하위 table, index object를 제거한다.
Dictionary instance (dict)에 접근한 후에 이 구문을 수행해야 한다.

구문

```
<drop instance> ::= DROP INSTANCE instance_name
                  ;
```

instance_name: 제거 대상 instance의 이름

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-0] a handle of dbmMetaManager not initialized
Command] <set instance demo>
```

truncate table (Queue)

기능

사용자가 지정한 테이블을 초기화한다. 테이블 내의 데이터와 extend 된 segment는 모두 제거되고 테이블 생성 시점에 지정된 init size로 재생성된다.

구문

```
<truncate table> ::= TRUNCATE TABLE table_name
                     |
                     TRUNCATE QUEUE table_name
                     ;
```

table_name: Truncate 할 table의 이름

사용 예

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

compact table

기능

사용자가 지정한 테이블에 대해 compaction을 수행한다. 삽입/삭제가 반복되면 늘어난 segment 자체는 OS로 반납할 수 없다. 따라서 이 구문은 segment의 공간을 줄이고자 할 경우에 사용한다.

이 명령은 내부적으로 데이터 export → truncate → 데이터 import 과정을 수행하기 때문에 application들이 접속된 상태에서 동작할 경우 오류가 발생할 수 있으므로 application을 모두 종료한 후 수행해야 한다.

구문

```
<compact table> ::= ALTER TABLE table_name COMPACT
                    ;
```

table_name: Compact 할 table의 이름

사용 예

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

add column

기능

사용자가 테이블에 column을 추가할 경우에 사용한다. 위치를 지정할 수 없으며 항상 마지막 column으로 추가된다. add column 기능은 메모리 모드이면서 일반 테이블 타입인 경우에만 사용할 수 있다.

이 명령은 내부적으로 데이터 export → 테이블 재생성 → 데이터 import 과정을 수행하기 때문에 application들이 접속된 상태에서 동작할 경우 오류가 발생할 수 있으므로 application을 모두 종료한 후에 수행해야 한다.

구문

```
<add column> ::= ALTER TABLE table_name ADD COLUMN ( column_name datatype [default_value] )
                 ;
```

- table_name: Column을 추가할 table의 이름
- column_name: 추가할 column의 이름
- datatype: Column의 데이터 타입 이름 (int, short, long, date, char, float, double)
- default_value: date type을 제외한 데이터 타입의 기본값을 지정할 경우에 정의한다. (상수만 허용)

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
```

- 데이터 건수에 따라 수행 완료까지 소요되는 시간이 길어질 수 있다.
- add column 구문이 동작하는 도중에 오류로 인해 복구가 필요한 경우에는 별도의 dbmMetaManager에 새롭게 접속하면 add column 작업에 대한 복구를 수행하여 원래의 상태로 복구할 수 있다.

  그러나 알 수 없는 원인으로 자동 복구되지 않을 경우에는 add column 수행 시점에 백업된 데이터를 이용해 수동으로 복구할 수 있다.

  백업 데이터는 $DBM_HOME/trc 경로에 table 이름과 SCN 정보를 기반으로 데이터 파일로 생성된다. 이 파일은 dbmImp가 binary 모드로 인식할 수 있는 파일이므로 새롭게 테이블을 생성한 후에 다음과 같은 명령으로 데이터를 복구할 수 있다.

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

drop column

기능

사용자가 테이블에서 column을 제거할 경우에 사용한다. drop column 기능은 메모리 모드이면서 일반 테이블 타입인 경우에만 사용할 수 있고 column이 index key로 사용 중일 경우에는 수행할 수 없다.

이 명령은 내부적으로 데이터 export → 테이블 재생성 → 데이터 import 과정을 수행하기 때문에 application들이 접속된 상태에서 동작할 경우 오류가 발생할 수 있으므로 application을 모두 종료한 후에 수행해야 한다.

구문

```
<drop column> ::= ALTER TABLE table_name DROP COLUMN column_name
                  ;
```

- table_name: Column을 삭제할 table의 이름
- column_name: 삭제할 column의 이름

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
```

- 데이터 건수에 따라 수행 완료까지 소요되는 시간이 길어질 수 있다.
- drop column 구문이 동작하는 도중에 오류로 인해 복구가 필요한 경우에는 별도의 dbmMetaManager에 새롭게 접속하면 drop column 작업에 대한 복구를 수행하여 원래의 상태로 복구할 수 있다.
  그러나 알 수 없는 원인으로 자동 복구되지 않을 경우에는 drop column 수행 시점에 백업된 데이터를 이용해 수동으로 복구할 수 있다.
  백업 데이터는 $DBM_HOME/trc 경로에 table 이름과 SCN 정보를 기반으로 데이터 파일로 생성된다. 이 파일은 dbmImp가 binary 모드로 인식할 수 있는 파일이므로 새롭게 테이블을 생성한 후에 다음과 같은 명령으로 데이터를 복구할 수 있다.

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

rename column

기능

사용자가 테이블에서 column의 이름만 변경할 경우에 사용한다.

이 명령을 사용하면 dictionary의 정보를 교체하는 과정이 발생하기 때문에 application들을 차단한 후에 수행해야 한다.

구문

```
<rename column> ::= ALTER TABLE table_name RENAME COLUMN org_column_name TO new_column_name
                  ;
```

- table_name: Column 이름을 변경할 대상 table의 이름
- org_column_name: 기존 column 이름
- new_column_name: 새로 사용할 column 이름

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
```

create replication

기능

Replication 대상 테이블을 등록한다.

구문

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

사용 예

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

alter replication

기능

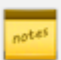
Replication 대상 테이블을 추가/삭제한다.

구문

```
<alter replication> ::= ALTER REPLICATION [ADD¦DROP] TABLE [TableList]
                 ;
<ADD> : 테이블을 이중화 대상으로 추가
<DROP> : 테이블을 이중화 대상에서 제거
<TableList] ::= TableName [, tableName]
```

사용 예

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


위 명령은 dbmPrepareTable을 수행하는 시점에만 유효하기 때문에 현재 동작 중인 이중화에는 영향을 주지 않는다.

alter system replication sync

기능

Master 측에서 다음 목적을 위해 사용한다.
- 미전송 로그를 slave로 전송한다.
- 특정 대상 테이블을 동기화 하기 위해 모두 scan하여 전송한다.

Slave 측에서 다음 목적을 위해 사용한다.
- 미전송 로그를 공유 디스크에 설정한 경우 slave에 mount하여 동기화를 수행한다.

구문

```
<alter system replication> ::= ALTER SYSTEM REPLICATION SYNC
                               [LOCAL | ALL | TableList]
                               ;
<LOCAL> : Slave 측에서 미전송 로그를 직접 읽어 반영 가능한 경우
<ALL> : Master 측에서 모든 이중화 대상 테이블의 데이터를 slave로 전송하고자 할 경우
<TableList] ::= TableName [, tableName]
```

사용 예

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

drop replication

기능

현재 설정된 이중화 대상 테이블을 목록에서 제거한다.

구문

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

사용 예

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

set instance

기능

사용자가 생성한 instance와 dictionary instance로 전환한다.

구문

```
<set instance> ::= SET INSTANCE instance_name
                 ;
```

instance_name: 전환할 instance의 이름

사용 예

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

set instance 구문은 dbmMetaManager에서만 제공되며 소문자 형태로만 동작한다.

alter sequence [currval]

기능

사용자가 생성한 sequence의 current value를 지정된 값으로 변경한다.

구문

```
<alter sequence> ::= ALTER SEQUENCE sequence_name SET CURRVAL = value
                 ;
```

- sequence_name: 대상 sequence의 이름
- value: 사용자가 변경할 current value

사용 예

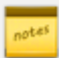
```
***********************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
***********************************************************
dbmMetaManager(DEMO)> select seq1.nextval from dual;
--------------------------------------------------------------------------------
```

```
NEXTVAL              : 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 reset checkpoint

기능

현재 체크포인트를 특정 시점부터 시작하도록 로그 파일 번호를 설정한다.



Archive log를 이용한 복구를 처음부터 시작해야 할 경우에 대비하여 특정 로그 파일 번호부터 재수행하도록 설정한다.

구문

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

사용 예

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

alter system reset perf

기능

모든 통계 누적 정보 (vsys_stat, vsess_stat)를 초기화한다.

구문

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

사용 예

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

통계 정보는 계속 누적되기 때문에 매우 오랜 기간동안 운영하여 표현 범위를 벗어날 경우나 모니터링이 필요한 경우 reset 명령을 사용하여 초기화 할 수 있다.

DML

데이터 추가/ 갱신/ 삭제/ 조회 및 queue table에 대한 enqueue/ dequeue 관련 구문을 설명한다.

insert

기능

사용자가 지정한 테이블에 한 건의 데이터를 추가한다. 구문에 expired 될 시간 (초)을 지정할 경우 해당 데이터는 삽입 시점부터 지정한 시간이 지나면 자동으로 삭제된다. 삽입된 후에 접근이 발생하지 않으면 데이터는 계속 유지되지만 언젠가 full scan 등과 같은 접근이 발생하면 자동으로 제거된다.

Expired time이 설정된 레코드는 별도의 데몬 프로세스에 의해 관리되지 않기 때문에 1초 정도의 오차 범위 내에서 조회될 수 있으며 이는 복구 과정에서도 1초 정도 유지된 후 제거될 수 있다는 것에 주의해야 한다.

구문

```
<insert> ::= INSERT INTO table_name
             [ column_name [, ...] ]
             VALUES
             ( value_expression [, ...] )
             [ EXPIRED expiredTime ]
             ;
```

- table_name: 데이터를 저장할 대상 테이블
- expiredTime: 저장 후 삭제될 시간을 초 단위로 지정 (커밋 시점이 아닌 호출 시점을 기준으로 동작)

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) expired 5;
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
```

- Insert 작업이 수행된 레코드는 commit 전까지는 다른 세션에서 조회되지 않는다. 또한 unique index에 동일한 데이터를 추가하는 다른 세션은 대기한다.
- Direct table에 index를 생성하지 않으면 테이블 저장 위치를 판단할 수 없기 때문에 반드시 index를 미리 생성해야 한다.
- metaManager에서 INSERT를 수행할 때 기술되지 않은 column의 값은 모두 0x00으로 저장된다.

update

기능

사용자가 지정한 테이블에서 한 건 이상의 데이터를 갱신한다.

구문

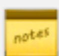
```
<update> ::= UPDATE table_name
             SET column_name = value_expression [, ...]
             [ WHERE cond_expression ]
             ;
```

- table_name: 데이터를 갱신할 대상 테이블
- column_name: 테이블 내의 column 이름
- value_expression: 갱신할 value
- cond_expression: 갱신할 조건절

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
```



update로 갱신된 레코드는 lock으로 다른 세션의 접근을 막는다. select는 update 이전 데이터를 조회하도록 동작한다. 따라서 갱신되고 있는 중에도 갱신 전 형태를 조회할 수 있다.

delete

기능

사용자가 지정한 테이블에서 한 건 이상의 데이터를 삭제한다.

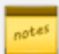
구문

```
<delete> ::= DELETE FROM table_name
             [ WHERE cond_expression ]
             ;
```

- table_name: 데이터를 갱신할 대상 테이블
- cond_expression: 갱신할 조건절

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
```



delete로 삭제할 레코드는 lock으로 다른 세션의 접근을 막는다. select는 update 이전 데이터를 조회하도록 동작한다. 따라서 갱신되고 있는 중에도 갱신 전 형태를 조회할 수 있다.

select

기능

사용자가 지정한 테이블에서 한 건 이상의 데이터를 조회한다.

구문

```
<select> ::= SELECT target_list FROM table_name [alias, table_name alias]
             [ WHERE cond_expression ]
             [ FOR UPDATE ]
             [ LIMIT [ start_offset, ] fetch_limit_count ]
             ;
```

- target_list ::= *
  | column_name [, ...]
- table_name: 데이터를 조회할 대상 테이블 또는 join 할 driving-table, driven-table
- cond_expression: 조회할 조건절
- FOR UPDATE: Select 할 때 lock을 걸어 변경되는 것을 방지해야 할 경우에 사용
- start_offset: 결과 집합에서 출력을 시작할 레코드의 순서를 지정
- fetch_limit_count: 결과 집합에서 출력할 레코드의 개수를 지정

사용 예

```
***********************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
***********************************************************
dbmMetaManager(DEMO)> select * from t1;
--------------------------------------------------------------------------------
C1                    : 1
C2                    : 1
C3                    : abc
--------------------------------------------------------------------------------
C1                    : 2
C2                    : 2
C3                    : def
--------------------------------------------------------------------------------
```

```
2 row selected
dbmMetaManager(DEMO)> select c1, c3 from t1 where c1 = 2;
--------------------------------------------------------------------------------
C1                   : 2
C3                   : def
--------------------------------------------------------------------------------
1 row selected
```

```
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)> insert into t1 values (2, 2)
success
dbmMetaManager(DEMO)> insert into t1 values (3, 3)
success
dbmMetaManager(DEMO)> insert into t1 values (4, 4)
success
dbmMetaManager(DEMO)> insert into t1 values (5, 5)
success
dbmMetaManager(DEMO)> commit
success
dbmMetaManager(DEMO)> create table t2 (c1 int, c2 int)
success
dbmMetaManager(DEMO)> create unique index idx_t2 on t2 (c1)
success
dbmMetaManager(DEMO)> insert into t2 values (1, 11)
success
dbmMetaManager(DEMO)> insert into t2 values (2, 22)
success
dbmMetaManager(DEMO)> insert into t2 values (3, 33)
success
dbmMetaManager(DEMO)> insert into t2 values (4, 44)
success
dbmMetaManager(DEMO)> insert into t2 values (5, 55)
success
dbmMetaManager(DEMO)> commit
success
dbmMetaManager(DEMO)> select * from t1 a, t2 b
```

```
where a.c1 = b.c1
Access Count = 5
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 1
C1                   : 1
C2                   : 11
--------------------------------------------------------------------------------
C1                   : 2
C2                   : 2
C1                   : 2
C2                   : 22
--------------------------------------------------------------------------------
C1                   : 3
C2                   : 3
C1                   : 3
C2                   : 33
--------------------------------------------------------------------------------
C1                   : 4
C2                   : 4
C1                   : 4
C2                   : 44
--------------------------------------------------------------------------------
C1                   : 5
C2                   : 5
C1                   : 5
C2                   : 55
--------------------------------------------------------------------------------
5 row selected
dbmMetaManager(DEMO)> select * from t1 a, t2 b
where a.c1 = 1
and a.c1 = b.c1
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 1
C1                   : 1
C2                   : 11
--------------------------------------------------------------------------------
1 row selected
----------------------------------
-- LIMIT Sample
```

```
----------------------------------
dbmMetaManager(DEMO)> select * from t1 where c1 > 9999 and c1 < 100000 limit 30, 3;
---------------------------------------------------------------------------------
C1                    : 10029
C2                    :
---------------------------------------------------------------------------------
C1                    : 10030
C2                    :
---------------------------------------------------------------------------------
C1                    : 10031
C2                    :
---------------------------------------------------------------------------------
3 row selected
```

- GOLDILOCKS LITE는 다른 DBMS와 달리 첫 번째로 명시된 테이블을 driving-table로 자동 선정하여 join 한다. 세 개 이상의 테이블은 join 할 수 없다.

- Direct 테이블은 지정된 key를 위치 판단용로 사용하기 때문에 일반 테이블과 달리 GT/ LT 연산을 정확하게 수행할 수 없다. 즉, 일종의 ordering 되지 않은 array와 같은 개념이므로 조회할 때 정렬되지 않거나 GT/ LT의 연산 결과가 기대한 것과 다를 수 있다. 따라서 direct 테이블은 key를 기준으로 한 equal 연산으로 빨리 처리해야 할 경우에 사용할 것을 권장한다.

- LIMIT 절은 실제 데이터를 가져오기 전, 최종 결과 집합을 만든 후에 적용되기 때문에 실행 속도를 줄일 수 없다.

GROUP BY 및 ORDER BY는 다음과 같이 지원된다.

```
<select> ::= SELECT target_list FROM table_name [alias, table_name alias]
             [ WHERE cond_expression ]
             [ GROUP BY target_id [, target_id] ]
             [ ORDER BY target_id [, target_id] ]
             ;
```

- target_list ::= *
  | column_name [, ...]
- table_name: 데이터를 조회할 대상 테이블 또는 join 할 driving-table, driven-table
- cond_expression: 조회할 조건절
- target_id: target_list에 기술한 column의 순서를 의미 (base=1)

group by에 의해 기술될 수 있는 target은 column identifier로 제한된다. 또한, aggregation 함수는 min, max, sum, avg, count로 제한된다. 만일, 일반적인 column identifier가 아닌 다른 함수가 기술되면 지원되지 않는다. 다음 예제를 참고한다.

```
dbmMetaManager(DEMO)> create table t1 (c1 int, c2 int, c3 int)
success
dbmMetaManager(DEMO)> insert into t1 values (10, 5, 4)
success
dbmMetaManager(DEMO)> insert into t1 values (10, 4, 4)
success
dbmMetaManager(DEMO)> insert into t1 values (1, 1, 1)
success
dbmMetaManager(DEMO)> insert into t1 values (1, 2, 1)
success
dbmMetaManager(DEMO)> insert into t1 values (2, 3, 1)
success
dbmMetaManager(DEMO)> insert into t1 values (2, 8, 1)
success
dbmMetaManager(DEMO)> insert into t1 values (3, 4, 1)
success
dbmMetaManager(DEMO)> insert into t1 values (3, 5, 1)
success
dbmMetaManager(DEMO)> insert into t1 values (4, 6, 1)
success
dbmMetaManager(DEMO)> insert into t1 values (4, 8, 1)
success
dbmMetaManager(DEMO)> insert into t1 values (4, 9, 1)
success
dbmMetaManager(DEMO)> insert into t1 values (4, 10, 1)
success
dbmMetaManager(DEMO)> commit
success
dbmMetaManager(DEMO)> select c1, sum(c2) from t1 group by 1
--------------------------------------------------------------------------------
C1                    : 1
SUM                   : 3
--------------------------------------------------------------------------------
C1                    : 2
SUM                   : 11
--------------------------------------------------------------------------------
C1                    : 3
```

```
SUM                    : 9
--------------------------------------------------------------------------------
C1                     : 4
SUM                    : 33
--------------------------------------------------------------------------------
C1                     : 10
SUM                    : 9
--------------------------------------------------------------------------------
5 row selected
```

select for update

기능

사용자가 지정한 테이블에서 한 건 이상의 데이터를 조회하고 해당 레코드에 lock을 수행한다.

구문

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

- target_list ::= *
  | column_name [, ...]
- table_name : 데이터를 조회할 대상 테이블
- cond_expression : 조회할 조건절

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> select * from t1 for update;
--------------------------------------------------------------------------------
C1                     : 1
C2                     : 1
C3                     : abc
--------------------------------------------------------------------------------
C1                     : 2
C2                     : 2
```

```
C3                    : def
--------------------------------------------------------------------------------
2 row selected
dbmMetaManager(DEMO)> select c1, c3 from t1 where c1 = 2 for update;
--------------------------------------------------------------------------------
C1                    : 2
C3                    : def
--------------------------------------------------------------------------------
1 row selected
```

- select_for_update에 의해 lock이 걸린 레코드에 접근하는 다른 세션에서는 select가 허용되지만 lock을 걸어야 하는 DML은 commit/ rollback 이전까지 대기한다.
- JOIN 구문에서는 FOR UPDATE 기능을 사용할 수 없다.

enqueue

기능

사용자가 지정한 테이블에 한 건의 데이터를 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.

사용 예

```
************************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
************************************************************
dbmMetaManager(DEMO)> create queue que1 size 100;
success
dbmMetaManager(DEMO)> enqueue into que1 (msg_type, priority, msg_size, message)
                 2 values (1, 90, 10, 'msg1234567');
success
```

```
dbmMetaManager(DEMO)> commit;
success
dbmMetaManager(DEMO)> select * from que1;
---------------------------------------------------------------------------------
MSG_TYPE             : 1
PRIORITY             : 90
ID                   : 1
MSG_SIZE             : 10
IN_TIME              : 2019/01/02 17:37:14.814588
MESSAGE              : msg1234567
---------------------------------------------------------------------------------
1 row selected
```

Target_list에 기술할 수 있는 항목은 msg_type, priority, msg_size, message 이다.
msg_size, message는 필수 항목이며 그 외의 항목들은 모두 0으로 저장된다.

```
dbmMetaManager(DEMO)> enqueue into que1 (msg_size, message) values (10, 'msg7654321');
success
dbmMetaManager(DEMO)> commit;
success
dbmMetaManager(DEMO)> select * from que1;
---------------------------------------------------------------------------------
MSG_TYPE             : 0
PRIORITY             : 0
ID                   : 2
MSG_SIZE             : 10
IN_TIME              : 2019/01/02 17:40:41.569927
MESSAGE              : msg7654321
---------------------------------------------------------------------------------
1 row selected
```

insert 구문을 사용하여 수행할 경우 다음과 같은 오류가 발생한다.

```
dbmMetaManager(DEMO)> insert into que1 (msg_type, priority, msg_size, message) values (1, 90,
10, 'msg1234567');
Command] <insert into que1 (msg_type, priority, msg_size, message) values (1, 90, 10, 'msg
1234567')>
ERR-102007] fail to execute a statement
ERR-102047] a operation can not be executed on target-table (check table type)
```

dequeue

기능

사용자가 지정한 테이블에서 한 건의 데이터를 dequeue 한다.

구문

```
<dequeue> ::= DEQUEUE FROM table_name
              [ WHERE cond_expression ]
              ;
```

- table_name: 대상 테이블이다.
- cond_expression: Dequeue 할 조건절이다.

사용 예

```
***********************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
***********************************************************
dbmMetaManager(DEMO)> select * from que1;
--------------------------------------------------------------------------------
MSG_TYPE             : 1
PRIORITY             : 90
ID                   : 1
MSG_SIZE             : 10
IN_TIME              : 2019/01/02 17:37:14.814588
MESSAGE              : msg1234567
--------------------------------------------------------------------------------
MSG_TYPE             : 0
PRIORITY             : 0
ID                   : 2
MSG_SIZE             : 10
IN_TIME              : 2019/01/02 17:40:41.569927
MESSAGE              : msg7654321
--------------------------------------------------------------------------------
2 row selected
dbmMetaManager(DEMO)> dequeue from que1 where msg_type = 1;
MSG_TYPE             : 1
PRIORITY             : 90
ID                   : 1
MSG_SIZE             : 10
```

```
IN_TIME                 : 2019/01/02 17:37:14.814588
MESSAGE                 : msg1234567
```

별도의 조건절이 없을 경우, msg_type이 0이면서 priority 값이 낮은 레코드들 중에 먼저 enqueue commit 된 레코드부터 dequeue 된다.

DCL

DCL은 사용자가 수행한 각 트랜잭션을 영구적으로 commit/ rollback 하기 위해 제공되는 명령문이다.

commit

기능

사용자가 발생시킨 모든 트랜잭션을 영구적으로 commit 한다.

구문

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

사용 예

```
***********************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
***********************************************************
dbmMetaManager(DEMO)> commit;
success
```

rollback

기능

사용자가 발생시킨 모든 트랜잭션을 원래 상태로 rollback 한다.

구문

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

사용 예

```
***********************************************************
* Copyright © 2010 SUNJESOFT Inc. All rights reserved.
***********************************************************
dbmMetaManager(DEMO)> rollback;
success
```

Built-in Function

- 사용상 편의를 위해 SQL 문 유형을 수행할 때 다음과 같은 built-in function을 제공한다.
- 내장 함수 중에 숫자형 관련 함수들을 bind하는 시점에 타입을 지정하면 해당 타입으로 전환되긴 하지만 내부적으로 반환되는 값은 모두 8 byte double을 채택하기 때문에 사용자도 type 변환 비용이 발생하지 않도록 double 타입으로 바인딩하는 것을 권장한다.
- 반환되는 문자열의 크기는 최대 32 K 이다.
- GOLDILOCKS LITE에서 제공하는 모든 숫자형 함수들은 overflow/ underflow와 관련된 에러를 체크하지 않는다.

Category	Function name	Return value	Desc
날짜/ 시간	sysdate	long (8 bytes)	내부 저장 용도의 8 byte long long 형태의 값으로 저장하고 출력한다.
	extract	int (4 bytes)	날짜형식의 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)	입력된 날짜 타입의 두 개인자 사이의 간격을 초단위로 출력한다.
Dump	dump	char (최대 1Mbyte)	ascii를 출력한다. byte당 2~4 byte로 출력된다.
	hex	char (최대 1Mbyte)	hex를 출력한다. byte당 2 byte로 출력된다.
	concat	char (최대 1Mbyte)	두 번째 인자의 문자열을 첫번째 인자에 붙여서 출력한다.

Category	Function name	Return value	Desc
문자형	instr	int (4 bytes)	소스문자열 내에 지정된 문자열이 존재할 경우 시작 위치를 1로 하여 offset을 출력한다.
	replace	char (최대 1Mbyte)	검색어를 찾아 지정된 문자열로 치환한다.
	substr	char (최대 1Mbyte)	문자열에서 지정된 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)	입력된 수의 제곱을 출력한다.
	sqrt	double (8 bytes)	입력된 수의 제곱근을 출력한다.
	log	double (8 bytes)	base를 기준으로 하는 자연로그 결과를 출력한다.
	exp	double (8 bytes)	e의 제곱을 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 사이의 random 정수를 출력한다.
Sequence	currval	long long (8 bytes)	-
	nextval	long long (8 bytes)	-
단방향 hash 암호화	digest	char (64 bytes)	알고리즘에 따라 반환되는 길이가 다르다.
Aggregation	min	double (8 bytes)	group by를 지원하지 않는다.
	max	double (8 bytes)	
	avg	double (8 bytes)	

Category	Function name	Return value	Desc
	sum	double (8 bytes)	
JSON OBJECT	JSON_STRING	char	일반 테이블 결과를 JSON 형태로 반환한다.
	JSON_KEY	char	create index를 수행할 때 json 내부의 path를 key로 설정한다.
	JSON_VALUE	char	json format string에서 특정 필드를 찾아 반환한다.
	JSON_QUERY	char	json format string에서 특정 필드를 찾아 json format으로 반환한다.
기타	decode	char (최대 1 Mbyte)	결과와 일치하는 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 구하여 date type으로 반환한다.
(단, dbmMetaManager에서 조회할 때는 long long type이 아닌 string 형태로 반환한다.)

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
```

다른 DBMS에서와는 달리 sysdate가 DML에서 날짜를 처리하기 위한 pseudo code처럼 사용된다. 따라서 sysdate가 binary 형태의 데이터로 출력된다. 현재 시각을 확인하려면 CURRENT를 사용해야 한다.

extract

주어진 날짜시간 타입의 데이터에서 입력된 항목의 필드를 숫자형으로 추출한다.
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 string으로 출력한다. 포맷은 YYYY-MM-DD H24:MI:SS.SSSSSS로 고정되어 있다.
인자로는 date type을 사용할 수 있다.

사용 예

```
dbmMetaManager(DEMO)> select datetime_str(sysdate) from dual;
--------------------------------------------------------------------------------
DATETIME_STR         : 2020-08-24 14:23:33.452934
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)>
```

to_date('문자열', 'Format')

사용자 입력 문자열을 일정한 형식의 date 타입 값으로 변환한다. Default 형식은 "YYYY/MM/DD H24:MI:SS.S6" 로 고정되어 있다.

Format 지정은 다음 표를 참조한다.

Format	설명
YYYY	4 자리 연도이다.
MM	월의 단위로서 (01~12) 범위이다.

Format	설명
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 않는 경우에는 사용자의 문자열을 기본 포맷 형태로 일치시켜야 하는데 이 때 연/월/일을 포함해야 하며 그렇지 않은 경우에는 오류가 발생한다.

```
dbmMetaManager(DEMO)> select to_date('2020/12/31 12:40') from dual
---------------------------------------------------------------------------------
TO_DATE               : 2020/12/31 12:40:00.000000
---------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select to_date('12:40') from dual
Command] <select to_date('12:40') from dual>
ERR-70033] fail to execute a statement
ERR-70047] invalid expression type
ERR-70001] fail to validate some parameters at internal processing
```

다음과 같이 DML에 함수를 포함시켜서 사용할 수 있다.

```
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 날짜 타입의 인자 두 개 (from ~ to) 사이의 간격을 초단위로 환산하여 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                 : 365.000000000
--------------------------------------------------------------------------------
1 row selected
```

dump(value, [base])

지정된 문자열 value에 대해 byte 단위로 ascii 값으로 변환한 text를 반환한다.
Base를 별도로 지정하지 않을 경우 default로 10 진수 ascii로 동작하며 16 진수를 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 출력한다.

사용 예

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

concat(target, append)

target 문자열 뒤에 append 문자열을 추가하는 연산을 수행한다. (strcat 연산)

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 처음부터 시작하고 음수인 경우 역순으로 검색한다. 만일 입력값이 0이면 출력값은 keyword에 상관없이 0으로 반환된다.
#appearance는 keyword가 source 내에 출현한 횟수가 일치하는 지점을 탐색한다.

사용 예

```
dbmMetaManager(DEMO)> select instr( 'abcdef abcdef', 'def' ) from dual
---------------------------------------------------------------------------------
INSTR                  : 11
---------------------------------------------------------------------------------
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 문자열 내에 keyword를 찾아 replace 문자열을 변경하는 연산을 수행한다.

사용 예

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

substrt(source, start_position, count)

Source 문자열의 start_position부터 count 만큼 문자열을 잘라 반환한다.

- Count가 생략될 경우 start_position부터 남은 문자열을 모두 반환한다.
- Start_position이 source 문자열 길이를 초과할 경우, 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
```

length(Source)

Source 문자열 길이를 반환한다. 중간에 NULL이 있다면 그 위치까지의 길이만 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select length( ltrim( rpad('  str', 20, 'x') ) ) from dual
---------------------------------------------------------------------------------
LENGTH                : 18
---------------------------------------------------------------------------------
1 row selected
```

ltrim / rtrim(source)

Source 문자열의 왼쪽 또는 오른쪽에 존재하는 공백 문자를 제거하여 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 또는 오른쪽에 사용자가 입력한 padding_string을 size 이내로 추가하여 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)

입력 항목의 절대값을 반환한다.
double의 표현 범위 내에서 사용해야 한다.

사용 예

```
dbmMetaManager(DEMO)> select abs(-123) from dual;
---------------------------------------------------------------------------------
ABS                      : 123.000000000
---------------------------------------------------------------------------------
1 row selected
```

mod(value1, value2)

(value1 / value2) 연산으로 발생한 나머지 값을 반환한다.
double의 표현 범위 내에서 사용해야 한다.

사용 예

```
dbmMetaManager(DEMO)> select mod(14, 3) from dual;
---------------------------------------------------------------------------------
MOD                      : 2.000000000
---------------------------------------------------------------------------------
1 row selected
```

ceil(value)

Value 보다 큰 가장 가까운 정수를 반환한다.
double의 표현 범위 내에서 사용해야 한다.

사용 예

```
dbmMetaManager(DEMO)> select ceil( 12.345) from dual;
---------------------------------------------------------------------------------
CEIL                    : 13.000000000
---------------------------------------------------------------------------------
1 row selected
```

floor(value)

Value 보다 작은 가장 가까운 정수를 반환한다.
double의 표현 범위 내에서 사용해야 한다.

사용 예

```
dbmMetaManager(DEMO)> select floor(13.678) from dual;
---------------------------------------------------------------------------------
FLOOR                   : 13.000000000
---------------------------------------------------------------------------------
1 row selected
```

round(value)

Value를 반올림하여 정수를 반환한다.
double의 표현 범위 내에서 사용해야 한다.

사용 예

```
dbmMetaManager(DEMO)> select round(12.345) from dual;
---------------------------------------------------------------------------------
ROUND                   : 12.000000000
---------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select round(12.678) from dual;
---------------------------------------------------------------------------------
ROUND                   : 13.000000000
```

```
--------------------------------------------------------------------------------
```

trunc(value, [pos])

Value가 소수점일 경우 지정된 pos 위치에서 소수점 이하의 수를 제거한 후에 반환한다.
지정되지 않은 경우 소수점 이하를 모두 버린 후에 반환한다.
double의 표현 범위 내에서 사용해야 한다.

사용 예

```
dbmMetaManager(DEMO)> select trunc(13.34567) from dual;
--------------------------------------------------------------------------------
TRUNC                    : 13.000000000
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select trunc(13.34567, 2) from dual;
--------------------------------------------------------------------------------
TRUNC                    : 13.340000000
--------------------------------------------------------------------------------
```

random(from, to)

From, To로 지정된 범위 내의 random 정수를 반환한다.
double의 표현 범위 내에서 사용해야 한다.

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
```

substr(value, [start], [count])

문자열 value에 대해 start 위치에서 count만큼 문자열을 잘라 반환한다.

- start, count를 입력할 경우 음수로 지정할 수 없다.
- count가 지정되지 않은 경우, start 위치부터 마지막 문자열까지 반환한다.

사용 예

```
dbmMetaManager(DEMO)> select substr( 'abcdefg', 3, 1) from dual;
---------------------------------------------------------------------------------
SUBSTR                    : c
---------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select substr( 'abcdefg', 3) from dual;
---------------------------------------------------------------------------------
SUBSTR                    : cdefg
---------------------------------------------------------------------------------
```

nextval

지정된 sequence의 다음 값을 반환한다.

사용 예

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

currval

지정된 sequence의 현재 값을 반환한다.

사용 예

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

Sequence 객체에 대해 nextval이 호출되지 않았을 때 currval을 호출하면 오류를 반환한다.

Digest (Value, SHA-type)

입력된 문자열 value와 SHA-type로 암호화 된 결과를 출력한다. Digest 처리된 결과는 binary 형태로써 화면에 정상적으로 출력되지 않을 수도 있다. 이 경우, 다음과 같이 hex 함수 등을 통해 확인할 수 있다.

사용 예

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

Min(Column)

Select 된 결과 집합에서 min 함수 내에 정의된 column 값 중 가장 작은 값을 반환한다.
double의 표현 범위 내에서 사용해야 한다.

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 된 결과 집합에서 max 함수 내에 정의된 column 값 중 가장 큰 값을 반환한다.
double의 표현 범위 내에서 사용해야 한다.

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)

Select 된 결과 집합 중에 avg 함수 내에 정의된 column 값의 평균값을 반환한다.
double의 표현 범위 내에서 사용해야 한다.

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)

Select 된 결과 집합에서 sum 함수 내에 정의된 column 값의 합계를 반환한다.
double의 표현 범위 내에서 사용해야 한다.

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_expr이 가진 값과 일치하는 case_cond 다음에 기술된 value를 반환한다. 만일 일치하는 경우가 없을 경우, e lse_expr이 존재하면 해당 값을 반환하며 그렇지 않으면 길이가 0인 값을 반환한다.
cond_expr과 case_cond의 데이터 타입은 동일한데 double 또는 문자열을 사용할 수 있으며 반환되는 value_expr, 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 text일 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 text일 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 테이블에 한해 전체 또는 주어진 column 목록에 해당하는 데이터들만 JSON 형태로 변환된 string을 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, 'pppppppppp1' )
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": "pppppppppp1" }
--------------------------------------------------------------------------------
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": "pppppppppp1" }
--------------------------------------------------------------------------------
3 row selected
```

JSON_KEY(Column_Name, Path, Key_Length)

create index 구문에서 Json의 특정 값을 key로 사용할 때 사용한다.
Path는 json path expression의 규칙을 따르지만 현재는 제한적으로 [](array), *(wildcard)만 사용할 수 있다.

Path는 '$.key1.key2'와 같이 single quote를 사용하여 입력하는데 depth는 dot으로 지정한다. 매칭되는 값이 없으면 NULL이 반환되고 그 외의 오류에 대해서는 에러를 반환한다.
Path는 '$.key1.*', '$.*.key1' 등과 같은 wildcard를 지원한다.
매칭되는 값이 여러 개인 경우 어떤 값을 반환할지 정해지지 않은 상태이므로 이런 식의 사용은 권장하지 않는다. 현재 구조에서는 마지막으로 매칭되는 값을 반환하지만 버전이 바뀔 경우 반환되는 값도 바뀔수 있다.
Path는 '$.key1[1].key2', '$.key1[*].key2' 처럼 array를 사용할 수 있다.

사용 예

```
dbmMetaManager(DEMO)> create table t1 (c1 int, c2 char(400));
success
dbmMetaManager(DEMO)> create unique index idx1_t1 on t1 ( json_key(c2, '$.key1', 32) );
success
```

JSON_VALUE(Column_Name, Path)

테이블에 저장된 Json text에서 path와 일치하는 필드를 찾아 그 값을 char 형태로 반환한다.
Path와 일치하는 필드가 string이 아닌 경우에는 NULL을 반환한다.
Path와 관련된 자세한 내용은 **JSON_KEY(Column_Name, Path, Key_Length)**를 참조한다.

사용 예

```
dbmMetaManager(DEMO)> insert into t1 values( 0,
    '{
        "key1":"100",
        "key2":{"key3":"100"},
        "key4":{"key41":"100", "key42":"100"},
        "ARRAY":[{"ID":"1a"},{"ID":"2aa"},{"ID":"3aa"}]
    }');
1 row selected
dbmMetaManager(DEMO)> select c1, json_value(c2, '$.key1') from t1 where c1 = 0;
--------------------------------------------------------------------------------
C1                   : 0
JSON_VALUE           : 100
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> delete from t1 where json_value(c2, '$.key1') = '100';
1 row deleted.
```

JSON_QUERY(Column_Name, Path)

테이블에 저장된 Json text에서 path와 일치하는 필드를 찾아 그 값을 json 형태로 반환한다.
Path와 일치하는 필드는 string type이 아니더라도 반환된다.
Path에 대한 자세한 내용은 **JSON_KEY(Column_Name, Path, Key_Length)**를 참조한다.

사용 예

```
dbmMetaManager(DEMO)> insert into t1 values( 0,
    '{
        "key1":"100",
        "key2":{"key3":"100"},
        "key4":{"key41":"100", "key42":"100"},
        "ARRAY":[{"ID":"1a"},{"ID":"2aa"},{"ID":"3aa"}]
    }');
success
dbmMetaManager(DEMO)> select c1, json_query(c2, '$.key1') from t1 where json_value(c2, '$.key
```

```
1') = '100';
---------------------------------------------------------------------------------
C1                      : 0
JSON_QUERY              : "100"
---------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select c1, json_query(c2, '$.key2') from t1 where c1 = 0;
---------------------------------------------------------------------------------
C1                      : 0
JSON_QUERY              : { "key3": "100" }
---------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select c1, json_query(c2, '$.ARRAY') from t1 where c1 = 0;
---------------------------------------------------------------------------------
C1                      : 0
JSON_QUERY              : [ { "ID": "1a" }, { "ID": "2aa" }, { "ID": "3aa" } ]
---------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)> select c1, json_query(c2, '$.ARRAY[2]') from t1 where c1 = 0;
---------------------------------------------------------------------------------
C1                      : 0
JSON_QUERY              : { "ID": "3aa" }
---------------------------------------------------------------------------------
1 row selected
```

USER_TYPE(Column_Name, Type_Name)

User type으로 지정된 column을 캐스팅하여 값을 출력한다.

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
```

DICTIONARY

initdb를 통해 dictionary instance가 생성되면 자동으로 사용자 object를 관리하기 위한 dictionary table들이 생성된다. 본 절에서는 각 테이블들에 대해 설명한다.

DIC_INST

사용자가 생성한 instance 정보를 저장한다.

Column name	설명
INST_NAME	Instance name
INIT_SIZE	생성 시 초기 크기 (단위: undo page의 개수)
EXTEND_SIZE	공간 확장 시 크기
MAX_SIZE	최대 확장 가능 크기

DIC_TABLE

사용자가 생성한 table 정보를 저장한다.

Column name	설명
INST_NAME	Instance name
TABLE_NAME	Table name

Column name	설명
TABLE_TYPE	TABLE 유형 • 1: TABLE • 2: QUEUE • 3: SEQUENCE • 4: DIRECT
COLUMN_COUNT	TABLE을 구성하는 column의 개수
ROW_SIZE	TABLE에 저장되는 align된 레코드 크기의 합
MSG_SIZE	QUEUE TABLE인 경우 message의 최대 크기
INDEX_COUNT	현재 테이블에 생성된 INDEX 개수
INIT_SIZE	생성 시 초기 크기 (단위: 레코드 개수)
EXTEND_SIZE	공간 확장 시 크기 (단위: 레코드 개수)
MAX_SIZE	최대 확장 가능 크기 (단위: 레코드 개수)
INDEX_ID	Index 생성 시점마다 부여되는 index 고유 번호 채번

DIC_COLUMN

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 • 14: USER_TYPE
OFFSET	레코드 전체 영역 중에 column이 저장되는 위치
SIZE	Column의 데이터 크기
COLUMN_ORDER	Column 순서

DIC_INDEX

테이블에 생성한 index의 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

Index를 구성하는 key column의 정보를 저장한다.

Column name	설명
INST_NAME	Instance name
TABLE_NAME	Table name
INDEX_NAME	Index name
COLUMN_NAME	Column name
KEY_COLUMN_ORDER	Key column의 나열 순서 (Ordering 순서를 의미한다.)
COLUMN_ORDER	테이블 내의 column 순서

DIC_SEQUENCE

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

Information View

GOLDILOCKS LITE의 상태 정보를 보여주기 위해 RDBMS 제품들과 유사한 view를 제공한다.

V$INSTANCE

현재 instance의 정보를 출력한다.

Column 이름	설명
SCN	현재 instance의 SCN 값
MIN_SCN	System이 인지한 minSCN 값
CURR_MIN_SCN	조회 시점의 실제 minSCN 값
MIN_SCN_TRANS_ID	전체 transaction 중 가장 작은 SCN을 가진 TransId
ACTIVE_MODE	Instance 활성화 여부
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 instance에 attach되어 사용 중인 세션의 정보를 출력한다.

Column 이름	설명
TRANS_ID	Session 고유 ID 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 중에 사용되는 undo page의 현재 ID 이다.
FIRST_UNDO_PAGE	트랜잭션 진행 중에 사용되는 첫번째 undo page의 ID 이다.
LAST_UNDO_PAGE	트랜잭션 진행 중에 사용되는 마지막 undo page의 ID 이다.
SAVEPOINT_UNDO_PAGE	트랜잭션 진행 중 implicit savepoint가 필요한 경우 해당 위치를 가리킨다.
SAVEPOINT_UNDO_OFFSET	트랜잭션 진행 중 implicit savepoint가 필요한 경우 해당 위치를 가리킨다.
WAIT_TRANS_ID	대기 중일 경우 LOCK을 점유한 상대 Trans ID 이다.
WAIT_OBJECT	대기 중일 경우 대상 테이블 이름이다.
WAIT_SLOT_ID	대기 중일 경우 데이터 공간의 고유 ID 이다.
STATUS	트랜잭션의 현재 상태이다.
IS_REPL	reserved
BEGIN_TIME	세션의 접속 시각이다.

```
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           : 2020-03-25 12:57:04
---------------------------------------------------------------------------------
```

V$TRANSACTION

세션들의 트랜잭션 진행 정보를 출력한다.

Column 이름	설명
TRANS_ID	Session 고유 번호
TRANS_SEQ	Session 내 트랜잭션 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_LOG_ENABLE로 설정하여 운영할 경우 관련된 disk logfile과 checkpoint와 관련된 정보를 출력한다.

Column 이름	설명
DISKLOG_ENABLE	Disk logging 설정 여부
ARCHIVE_ENABLE	Archive 설정 여부
CURR_FILE_NO	Reserved
CURR_FILE_OFFSET	Reserved
LAST_CKPT_FILE_NO	CheckPoint가 수행될 logfile의 시작 번호
LAST_ARCHIVE_FILE_NO	Archive가 시작될 대상 logfile 번호
LAST_CAPTURE_FILE_NO	unused
LOGCACHE_WRITE_IND	LogCache 공간 중 트랜잭션이 기록할 수 있는 현재 위치
LOGCACHE_READ_IND	LogCache 공간 중 flusher가 읽을 현재 위치
FLUSHER_FILE_NO	Flushser가 마지막으로 기록한 logfile의 번호
FLUSHER_FILE_OFFSET	Flushser가 마지막으로 기록한 logfile의 offset
LOG_DIR	Disk LogFile이 저장되는 경로
ARCHIVE_DIR	Checkpoint시점에 archiving 될 logfile이 저장될 경로

```
dbmMetaManager(DEMO)> select * from v$log_stat;
--------------------------------------------------------------------------------
DISKLOG_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/lim272/4th_iter/pkg/wal
ARCHIVE_DIR          :
---------------------------------------------------------------------------------
```

V$SYS_STAT

Instance가 생성된 후에 발생한 각 유형별 수행 횟수에 대해 누적된 정보를 출력한다.

Column 이름	설명
NAME	누적 유형의 이름
ACCUM_COUNT	누적 수치

```
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 항목에 대한 표는 다음과 같은데 v$sess_stat에서의 항목도 동일하다. 각 항목의 누적 수치는 성공/실패와 상관없이 내부 처리 과정에 진입한 횟수를 의미한다.

누적 항목	설명
init_handle_op	Instance에 D/A mode로 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 scan하는 모든 횟수
delete_op	delete가 수행된 횟수
enqueue_op	enqueue가 수행된 횟수
dequeue_op	dequeue가 수행된 횟수
aging_op	참조되지 않는 공간을 회수하는 과정이 처리된 횟수
commit_op	commit이 호출된 횟수

누적 항목	설명
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

V$SESS_STAT

Instance가 생성된 후에 발생한 세션 번호에 기록된 유형별 수행 횟수에 대해 누적된 정보를 출력한다. 이 정보는 세션에 접속한 이후의 정보가 아닌 누적치를 의미한다. 따라서 분석이 필요한 경우 before/ after와 같이 스냅샷을 조회하여 비교해야 접속 후의 변경량을 추정할 수 있다.

Column 이름	설명
TRANS_ID	세션의 고유번호
NAME	누적 유형의 이름
ACCUM_COUNT	누적 수치

```
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
-------------------------------------------------------------------------------
```

dbmMetaManager 명령어

dbmMetaManager는 DDL/ DML/ DCL을 interactive하게 수행하여 사용자가 별도의 프로그램을 작성하지 않고도 테스트할 수 있는 utility이다. 본 절에서는 dbmMetaManager에서만 동작하는 명령어에 대해 설명한다.

입력 옵션	설명
-i <Instance name>	특정 instance name을 지정
-f <script file>	특정 script file 내의 SQL을 순차적으로 실행 후 종료
-p <nick name>	dbmMetaManager가 구분되도록 별칭을 지정

list

현재 접속된 instance 이하의 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
```

MVCC 방식으로 운영되는 경우, 실제 사용공간 중 일부는 aging 대상으로만 존재할 수 있기 때문에 version-2와 달리 USED/FREE의 표시 부분이 실제 레코드 개수와 다를 수 있다.
USED는 실제 데이터와 aging 대상 공간의 합계로 해석해야 한다.

desc

테이블의 생성 정보를 화면에 출력한다.

```
dbmMetaManager(DICT)> desc dic_index_column;
-------------------------------------------------------------------------------------
Instance=(DICT) Table=(DIC_INDEX_COLUMN) Type=(TABLE) RowSize=(136)
-------------------------------------------------------------------------------------
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, TABLE_NAME, INDEX_NAME, COLUMN_NAME,
KEY_COLUMN_ORDER)
-------------------------------------------------------------------------------------
success
```

set index

테이블을 조회할 때 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
Access Count = 2
---------------------------------------------------------------------------------
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
```

set vertical [on/off]

dbmMetaManager 조회 결과는 기본적으로 column 별로 한 줄씩 출력된다. set vertical option을 통해 한 개 레코드 단위로 출력할 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 shell command 등을 수행할 필요가 있을 때는 +를 표기한 후에 명령어를 기술한다. 다음 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. 이 경우, 원격지에 dbmListener가 구동된 상태이어야 한다.

```
Connect := CONNECT <remote ip> <remote listen portNo> <instance>
```

다음은 Listener에 접속하여 instance를 생성하는 예이다.

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

C Type 구조체 출력

현재 생성되어 있는 테이블에 대한 C Type 구조체 형태를 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
```

1.4 복구 가이드

GOLDILOCKS LITE는 shared memory를 기반으로 하므로 OS fatal/ memory fault 상태가 발생하지 않으면 모든 데이터가 메모리상에 존재한다. 따라서 데이터 자체는 유실되지 않지만 전원 OFF 등으로 인한 장애로 인해 사용 환경에 따라 데이터 복구가 필요할 수 있다. 본 장에서는 이런 경우에 대비하여 사용자가 디스크 로깅과 데이터 파일을 이용하여 복구하는 방법을 설명한다.

디스크 로깅은 instance 생성 시점의 프로퍼티에 의해 설정되며 프로퍼티를 변경할 경우 instance를 재생성해야 하기 때문에 운영 전에 충분히 고려하여 프로퍼티를 설정해야 한다.

일시적 백업/ 복구

운영 중에 정기적인 PM 작업 등으로 인해 서버 전원을 차단해야 할 경우에 대비하여 전체 object/ data를 plain text 형태로 내려받는 tool (dbmExp)을 제공한다. 해당 tool을 이용하여 데이터를 백업한 후 서버 작업이 마무리되면 dbmExp를 통해 받은 파일을 이용하여 전체 object/ data를 복구할 수 있다.

```
[lim272@tech10 src]$ dbmExp -h
Usage] dbmExp <-i> <-t> <-r> <-c> <-d>
-i : specify a instance name to export
-t : specify a table name to export. (export all tables if exclude)
-r : specify a row delimeter when export data
-c : specify a column delimeter when export data
-d : use this option if need data-exporting
-h : Display this information
```

다음은 전체 instance의 모든 object/ data를 내려받는 예이다.

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

dbmExp를 수행하면 작업 경로 내에 다음과 같은 형태의 파일이 생성되는데 sql 파일들은 object를 생성할 SQL 구문을 포함한다. dat 파일들은 각 테이블별 데이터를 저장하고 있다. 복구할 때는 *_create.sql 파일을 이용하여 obje

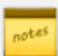
ct들을 생성한 후 *_in.sh을 수행하여 데이터를 다시 메모리에 로딩할 수 있다.

```
-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
```

디스크 로깅

디스크 로깅 방식은 기본적으로 트랜잭션의 commit 된 log를 disk에 모두 기록한다. 각 방식에 따라 세 가지 모드로 설정할 수 있다.

DBM_LOG_CACHE_MODE	설명
NONE (0)	각 세션별로 parallel 하게 디스크 로그 파일에 commit log를 기록한다.
NVDIMM (1)	NVDIMM 매체에 commit log를 기록한다. 디스크로 저장되는 작업은 별도의 프로세스가 처리하기 때문에 트랜잭션이 디스크 기록까지 대기하지 않는다.
SHM (2)	Shared memory를 이용하여 commit log를 기록한다.



Non-volatile Dual Inline Memory Module (NVDIMM)은 비휘발성 메모리 매체를 의미하는데 서버 전원이 갑자기 OFF 되더라도 자체 배터리와 SSD를 이용하여 NVDIMM 내에 저장된 데이터가 유실되지 않도록 보장하는 장치이다.

NVDIMM/ SHM 모드를 설정하면 제한된 메모리 공간을 이용하여 commit log를 기록하는데 이를 실제 디스크로 저장하려면 dbmLogFlusher라는 daemon을 구동시켜야 한다. dbmLogFlusher는 NVDIMM/ SHM 공간에 저장된 내용을 디스크 로그 파일로 저장한다.

디스크 저장은 일반적으로 process에서 kernel-buffer까지 또는 process에서 disk cache, physical disk까지 sync 하는 두 가지 방식으로 동작하며 이러한 동작을 위해 GOLDILOCKS LITE는 DBM_COMMIT_WAIT_MODE 설정이 가능하도록 지원한다. 이 프로퍼티는 성능 본위의 제품 특성상 비활성화되어 있는데 활성화시킬 경우 DISK까지 sync

를 보장한다. 다만 이 속성의 활성화는 결과적으로 안정성 보장만 목표로 하기 때문에 DML 성능은 효과적으로 발휘할 수 없게 된다.

체크포인트

GOLDILOCKS LITE는 디스크 로깅을 활성화했을 때 로그 파일을 계속 증가시키는 방식으로 파일을 생성/ 추가한다. 따라서 주기적으로 체크포인트를 수행하여 데이터 파일을 만들고 복구에 필요하지 않은 로그 파일을 삭제하여 디스크 사용 공간이 계속 증가되지 않도록 할 수 있다.

dbmCkpt daemon process는 체크 포인트를 담당하며 지정된 경로에 생성되는 로그 파일을 감지하여 데이터 파일에 commit log를 반영하고 반영 완료된 로그 파일을 archive 경로로 옮기거나 삭제하는 동작을 수행한다.
대량의 I/O가 발생할 수 있으므로 cron-tab 등에 주기를 설정하여 동작시키는 것이 바람직하며 이 때 해당 주기 동안 축적되는 로그 파일로 인해 디스크 용량 부족 등이 발생하지 않도록 해야 한다.

복구

디스크 로그와 체크포인트로 운영할 경우, 전원 OFF 후 복구 시점에 다음과 같이 복구할 수 있다.

```
~work> dbmCkpt -i demo -f
~work> dbmMetaManager
dbmMetaManager(unknown)> startup;
success
```

반드시 dbmCkpt를 먼저 수행해야 하는데 이는 마지막 체크포인트부터 반영되지 않은 로그 파일을 데이터 파일에 반영해야 가장 최신 데이터로 데이터 파일을 만들 수 있기 때문이다.
그 다음 dbmMetaManager를 수행하여 startup 명령을 통해 복구를 수행한다.
Startup 과정은 기존에 존재하던 object/ data를 마지막 시점까지 복구하는 일련의 과정을 모두 수행한다. 생성되는 데이터는 데이터 파일을 기준으로 적재되기 때문에 마지막으로 수행한 체크포인트 시점까지 반영된 데이터 파일을 기준으로 적용된다.

운영 중 복구 과정에서 마지막 체크포인트를 강제로 수행하지 못하고 실수로 startup 했다면 기존의 shared memory를 제거한 후 다시 체크포인트 수행하고 startup 하면 최종본까지 데이터를 복구할 수 있다. (기존의 shared memory만 잘 제거된 상태라면 startup은 반복적으로 수행해도 된다.)

1.5 Utility

GOLDILOCKS LITE에서 제공하는 사용자 utility에 대해 설명한다.

dbmExp

GOLDILOCKS LITE 내의 object 생성과 관련된 script와 데이터를 추출하는 utility 이다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance Name>	특정 instance를 지정할 경우에 입력한다. 입력하지 않을 경우, 전체를 추출한다.
-t <table Name>	특정 table을 지정할 경우에 입력한다. 입력하지 않을 경우 전체를 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	데이터를 추출하고자 할 경우에 입력한다.
-b	LITE간 binary 형태로 데이터를 추출할 경우에 입력한다. 추후 dbmImp로 로딩할 때 parsing 비용을 줄일 수 있다. (타 DBMS와는 호환 불가한 형식이다.)

- dbmExp에 의해 추출된 데이터는 plain text로 저장되며 다른 DBMS로 데이터를 이관하여 사용할 수 도 있다.
- csv 형태 추출을 지원하는데 이 경우 별도 옵션없이 사용자가 row와 column 구분자를 지정하지 않아야 한다.

사용 예 (특정 instance의 하위 object와 데이터 추출)

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

- 생성되는 각 파일 이름은 INSTANCENAME_OBJECTNAME와 같은 형식이다.
- 데이터 추출 옵션이 활성화되면 form file을 생성한다.
- Sequence는 current 값을 start with 값으로 설정하여 출력한다.

dbmExp 추출 결과물

dbmExp를 통해 아래의 표와 같은 script 결과물이 생성된다.
생성되는 script 파일 형식은 <INSTANCE_NAME>_xxx.sh(sql) 이다.

File name	설명
DEMO_create.sql	Table, queue, index와 같이 데이터를 저장하는 object를 생성하는 스크립트를 저장한다.
DEMO_create_proc.sql	Procedure 생성 스크립트를 저장한다.
DEMO_in.sh	모든 테이블의 데이터를 로딩하는 명령어를 저장한다.

dbmImp

dbmExp를 통해 추출된 데이터 또는 다른 DBMS로부터 추출된 구분자를 가진 text 형태의 데이터를 분석하여 적재하는 utility 이다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance Name>	대상 instance name을 지정한다.
-t <table Name>	대상 table name을 지정한다.
-r <delimeter>	레코드 단위 구분자를 지정한다.
-c <delimeter>	Column 단위 구분자를 지정한다.
-d <data file name>	데이터 파일 이름을 지정한다.
-f <form file name>	Form file 이름을 지정한다.
-b	데이터 파일이 dbmExp에 의해 만들어진 binary format일 경우에 지정한다. (Parsing 비용을 줄일 수 있다.)

Form file 정보를 수정하여 특정 column을 올리지 않거나 순서를 변경하여 데이터를 축적할 수 있다.

특수 문자로 제공되는 구분자는 다음과 같다.

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

사용 예 (TABLE, QUEUE에 데이터 축적)

```
$ 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 file을 이용하여 데이터에 존재하는 특정 column을 올리지 않으려면 Y 항목을 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 C2, C3 line을 제거한 후에 수행한다.
- 데이터 파일에 C2, C3 column의 순서가 바뀌어 저장된 경우, C2, C3 line을 서로 바꾸어 처리한다.

DATE 타입은 기본적으로 YYYY/MM/DD HH:MI:SS.SSSSSS 형태의 문자열을 기반으로 데이터를 해석하기 때문에 다른 DBMS에서 데이터를 추출하여 올려야 할 경우 해당 DBMS에서 동일한 형식으로 DATE를 문자열로 추출해야 한다.

dbmCkpt

DISK mode가 활성화된 상태에서 GOLDILOCKS LITE을 운영할 경우, 트랜잭션 정보를 저장한 logfile을 생성한다. 이 로그 파일은 영구적으로 증가하기 때문에 이를 제거하고 데이터 파일을 구성할 방법이 필요한데 dbmCkpt가 이런 역할을 수행한다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.

입력 옵션	설명
-i <instance name>	대상 instance name을 지정한다.
-v	한 번만 수행한 후에 종료한다.
-f	체크포인트 대상 logfile이 다 쓰여지지 않은 상태라도 강제로 체크포인트를 수행할지 여부를 지정한다. (Startup 직전 수행시점에 반드시 옵션을 포함한 체크포인트가 필요하다.)
-s	체크포인트를 수행한 후에 지정된 시간만큼 대기한다. (단위: 초)

체크포인트를 수행하면 다량의 disk I/O가 발생하여 성능에 영향을 줄 수 있기 때문에 crontab 등에 주기적으로 동작하도록 설정하거나 sleep interval을 명시하여 수행 주기를 지정해야 한다.

dbmCkpt 환경에서 데이터 복구

dbmCkpt를 사용하는 환경에서는 특정 시점에 체크포인트를 진행하여 반영된 데이터 파일과 이 과정에서 발생하는 데이터 파일에 미반영된 트랜잭션 로그 파일을 이용하여 최종 상태로 복구할 수 있다.
따라서 사용자가 다음 절차를 수행하면 shared memory가 유실되었더라도 데이터를 원래의 상태로 복원할 수 있다.

1. Force mode를 지정하여 dbmCkpt를 수행한다. (-f 옵션 사용)

```
Prompt> dbmCkpt -i demo -f -v
```

이 다음에 수행하는 startup 단계에서는 현재까지 반영된 데이터 파일을 기준으로 데이터를 로딩하기 때문에 마지막 체크포인트 시점부터 미반영된 트랜잭션 로그 내역은 아직 반영되지 않은 상태이다. 따라서 사용자가 임의로 체크포인트를 수행하여 미반영된 트랜잭션 로그를 반영하는 과정을 수행해야 한다.

2. startup을 수행한다.

```
dbmMetaManager(unknown)> startup;
```

현재의 데이터파일 이미지로 테이블/ index/ 데이터 등을 메모리에 원래의 상태로 복원한다.

Archive Log를 통한 데이터 파일 복구

데이터 파일이 삭제되었을 때 archive 방식이 아닐 경우 데이터는 복구할 수 없다. 그러나 archive 방식을 사용한 경우에는 checkpoint가 수행된 로그 파일도 archive directory에 보관되어 있으므로 이를 통해 데이터를 복구할 수 있다.

1. Checkpoint 시작 위치를 reset 한다.

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

이 명령은 현재 log anchor file 정보가 유효할 때 실행한 것이며 그렇지 않다면 shared memory가 삭제된 상태일 것이다. 이 경우 object 생성까지만 수행하므로 이 단계는 수행하지 않아도 된다.

2. Force mode로 dbmCkpt를 수행한다.

```
Prompt> dbmCkpt -i demo -f -v
```

3. Startup을 수행한다.

```
dbmMetaManager(unknown)> startup;
```

Dictionary datafile들인 "DIC_"를 prefix로 갖는 파일들이 손상되거나 유실된 경우에는 수동으로 object를 생성한 후 위의 복구 과정을 수행해야 한다. 단, 생성할 때 새로운 로그가 checkpoint에 적용되면 안되므로 object를 생성하는 세션은 DBM_DISK_LOG_ENALBE을 false 상태로 두고 접속해야 한다.

dbmDump

dbmDump는 여러 세트먼트와 파일을 dump하는 tool이다.
옵션과 파일 타입에 따라 해당되는 내용을 dump한다.

dbmDump (Instance)

Instance dump: Instance에서는 접근하는 session의 상태 정보와 트랜잭션의 진행 상태를 기록하고 있다.
dbmDump (Instance)는 이런 상태 정보를 출력하여 문제가 될 만한 session을 trace 하기 위한 용도로 제공한다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance name>	대상 instance name을 지정한다.
-t <session ID>	특정 session ID와 관련된 정보만 출력할 경우에 지정한다.

사용 예

```
$ dbmDump -i demo
InstName=DEMO, TransId=-1
Lock=-1, SCN=3000010, mSeed=0, ObjectId=0, Name=DEMO, CreateTime=2019-01-02 18:12:44
==============================================================================================
==============
TxInfo(T=1, P:48890, T=48890), Stat=transaction ready or running, First/Last(26:26), SavePoint
(-1:-1) WaitTrans=-1, SCN=-1
Page=26, PrevOffset=-1, Offset=32, NextLogPage=26, NextLogOffset=280, Size=104, LogType=LOCK_
ROW, RelSlot=1, ObjectName=(T1,) NeedSkip=0, Valid=1, MemCommit=0
Page=26, PrevOffset=32, Offset=280, NextLogPage=-1, NextLogOffset=-1, Size=104, LogType=UPDATE
_TABLE, RelSlot=1, ObjectName=(T1,) NeedSkip=0, Valid=1, MemCommit=0
```

dbmDump (Instance)는 instance header 정보와 각 session 별 트랜잭션 정보를 출력한다.
Instance header에 해당하는 정보들은 다음 표와 같다.

항목	설명
Lock	Instance lock 설정 여부
SCN	System commit number 용도
mSeed	Reserved
ObjectId	Reserved
CreateTime	Instance 생성 시각

Session 별 트랜잭션 출력 정보는 다음과 같으며 트랜잭션 수행 순서대로 출력된다.
Session의 master 정보는 다음 표와 같다.

항목	설명
TxInfo(T: P: T)	Session 정보 • T: 내부에서 할당된 session ID • P: Process ID • T: Thread ID
Stat	트랜잭션 상태 • transaction ready or running: 트랜잭션 진행 중 • memory commit completed: 메모리 커밋 완료 • memory rollback completed: 메모리 롤백 완료 • memory abnormal recovery completed: 비정상 복구 완료
First/ Last	Session이 사용 중인 undo 공간의 첫 번째와 마지막 PageId
SavePoint	트랜잭션이 진행되는 도중에 기록되는 implicit savepoint 정보
WaitTrans	트랜잭션이 LockWait 상태일 경우에 대기하는 상대편 SessionID
SCN	Commit 시점에 채번된 SCN

Session 별로 출력되는 트랜잭션 로그는 다음 표와 같다.

항목	설명
Page	현재 트랜잭션 로그가 기록된 현재 page ID
PrevOffset	이전 트랜잭션 로그가 기록된 page 내의 offset 정보
Offset	현재 트랜잭션 로그가 기록된 page 내의 offset 정보
NextLogPage	다음 트랜잭션 로그가 기록된 page ID
NextLogOffset	다음 트랜잭션 로그가 기록된 page 내의 offset 정보
Size	Rollback image의 크기
LogType	트랜잭션 로그 유형
RelSlot	Table내에 고유 레코드 ID
ObjectName	트랜잭션이 발생한 object name
NeedSkip	Partial rollback등으로 트랜잭션 로그가 무효화되는 경우에 1로 설정된다.
Valid	아직 로그 기록이 완료되기 전이거나 partial rollback 등으로 인해 트랜잭션 로그가 무효화 될 경우 0으로 설정된다.
MemCommit	Commit 과정에서 메모리 반영이 완료된 로그일 경우 1로 설정된다.

dbmDump (Anchor)

Anchor 파일의 이상 유무를 추적하는 tool이다

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-f <anchor filename>	대상 anchor filename을 지정한다

사용 예

```
$ dbmDump (Anchor) -f pkg/wal/DEMO.anchor
TransId=1, LogFileNo(0:3072), ArchiveStartFile(-1), MinCkptFileNo(-1), CaptureFileNo(-1)
LogCache: WriteInd=0, ReadInd=0, LogFileNo=0, Offset=0
```

dbmDump (Datafile)

Datafile의 이상 유무를 추적하는 tool 이다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-f <data filename>	대상 data filename을 지정한다

사용 예

```
$ dbmDump -f pkg/dbf/DEMO_T1.dbf
InstName=DEMO, TableName=T1, SlotSize=2128, CreateSCN=300010
ColumnName (C1), Order=1, Type=int, Offset=0, Size=4
ColumnName (C2), Order=2, Type=int, Offset=4, Size=4
ColumnName (C3), Order=3, Type=char, Offset=8, Size=2048
Dump (pkg/dbf/DEMO_T1.dbf) completed
```

dbmDump (Index)

Index의 이상 유무를 추적하는 tool 이다.

Input Option

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

사용 예

```
$ dbmDump -i demo -t t1 -d idx1_t1
InstName=DEMO, TableName=T1, IndexName=IDX1_T1
try to attach a index segment
Lock=-1, RootNodeId=-1, Unique=1, WorkType=0, mBeforeNewNodeToSplit=-1, mBeforeSrcNodeToSplit
=-1, BeforeParent=-1
         OldNewNodeToSplit=-1, OldSrcNodeToSplit=-1, OldParentNode=-1
         mBeforeRemoveNodeId=-1, mRecoveryTID=-1
```

dbmDump (Logfile)

로그 파일의 이상 유무를 추적하는 tool이다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-f <log filename>	대상 log filename을 지정한다

사용 예

```
$ dbmDump -f pkg/wal/DEMO.0.0 | tail
BlockSCN=204796, BlockCount=1, logCount=1
   + logType(DELETE_TABLE), object(T1), SlotId(4787)
BlockSCN=204797, BlockCount=1, logCount=1
   + logType(DELETE_TABLE), object(T1), SlotId(4788)
BlockSCN=204798, BlockCount=1, logCount=1
   + logType(DELETE_TABLE), object(T1), SlotId(4789)
BlockSCN=204799, BlockCount=1, logCount=1
   + logType(DELETE_TABLE), object(T1), SlotId(4790)
BlockSCN=204800, BlockCount=1, logCount=1
   + logType(DELETE_TABLE), object(T1), SlotId(4791)
```

dbmDump (Table)

테이블의 이상 유무를 추적하는 tool 이다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance name>	대상 instance name을 지정한다
-t <table name>	대상 table name을 지정한다
-s	slot-area 정보를 출력한다.

사용 예

```
$ dbmDump -i demo -t T1
InstName=DEMO, TableName=T1, RowSize=2056, CreateTime=2020-02-11 12:47:18
1] C1 int(2) 4 0
2] C2 int(2) 4 4
3] C3 char(6) 2048 8
```

dbmListener

원격지에서 GOLDILOCKS에 접속하기 위해 실행해야 하는 통신 데몬 프로세스이다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance name>	대상 instance name을 지정한다
-v	Non 데몬 모드로 구동된다.

dbmListener를 구동한 후에 사용자가 접속하려면 다음과 같은 두 가지 방식으로 테스트해 볼 수 있다.

```
dbmMetaManager> connect <ip> <port> <instanceName> ;
```

또는

dbmConnect API 사용

관련 환경변수	비고
DBM_TCP_CONN_TIMEOUT	지정된 시간 동안 연결에 성공하지 못하면 에러를 반환한다. (millisecond단위)
DBM_TCP_RECV_TIMEOUT	Execution 도중에 지정된 시간 동안 응답을 받지 못하면 에러를 반환한다 (millisecond단위)
DBM_LISTEN_PORT	LISTEN할 PortNo를 지정한다.

dbmLogFlusher

Log cache를 사용할 때 disk에 log cache를 저장하는 데몬 프로세스이다.

Input Option

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

관련 환경변수	비고
DBM_LOG_CACHE_FLUSH_INTERVAL	Log cache가 flush되는 인터벌을 설정한다. (millisecond 단위)
DBM_LOG_CACHE_EMPTY_INTERVAL	Log cache에 잘못된 값이 있어 skip 하는 인터벌을 설정한다. (millisecond 단위)

dbmMonitor

GOLDILOCKS LITE의 상태를 모니터링 하는 데몬 프로세스이다.

Input Option

입력 옵션	설명
-h	도움말을 출력한다.
-i <instance name>	대상 instance name을 지정한다
-s <interval time>	모니터링 하는 interval 시간을 설정한다. (second 단위)
-v	Non 데몬 모드로 구동된다.

사용 예

```
$ dbmMonitor -i demo
2020/02/11 11:15:28
[Table Usage]
TableName                                  TotalSize        AllocSize        UsedSize
FreeSize  Usage
----------------------------------------------------------------------------------------------
---------
REPL_LOG                                      100000             1024               0
1024   0.00
REPL_UNSENT                                  1000000             1024               0
1024   0.00
T1                                            102400             1024               0
1024   0.00
----------------------------------------------------------------------------------------------
---------
[Lock Status]
  WaitTransId   WaitTransPID   LockTransId   LockTransPID LockObject   LockSlot
-------------------------------------------------------------------------------
no waiting trans
-------------------------------------------------------------------------------
```

dbmReplica

GOLDILOCKS LITE의 이중화 방식은 active/ standby 구조로 동작한다.
Standby (slave) 측에서 데이터를 수신/반영하기 위해 구동하는 프로세스가 dbmReplica이다.

Input Option

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

사용 예

```
$ dbmRepica -i demo
```

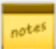
1.6 Sizing

GOLDILOCKS LITE를 사용하기 전에 필요한 sizing과 관련된 내용을 설명한다.

Instance Sizing

Instance는 rollback 하기 위해 before image를 저장하는 공간으로써 logging space는 1 page 당 1M이다. 이 공간에 N 개의 트랜잭션 로그를 기록하기 때문에 사용량은 가변적이며 확인할 수는 없다. 다만 instance segment의 크기는 아래의 수식을 통해 계산할 수 있다.

```
Instance Sizing =  sizeof(segment Header) +                // 240 byte
                   sizeof(trans Header) +                  // 753904 byte
                   sizeof(long long) * (init_size + 1) +
                   (1M * init_size)
                  ;
```



대량의 bulk update 등의 작업을 수행할 경우, instance 공간은 확장될 수 있고 최대 크기가 부족할 수 있다. 이런 bulk update는 GOLDILOCKS LITE의 특성과는 적합하지 않으나 부득이하게 실행해야할 경우, 미리 공간을 크게 설정해야 한다.

Table Sizing

Table은 실제 레코드가 저장되는 공간으로써 table header 공간 및 공간 관리를 위한 관리자 공간을 포함하여 각 record마다 기록되는 record header를 고려하여 sizing 해야 한다.

```
Table Sizing = sizeof(segment header)  +          // 240 byte
               sizeof(table header)    +          // 82052 byte
               sizeof(long long) * (init_size + 1) +
               (72(Record Header) + RecordSize) * init_size
               ;
```

init_size는 create table 시점에 지정한 option 값이다. Extend 된 segment는 init_size가 아닌 extent option 값으로 계산해야 한다.

Index Sizing

Index에는 별도의 공간을 설정하지 않고 table에 설정된 size에 key column의 key size를 계산하여 이를 최대 두 배수로 split 할 것을 가정한 공간을 설정한다. Splay, direct table 유형은 dummy index segment 만 만들기 때문에 sizing은 고려할 필요 없다.

1.7 Monitoring

본 장에서는 LITE를 실시간으로 모니터링하는 방법들에 대해 설명한다.

LOCK 정보 확인

모든 변경 연산들에 대해 record 단위로 lock을 점유하여 다른 세션으로부터의 데이터가 변경되지 않도록 보호한다. 정상적인 상황일 수도 있지만 사용자 실수로 commit/ rollback 하지 않아 장시간 대기하는 경우들을 찾기 위해 다음 방법을 사용한다.

V$SESSION 테이블에서 세션 중에 lock으로 인해 대기하는 경우는 다음과 같이 조회할 수 있다.

```
dbmMetaManager(DEMO)> select * from v$session where wait_trans_id <> -1;
--------------------------------------------------------------------------------
TRANS_ID             : 3
PID                  : 40832
TID                  : 40832
OLD_TID              : 40832
VIEWSCN              : 16
CURR_UNDO_PAGE       : 30
FIRST_UNDO_PAGE      : 30
LAST_UNDO_PAGE       : 35
SAVEPOINT_UNDO_PAGE  : 30
SAVEPOINT_UNDO_OFFSE : -1
WAIT_TRANS_ID        : 2
WAIT_OBJECT          : T1
WAIT_SLOT_ID         : 512
STATUS               : transaction ready or running
IS_LOGGING           : 0
IS_REPL              : 0
BEGIN_TIME           : 2021/04/16 13:25:05
--------------------------------------------------------------------------------
1 row selected
```

위 결과에서 WAIT_TRANS_ID 항목이 -1이 아니라면 해당 TransID를 가진 세션을 기다리고 있는 상태라는 것을 의미한다. 대기 중인 대상 테이블은 T1 이고 512번 slot을 가진 record에서 대기하고 있다는 의미이다.

위의 예에서처럼 TransID=2인 세션이 무엇을 하고 있는지는 다음과 같이 조회할 수 있다.

```
bmMetaManager(DEMO)> select * from v$transaction where trans_id = 2;
--------------------------------------------------------------------------------
TRANS_ID                : 2
TRANS_SEQ               : 1
TRANS_TYPE              : LOCK_ROW
OBJECT_NAME             : T1
SLOT_ID                 : 0
EXTRA_KEY               : 1
COMMIT_FLAG             : 0
SKIP_FLAG               : 0
VALID_FLAG              : 1
--------------------------------------------------------------------------------
TRANS_ID                : 2
TRANS_SEQ               : 2
TRANS_TYPE              : DELETE_TABLE
OBJECT_NAME             : T1
SLOT_ID                 : 0
EXTRA_KEY               : 1
COMMIT_FLAG             : 0
SKIP_FLAG               : 0
VALID_FLAG              : 1
--------------------------------------------------------------------------------
TRANS_ID                : 2
TRANS_SEQ               : 3
TRANS_TYPE              : LOCK_ROW
OBJECT_NAME             : T1
SLOT_ID                 : 512
EXTRA_KEY               : 513
COMMIT_FLAG             : 0
SKIP_FLAG               : 0
VALID_FLAG              : 1
--------------------------------------------------------------------------------
TRANS_ID                : 2
TRANS_SEQ               : 4
TRANS_TYPE              : INSERT_TABLE
OBJECT_NAME             : T1
SLOT_ID                 : 512
EXTRA_KEY               : 513
COMMIT_FLAG             : 0
SKIP_FLAG               : 0
VALID_FLAG              : 1
```

```
--------------------------------------------------------------------------------
4 row selected
```

SlotID=512를 가진 record를 변경하고 있는 세션이며 아직 commit/ rollback을 하지 않아 현재 상태가 유지되고 있다. TransID=2를 가진 세션이 어떤 프로세스인지 v$session을 통해 다음과 같이 확인할 수 있다.

```
dbmMetaManager(DEMO)> select * from v$session where trans_id = 2;
--------------------------------------------------------------------------------
TRANS_ID              : 2
PID                   : 40656
TID                   : 40656
OLD_TID               : 40656
VIEWSCN               : 9223372036854775807
CURR_UNDO_PAGE        : 24
FIRST_UNDO_PAGE       : 24
LAST_UNDO_PAGE        : 29
SAVEPOINT_UNDO_PAGE   : -1
SAVEPOINT_UNDO_OFFSE  : -1
WAIT_TRANS_ID         : -1
WAIT_OBJECT           :
WAIT_SLOT_ID          : -1
STATUS                : transaction ready or running
IS_LOGGING            : 0
IS_REPL               : 0
BEGIN_TIME            : 2021/04/16 13:24:41
--------------------------------------------------------------------------------
1 row selected
```

위의 예에서 PID=40656을 가진 프로세스라는 것을 확인할 수 있다.

처리량 확인

LITE는 각 주요 operation에 대한 누적 정보를 기록하고 있다. 다음과 같이 전체 누적량을 확인할 수 있다.

```
dbmMetaManager(DEMO)> select * from v$sys_stat;
--------------------------------------------------------------------------------
NAME                  : init_handle_op
ACCUM_COUNT           : 6
--------------------------------------------------------------------------------
NAME                  : free_handle_op
```

```
ACCUM_COUNT          : 3
--------------------------------------------------------------------------------
NAME                 : prepare_op
ACCUM_COUNT          : 27
--------------------------------------------------------------------------------
NAME                 : execute_op
ACCUM_COUNT          : 27
--------------------------------------------------------------------------------
NAME                 : insert_op
ACCUM_COUNT          : 514
--------------------------------------------------------------------------------
NAME                 : update_op
ACCUM_COUNT          : 2
--------------------------------------------------------------------------------
NAME                 : delete_op
ACCUM_COUNT          : 0
--------------------------------------------------------------------------------
NAME                 : scan_op
ACCUM_COUNT          : 1029
--------------------------------------------------------------------------------
NAME                 : enqueue_op
ACCUM_COUNT          : 0
--------------------------------------------------------------------------------
NAME                 : dequeue_op
ACCUM_COUNT          : 0
--------------------------------------------------------------------------------
NAME                 : aging_op
ACCUM_COUNT          : 0
--------------------------------------------------------------------------------
NAME                 : commit_op
ACCUM_COUNT          : 16
--------------------------------------------------------------------------------
NAME                 : rollback_op
ACCUM_COUNT          : 4
--------------------------------------------------------------------------------
NAME                 : recovery_rollback_op
ACCUM_COUNT          : 0
--------------------------------------------------------------------------------
NAME                 : recovery_commit_op
ACCUM_COUNT          : 0
--------------------------------------------------------------------------------
```

```
NAME                 : split_index_node
ACCUM_COUNT          : 2
--------------------------------------------------------------------------------
16 row selected
```

성능면에서 다음 항목들의 변화 추이를 확인해봐야 한다.

- 빈번한 연결/ 해제 반복 (init_handle 증가)
- 빈번한 prepare 호출 (prepare_op 증가)
- 개별 operation에 대한 처리량 변화 추이 (insert_op, update_op, delete_op 등의 변화추이)
- aging_op 증가는 대량 변경 작업으로 versioning 된 row들이 많아질 때 발생할 수 있다.
- aging 작업이 많을 경우 변경 DML 또는 조회 시점마다 aging 작업 즉 참조되지 않는 레코드 공간을 정리하는 작업이 발생하여 성능이 저하될 수 있다.

비정상적인 프로세스 종료 등이 발생한 경우, 다음 항목을 확인해봐야 한다.

- recovery_rollback_op (프로세스가 종료된 후 rollback 형태로 복구가 진행된 경우)
- recovery_commit_op (프로세스가 이미 commit 된 후 종료하여 이에 대한 리소스만 정리한 경우)

위와 같은 항목들은 프로세스 별로 V$sess_stat에서 동일한 항목으로 조회가 가능하므로 특정 프로세스의 추이를 추적하고자 할 경우 v$sess_stat을 참고한다.

Log Cache 및 Checkpoint 상태

v$log_stat 정보는 디스크 로깅과 관련된 전반적인 상태를 출력한다.

```
dbmMetaManager(DEMO)> select * from v$log_stat;
--------------------------------------------------------------------------------
DISKLOG_ENABLE       : 1
ARCHIVE_ENABLE       : 0
CURR_FILE_NO         : 45
CURR_FILE_OFFSET     : 45
LAST_CKPT_FILE_NO    : 45
LAST_ARCHIVE_FILE_NO : 45
LAST_CAPTURE_FILE_NO : -1
LOGCACHE_WRITE_IND   : 9600016
LOGCACHE_READ_IND    : 9600016
FLUSHER_FILE_NO      : 46
FLUSHER_FILE_OFFSET  : 0
LOG_DIR              : /home/lim272/4th_iter/pkg/wal
ARCHIVE_DIR          : /home/lim272/4th_iter/pkg/arch
```

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

체크포인트 운영 시 확인 사항

- CURR_FILE_NO
  - 트랜잭션이 기록되는 가장 마지막 로그 파일 번호로써 이것이 증가하지 않을 경우 확인해야 한다.
- LAST_CKPT_FILE_NO
  - dbmCkpt에 의해 체크포인트가 진행된 후 완료된 마지막 로그 파일 번호이다.
- LAST_ARCHIVE_FILE_NO
  - Archiving 동작이 반영된 마지막 로그 파일 번호이다.

위의 세 값들은 상황에 따라 다를 수 있으나 기본적으로 dbmCkpt, dbmLogFlusher 들의 상태를 의미하므로 트랜잭션이 지속적으로 발생하는데도 이 값들이 변동되지 않을 경우에는 dbmCkpt와 dbmLogFlusher 프로세스들의 동작 상태를 체크해야 한다. 만일 프로세스 문제가 아니라면 디스크 공간 부족 등의 HW 적인 원인으로 인해 오류가 발생할 수도 있으므로 HW 부분을 체크해야 한다.

LOG_CACHE MODE 운영 시 확인 사항

- LOGCACHE_WRITE_IND
  - LOG_CACHE MODE로 설정했을 때 변경되며 트랜잭션이 commit 될 때마다 증가한다.
  - 만일 log cache 내에 공간이 부족할 경우 이 값은 증가하지 않고 같은 값으로 유지된다.
- LOGCACHE_READ_IND
  - dbmLogFlusher가 LogCache의 내용을 디스크로 기록한 후에 증가된다.
  - dbmLogFlusher가 정상적으로 동작하지 않으면 위 두 개 항목의 값들도 모두 변경되지 않으므로 이를 확인해야 한다.

2.

API Reference

2.1 API 공통사항

- 모든 API 호출은 정상 처리 결과를 "0"으로 반환하며 오류에는 개별 상황에 따른 에러코드를 반환한다.
- Compile 할 때 $DBM_HOME/include/dbmUserAPI.h을 사용자가 작성한 Makefile 내에 포함해야 한다.
- Linking 할 때 $DBM_HOME/lib/libdbmCore.so을 사용자가 작성한 Makefile 내에 포함해야 하며 사용자 환경 변수인 LD_LIBRARY_PATH에도 추가해야 한다.
- 모든 API 동작은 non-autoCommit 모드이기 때문에 select를 제외한 트랜잭션 마지막에는 dbmCommit 또는 dbmRollback을 호출해야 반영이 완료된다.
- 다른 API와 달리 dbmExecuteStmt로 수행된 결과에 결과 집합이 없을 경우에는 NOT_FOUND 오류를 반환하지 않으므로 dbmGetRowCount를 통해 결과 개수를 확인해야 한다.
- API는 TableName을 이용하거나 dbmTableHandle을 이용하는 두 가지 유형이 있으므로 API 상세 spec을 참고하여 적절한 유형을 사용해야 한다.

2.2 dbmInitHandle

기능

API 사용을 위한 handle 초기화 작업을 수행한다. 모든 API를 사용하기 위해 반드시 선행되어 호출되어야 한다. dbmInitHandle 내에서는 아래의 과정을 내부적으로 수행한 후 성공/실패를 반환한다.

- Process 정보 저장
- Undo space 할당
- Dictionary 준비
- 트랜잭션 처리를 위한 내부 공간 할당

인자

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

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

사용 예

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

dbmHandle 변수는 (void *) 형태로 dbmInitHandle을 통해 내부적으로 필요한 공간을 할당하여 사용자에게 반환한다.

2.3 dbmFreeHandle

기능

dbmInitHandle에 의해 할당된 자원을 모두 해제한다.
만일 변경 트랜잭션을 commit 하지 않은 상태에서 dbmFreeHandle이 호출되면 자동으로 rollback을 먼저 수행한다.
dbmFreeHandle을 호출하지 않고 프로그램을 종료할 수 있지만 dbmFreeHandle 없이 dbmInitHandle를 계속 호출하면 메모리가 증가하며 동시에 접속 가능한 공간이 부족한 오류가 발생할 수 있다.

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 );
}
```

2.4 dbmPrepareTable

기능

사용자가 수행할 table을 handle에 준비시킨다. (Shared memory attach 등을 수행)

인자

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

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수가 사용되어야 한다.
aTableName	char *	In	Prepare할 TableName을 입력한다.

사용 예

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

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

2.5 dbmPrepareTableHandle

기능

사용자가 DML/ SELECT를 수행할 table에 대한 TableHandle을 생성하여 반환한다. (Shared memory attach 등을 수행한다.)

인자

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

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수가 사용되어야 한다.
aTableName	char *	In	Prepare 할 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 );
}
```

만일 대상 table에 DDL이 발생하면 내부적으로 prepare를 재수행한다. 이 때 table의 변경 정보를 재구축하는 과정에서 메모리 사용량이 일부 증가할 수 있으며 성능의 jitter가 발생할 수 있다.

2.6 dbmPrepareStmt

기능

사용자가 수행할 SQL을 실행 직전 단계로 만든다.

- 구문 분석
- Object 정합성 등의 validation 수행
- DDL/ DML을 모두 사용할 수 있다. (단, commit/ rollback 구문은 지원하지 않는다.)

인자

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

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수가 사용되어야 한다.
aSQLString	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 );
}
```

- SQL 내에서 parameter를 기술하는 방법은 다음 두 가지인데 섞어서 쓸 수는 없으며 한 가지 형태로만 나열해야 한다.
  - Marker (?) 를 이용하여 순서대로 나열하는 방법
  - :v1 과 같이 이름을 명시하는 방법
- dbmStmt는 (void *) 형태 변수로 dbmPrepareStmt는 prepared 된 결과를 dbmStmt 변수에 반환한다.

Procedure는 다음과 같은 방식으로 호출한다. Procedure를 호출할 때는 out mode parameter인 경우에도 BindParamById와 같은 BindParam 함수를 사용해야 한다.

```
rc = dbmPrepareStmt( sHandle, "exec proc1(?, ?, ?, ?)", &sStmt );
    TEST_ERR( sHandle, rc, "prepareStmt" );
    rc = dbmBindParamById( sHandle, sStmt, 1, DBM_BIND_DATA_TYPE_INT, &sData.c1, NULL );
    TEST_ERR( sHandle, rc, "bindParam1" );
    rc = dbmBindParamById( sHandle, sStmt, 2, DBM_BIND_DATA_TYPE_INT, &sData.c2, NULL );
    TEST_ERR( sHandle, rc, "bindParam2" );
    rc = dbmBindParamById( sHandle, sStmt, 3, DBM_BIND_DATA_TYPE_INT, &sData.c3, NULL );
    TEST_ERR( sHandle, rc, "bindParam3" );
    rc = dbmBindParamById( sHandle, sStmt, 4, DBM_BIND_DATA_TYPE_INT, &sData.c4, NULL );
    TEST_ERR( sHandle, rc, "bindParam4" );
    for( i = 0 ; i < 10 ; i ++ )
    {
        sData.c1 = i;
        sData.c2 = i;
        sData.c3 = i;
        rc = dbmExecuteStmt( sHandle, sStmt );
        TEST_ERR( sHandle, rc, "executeStmt" );
        fprintf( stdout, "c4=%d\n", sData.c4 );
    }
```

dbmPrepareStmt에서 생성되는 dbmStmt 객체는 공유할 수 없다. 따라서 서로 다른 handle을 쓸 경우, dbmPrepareStmt를 통해 각 handle별 dbmStmt 객체를 생성해야 한다.

2.7 dbmFreeStmt

기능

사용자가 dbmPrepareStmt를 통해 할당 dbmStmt Handle의 메모리를 해제시킨다.

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 );
}
```

2.8 dbmBindParamById

기능

dbmPrepareStmt에서 사용된 사용자 parameter marker (?)에 대응되는 변수를 순서대로 binding 한다.

- 사용자 변수는 input mode로만 사용할 수 있다.
- Binding 되는 변수의 pointer를 지정해야 하며 dbmPrepareStmt와 dbmExecuteStmt 사이에 호출한다.
- 길이 정보를 포함하는 변수를 지정하지 않을 경우, CHAR 타입은 실행 시점에 바인딩 된 변수의 string length를 구해 실행한다. 따라서 NULL을 포함하지 않는 변수일 경우 알 수 없는 오류가 발생할 수 있다. 그 외의 타입은 NULL로 지정할 수 있다.

인자

```
int dbmBindParamById( dbmHandle          * aHandle,
                      dbmStmt            * aStmt,
                      int                  aBindId,
                      dbmBindDataType      aBindType,
                      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	-	Prepare 시점에 나열된 parameter의 순서 (base=1) 이다.
aBindType	dbmBindDataType	In	Binding 변수의 데이터 타입이다.
aData	void *	In	사용자 변수 포인터이다.
aSizePtr	long *	In	데이터 크기를 저장하는 8 byte 변수 포인터

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 );
}
```

dbmBindParamById는 사용자 데이터만 연결할 수 있고 테이블이나 column 이름은 binding 할 수 없다.

JSON Object 예제

다음은 json에 대해 dbmBindParamById를 사용하는 예이다.

```
dbmStmt * sStmt = NULL;
int sC1;
```

- 일반 int 변수 bind

```
sRc = dbmPrepareStmt( aHandle,
           "select c1, json_value(c2, '$.key1') from t1 where c1 = ?",
           &sStmt );
PRT_ERROR( sRc );
sRc = dbmBindParamById( aHandle,
                        sStmt,
                        1,
                        DBM_BIND_DATA_TYPE_INT,
                        &sC1,
                        NULL );
PRT_ERROR( sRc );
```

```
dbmStmt * sStmt = NULL;
int sC1;
char * sPath;
// json path bind
sRc = dbmPrepareStmt( aHandle,
           "select c1, json_query(c2, ?) from t1 where c1 = ?",
           &sStmt );
PRT_ERROR( sRc );
sRc = dbmBindParamById( aHandle,
                        sStmt,
                        1,
                        DBM_BIND_DATA_TYPE_CHAR,
                        &sPath,
                        NULL );
PRT_ERROR( sRc );
sRc = dbmBindParamById( aHandle,
                        sStmt,
                        2,
                        DBM_BIND_DATA_TYPE_INT,
                        &sC1,
                        NULL );
PRT_ERROR( sRc );
sPath = "$.key1";
```

```
dbmStmt * sStmt = NULL;
char * sSelectPath;
char * sWherePath;
// json path bind
sRc = dbmPrepareStmt( aHandle,
                      "select c1, json_query(c2, ?) from t1 "
                      "where json_value(c2, ?) = ?",
                      &sStmt );
PRT_ERROR( sRc );
sRc = dbmBindParamById( aHandle,
                        sStmt,
                        1,
                        DBM_BIND_DATA_TYPE_CHAR,
                        &sSelectPath,
                        NULL );
PRT_ERROR( sRc );
sRc = dbmBindParamById( aHandle,
```

```
                        sStmt,
                        2,
                        DBM_BIND_DATA_TYPE_CHAR,
                        &sWherePath,
                        NULL );
PRT_ERROR( sRc );
sSelectPath = "$.key2";
sWherePath = "$.key1";
```

2.9 dbmBindParamByName

기능

dbmPrepareStmt에서 사용자가 기술한 변수 이름을 기반으로 사용자 변수를 binding 한다.

- 사용자 변수는 input mode로만 사용할 수 있다.
- Binding 되는 변수의 pointer를 지정해야 하며 dbmPrepareStmt와 dbmExecuteStmt 사이에 호출한다.

인자

```
int dbmBindParamByName( dbmHandle          * aHandle,
                        dbmStmt            * aStmt,
                        char               * aVarName,
                        dbmBindDataType      aBindType,
                        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.
aBindType	dbmBindDataType	In	Binding 변수의 데이터 타입이다.
aData	void *	In	사용자 변수 포인터이다.
aSizePtr	long *	In	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 );
}
```

dbmBindParamByName은 사용자 데이터만 연결할 수 있고 테이블이나 column 이름은 binding 할 수 없다.

2.10 dbmBindCol

기능

Select 구문에 의해 수행된 결과를 저장하기 위한 사용자 변수와 매핑하는 역할을 수행한다.

- dbmExecuteStmt에 의해 수행된 질의가 select 구문이어야 한다.
- 실제 select target 절의 데이터 크기와 사용자 변수의 크기가 다르므로 결과를 복사할 때 오류가 발생하지 않도록 주의해야 한다.

인자

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

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	Prepared 된 stmt 변수이다.
aBindIndex	int	In	Target column이 출현하는 순서이다. (Base=1)
aBindDataType	dbmBindDataType	In	Target 변수의 데이터 타입이다.
aData	void *	In	사용자 변수 포인터이다.
aMaxSize	long	In	데이터 크기의 최대값이다
aSizePtr	long *	In	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 );
}
```

2.11 dbmBindColStruct

기능

Select 구문에 의해 수행된 결과를 저장하기 위해 사용자 변수와 매핑한다는 점에서 dbmBindCol과 동일하다. 다만 사용자가 정의한 구조체 변수로 반환받고자 할 때 사용한다는 차이가 있다.

- dbmExecuteStmt에 의해 수행된 질의가 select 구문이어야 한다.
- 구조체를 한 개만 binding 할 수 있고 dbmBindCol과 혼용하여 사용할 수 없다.
- 실제 select target 절의 데이터 크기와 사용자 변수의 크기가 다른 경우 메모리 침범등의 오류가 발생할 수 있으므로 target 절이 모두 column이거나 구조체가 반환되는 target 절의 크기와 일치하는 경우에 사용할 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" );
```

- Prepare select stmt

```
c1 = 10;
    rc = dbmPrepareStmt( sHandle,
                         "select * from t1 where c1 = ?",
                         & sStmt );
    TEST_ERR( sHandle, rc, "prepareStmt" );
```

- 조건절 binding

```
rc = dbmBindParamById( sHandle,
                           sStmt,
                           1,
                           DBM_BIND_DATA_TYPE_INT,
                           &c1,
                           NULL );
TEST_ERR( sHandle, rc, "selectBindParam" );
```

- Target binding

```
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;
}
```

2.12 dbmExecuteStmt

기능

dbmPrepareStmt에 의해 처리된 statement를 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 );
}
```

Select하고 대상을 추려야 하는 update/ delete 문의 경우, 명시적인 dbmBeginCursor() API 호출이 없었다면 실행되는 시점의 SCN을 자신의 view-SCN으로 설정하여 데이터의 visibility를 판별한다.

2.13 dbmFetchStmt

기능

dbmExecuteStmt에 의해 수행된 select 문 조회 결과를 한 건씩 가지고 온다.

- Select 문이 아닌데 호출될 경우 오류가 발생한다.
- dbmBindCol에 의해 미리 사용자 변수가 지정되어야 하며 잘못 설정된 경우 오류가 발생한다.

인자

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

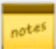
인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	Executed 된 stmt 변수이다.

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 );
    rc = dbmFetchStmt( sHandle,
                       sStmt );
}
```



dbmFetchStmt가 호출되면 앞서 dbmBindCol 했던 각 변수에 select target 절에 해당하는 값들이 저장된다. 여러 건이 있을 경우, loop 안에 NOT_FOUND 오류가 나올 때까지 dbmFetchStmt를 반복적으로 호출하면 된다.

2.14 dbmFetchStmt2Json

기능

dbmExecuteStmt가 select 문을 조회한 결과를 한 건씩 JSON format text로 가져온다.

- Select 문이 아닌데 호출될 경우 오류가 발생한다.
- 입력된 버퍼의 크기는 충분히 크게 할당하여 사용해야 한다.

인자

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

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aStmt	dbmStmt *	In	Execute 된 stmt 변수이다.
aDataPtr	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 );
    }
}
```

2.15 dbmInsertRow

기능

한 개의 사용자 데이터를 지정된 테이블에 추가한다.

인자

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

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	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) );
}
```

2.16 dbmInsertRowExpired

기능

지정된 시간이 경과하면 expired 될 record를 삽입한다. 입력된 시간이 경과하면 데이터는 제거된다. 0을 입력하면 데이터는 유지되고 시간 (초 단위)을 지정하면 해당 시간이 경과된 후에 제거된다.

인자

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

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 테이블 이름이다.
aUserData	void *	In	사용자 변수 포인터이다.
aDataSize	int	In	데이터 크기이다.
aSecond	int	in	0: dbmInsertRow와 동일하게 동작 aSecond > 0: 해당 시간 경과 후 데이터 제거

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 = dbmInsertRowExpired( sHandle,
                              "t1",
                              & sData,
                              sizeof(DATA),
                              600 );         // 10분 후 제거됨.
}
```

2.17 dbmInsert

기능

한 개의 사용자 데이터를 table handle에 지정된 테이블에 추가하고 row의 slot ID를 반환한다.

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.

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 = dbmInsertRow( sHandle,
                       sTableHandle,
                       & sData,
                       sizeof(DATA),
                       & sSlotId );
}
```

2.18 dbmUpdateRow

기능

한 건 이상의 사용자 데이터를 갱신한다.

- Key를 갱신하지 않고 데이터만 갱신할 경우에 빠르게 처리하기 위해 호출한다.
- Before/ after의 update로 인해 key column의 값이 갱신될 경우, dbmPrepareStmt/ dbmExecuteStmt를 통해 수행해야 한다.

인자

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

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 테이블 이름이다.
aUserData	void *	In	사용자 변수 포인터이다.
aAffectedCount	int *	Out	Updated 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 );
}
```

Unique index의 경우 dbmUpdateRow는 한 건만 갱신되지만 non-unique index의 경우 여러 건이 갱신될 수 있는데 이 경우 해당 건수는 반환되는 aAffectedCount를 통해 확인할 수 있다.

2.19 dbmUpdate

기능

지정된 table handle의 테이블에서 주어진 데이터에 해당하는 key를 가진 데이터를 변경하며, 필요한 경우 변경 전의 data를 반환한다.

aSlotID 인자가 -1 이 아닌 경우, 해당 slot의 데이터를 변경한다.

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, 해당 slot의 데이터를 변경한다.
aRowCount	int *	Out	Updated 된 row 개수이다.
aReturnOldData	void *	Out	NULL이 아닐 경우 이전 data를 반환한다.

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 );
}
```

2.20 dbmUpsert

기능

지정된 table handle의 테이블에서 주어진 데이터에 해당하는 key를 가진 데이터가 있으면 변경하고 없으면 insert를 수행한다. Update 되었을 때 필요한 경우 변경되기 전의 data를 반환한다.

변경 전의 data가 필요 없으면 aReturnOldData를 NULL로 설정해야 하지만, aInsertCnt는 NULL이 아닌 경우 insert로 동작이 성공하면 1을 반환하고 update로 성공하면 0을 반환한다.

변경 전의 data가 필요하면 aReturnOldData를 NULL이 아닌 값으로 설정해야 하고, 해당 변수값은 update 일 때만 유효하므로 이 변수가 유효한지 여부를 판단하기 위해 aInsertCnt도 NULL이 아닌 값으로 설정해야 한다.

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 아닐 경우 이전 data를 반환한다.
aInsertCnt	int *	Out	NULL이 아닐 경우 • Insert로 성공하면 1을 반환한다. • Update로 성공하면 0을 반환한다.

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);
}
```

2.21 dbmDeleteRow

기능

한 건 이상의 사용자 데이터를 삭제한다.

인자

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

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

2.22 dbmDelete

기능

주어진 table handle의 테이블에서 한 건 이상의 사용자 데이터를 삭제한다.

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 데이터이다.

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 = dbmDeleteRow( sHandle,
                       sTableHandle,
                       & sData,
                       -1,
                       & sRowCount,
                       & sOldData );
}
```

Splay table type에서 delete 하면 즉시 key를 삭제한다. 따라서 트랜잭션이 진행되는 도중에 삭제된 key는 조회되지 않는다. 또한 삭제된 트랜잭션을 롤백하려고 할 때 이미 다른 세션에 의해 삽입되었을 경우에는 삭제된 트랜잭션이 롤백되지 않고 해당 레코드는 유실된다.

2.23 dbmBindColumn

기능

dbmUpdateRowByCols를 호출할 때 특정 column에 사용자 데이터를 저장하기 위해 사용한다.
dbmBindColumn을 수행하는 시점에 내부 임시 버퍼에 사용자 데이터가 복제된다. 따라서 execution하기 전에 반드시 dbmBindColumn을 호출해야 한다.
동일한 column을 대상으로 반복적으로 dbmBindColumn을 호출할 경우 마지막에 호출한 값이 저장되며 존재하지 않는 column에 대해 수행할 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	char *	In	대상 테이블 이름이다.
aColumnName	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) );
}
```

2.24 dbmClearBind

기능

dbmBindColumn을 수행한 정보를 삭제한다. 특정 column만 지정할 수 있으며 column을 명시하지 않고 NULL로 지정할 경우 이전에 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	char *	In	대상 테이블 이름이다.
aColumnName	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 );
}
```

2.25 dbmUpdateRowByCols

기능

특정 column만 갱신하고자 할 경우에 사용한다. 호출하기 전에 미리 dbmBindColumn API를 이용하여 key value를 포함하여 갱신하려는 column에 값을 설정해야 한다.

인자

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

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	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 );
}
```

2.26 dbmSelectRow

기능

한 건의 사용자 데이터를 조회하고 fetch 한다.
여러 건이 조회되더라도 첫 번째 한 건만 가지고오는데 다음 건을 가져와야 할 경우, dbmFetchNext 계열의 API를 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 );
}
```

대상 레코드를 조회하려면 dbmSelectRow, dbmUpdateRow, dbmDeleteRow 모두 실행 시점에 사용자 데이터에 key를 포함한 상태여야 한다. dbmSelectRow는 사용자 데이터를 조회한 후에 사용자 변수에 저장하기까지 한다.

Date 타입을 반환받아야 할 경우에는 사용자가 unsigned long long 형태의 8 byte 변수를 지정하여 값을 저장할 수 있다.

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;
} 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
```

API에 의해 변경된 시간정보가 table에도 동일하게 반영되어 있다는 것을 확인할 수 있다.

```
dbmMetaManager(DEMO)> select * from t1;
--------------------------------------------------------------------------------
C1                   : 1
C2                   : 2020/12/23 17:49:33.214220
--------------------------------------------------------------------------------
1 row selected
dbmMetaManager(DEMO)>
```

2.27 dbmSelect

기능

주어진 table handle의 테이블에서 주어진 조건에 맞는 사용자 데이터를 조회하기 시작하고 fetch 한다.
한 건 이상의 데이터를 조회하려면 계속해서 dbmFetch 함수를 사용해야 한다.

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. 검색된 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	• DBM_SCAN_DIR_BACKWARD: 인덱스의 역방향 탐색 • DBM_SCAN_DIR_FORWARD: 인덱스의 정방향 탐색 • DBM_SCAN_DIR_EQUAL: 같은 값을 가지는 key 탐색
aScanType	dbmScanType	in	• DBM_SCAN_TYPE_RDONLY: read-only 로 검색 • DBM_SCAN_TYPE_FOR_UPDATE: FOR UPDATE 모드로 검색

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 );
}
```

DBM_SCAN_DIR_FORWARD는 aUserData 보다 큰 다음 (index order 기준) 데이터를 반환하고, UntilData의 key 보다 큰 경우에는 NOT_FOUND를 반환한다.
DBM_SCAN_DIR_BACKWARD는 aUserData 보다 작은 다음 (index order 기준) 데이터를 반환하고, UntilData의 key 보다 작은 경우에는 NOT_FOUND를 반환한다.
DBM_SCAN_DIR_EQUAL은 aUserData와 값이 같은 다음 데이터를 반환한다.

2.28 dbmFetch

기능

주어진 table handle의 테이블에서 수행한 이전 검색에 이어서 다음 데이터를 조회한다.

반드시 dbmSelect가 먼저 호출된 이후에 사용해야 한다.

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
aScanDir	dbmScanDirection	in	• DBM_SCAN_DIR_BACKWARD : 인덱스의 역방향 탐색 • DBM_SCAN_DIR_FORWARD : 인덱스의 정방향 탐색 • DBM_SCAN_DIR_EQUAL : 같은 값을 가지는 key 탐색
aScanType	dbmScanType	in	• DBM_SCAN_TYPE_RDONLY : read-only 로 검색 • DBM_SCAN_TYPE_FOR_UPDATE : FOR UPDATE 모드로 검색

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;
   }
}
```

DBM_SCAN_DIR_FORWARD는 이전에 반환한 데이터보다 큰 다음 (index order 기준) 데이터를 반환하고, UntilData의 key 보다 큰 경우에는 NOT_FOUND를 반환한다.
DBM_SCAN_DIR_BACKWARD는 이전에 반환한 데이터보다 작은 다음 (index order 기준) 데이터를 반환하고, UntilData의 key 보다 작은 경우에는 NOT_FOUND를 반환한다.
DBM_SCAN_DIR_EQUAL은 이전에 반환한 데이터와 값이 같은 다음 데이터를 반환하고, 해당 인덱스가 non-unique 인덱스 속성일 때만 의미가 있다.

2.29 dbmSelectMax

기능

주어진 table handle의 direct 테이블에 대해서만 key로 설정된 데이터 중 최대값을 조회한다.

테이블이 가진 segment들 중에 데이터가 하나라도 들어간 마지막 segment를 찾아서 이진 탐색을 통해 최대값을 가진 데이터를 반환한다.

이진 탐색 중에 데이터가 들어 있지 않은 slot을 발견할 경우, 잘못된 결과를 반환할 수 있다.

인자

```
int dbmSelectMax( dbmHandle      * aHandle,
                  dbmTableHandle * aTableHandle,
                  void           * aUserData );
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableHandle	dbmTableHandle *	In	대상 테이블의 핸들이다.
aUserData	void *	In/ out	사용자 변수 포인터이다.

사용 예

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

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

dbmSelectMax API는 direct table만 지원한다.

2.30 dbmSelectCount

기능

입력된 데이터와 일치하는 레코드의 개수를 반환한다.

인자

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

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.
aRetCount	int *	out	데이터의 건수를 반환 받을 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 = dbmSelectCount( sHandle,
```

```
                            "t1",
                            & sData,
                            & sRowCount );
}
```

일치하는 대상이 없을 경우 RowCount는 0으로 반환된다.

2.31 dbmSelectRowGT

기능

입력된 데이터의 key 값보다 큰 데이터를 조회한다.

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 );
}
```

위의 사용 예에서 dbmSelectRowGT를 수행하여 1보다 큰 데이터가 존재할 경우, 이를 가지고 온다.

2.32 dbmSelectRowLT

기능

입력된 데이터의 key 값보다 작은 데이터를 조회한다.

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 );
}
```

위의 예에서 dbmSelectRowLT를 수행하여 10보다 작은 데이터가 존재할 경우, 이를 가지고 온다.

2.33 dbmFetchNext

기능

입력된 데이터의 key 값과 동일한 다음 데이터를 조회한다.
Non-unique index에 여러 건의 데이터가 존재하는 까닭에 동일한 key 값을 갖는 여러 레코드를 가져오기 위해 dbmSelectRow와 조합하여 fetch를 수행해야 할 경우에 사용한다.

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;
    }
}
```

2.34 dbmFetchNextGT

기능

입력된 데이터의 key 값보다 큰 데이터를 조회한다. dbmSelectRow 또는 dbmSelectRowGT에서 시작된 값을 기준으로 조회한다.

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;
    }
}
```

2.35 dbmFetchNextLT

기능

입력된 데이터의 key 값보다 작은 데이터를 조회한다.

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;
    }
}
```

2.36 dbmSelectForUpdateRow

기능

dbmSelectRow와 동일하며 조회된 레코드를 LOCK 한다.

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 );
}
```

다른 갱신 DML들과 마찬가지로, dbmSelectForUpdateRow를 수행한 경우 반드시 dbmCommit 이나 dbmRollback을 호출하여 트랜잭션을 종료해야 한다.

레코드 lock을 유지하는 API 종류는 다음과 같다.

- dbmSelectForUpdateRowGT
- dbmSelectForUpdateRowLT
- dbmFetchNextUpdateRowGT
- dbmFetchNextUpdateRowLT

2.37 dbmInsertArray

기능

N 개의 레코드를 삽입할 때 사용한다. N 개의 operation 중에 오류가 한 건 이상 발생하면 에러를 반환하는데 오류가 발생한 경우에도 operation은 진행되며 모든 operation이 완료된 후에 결과가 반환된다.

인자

```
int dbmInsertArray( dbmHandle           * aHandle,
                    char                * aTableName,
                    void                * aUserData,
                    int                   aDataSingleSize,
                    int                   aDataCount,
                    int                   aRetArr[] )
```

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.
aDataSingleSize	int	In	데이터 한 개의 크기를 지정한다.
aDataCount	int	In	aUserData에 담긴 데이터 개수를 지정한다.
aRetArr	int	Out	각 operation의 에러코드를 순서대로 담는다.

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

```
rc = dbmInsertArray( sHandle, "t1", sData, sizeof(DATA), 10, sErrCode );
    if( rc ) exit(-1);
    rc = dbmCommit( sHandle );
    if( rc ) exit(-1);
```

- 에러 처리

```
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;
}
```

사용자가 입력한 데이터 한 개의 크기와 개수에 따라 설정해야 하는 사용자 버퍼가 잘못 설정되면 올바르지 않은 포인터가 접근할 수 있으므로 주의해서 사용해야 한다.

2.38 dbmUpdateArray

기능

N 개의 레코드를 변경할 때 사용한다. N 개의 operation 중에 오류가 한 건 이상 발생하면 에러를 반환하는데 오류가 발생한 경우에도 operation은 진행되며 모든 operation이 완료된 후에 결과가 반환된다.

인자

```
int dbmUpdateArray( dbmHandle          * aHandle,
                    char               * aTableName,
                    void               * aUserData,
                    int                  aDataCount,
                    int                * aAffectedRowCount,
                    int                  aRetArr[] )
```

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.
aDataCount	int	In	aUserData에 담긴 개수를 지정한다.
aAffectedRowCount	int	Out	전체 처리된 건수를 반환한다.
aRetArr	int	Out	각 operation의 에러코드를 순서대로 담는다.

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;
}
```

사용자가 입력한 데이터 한 개의 크기와 개수에 따라 설정해야 하는 사용자 버퍼가 잘못 설정되면 올바르지 않은 포인터가 접근할 수 있으므로 주의해서 사용해야 한다.

2.39 dbmSelectArray

기능

N 개의 레코드를 조회할 때 사용한다. N 개의 operation 중에 오류가 한 건 이상 발생하면 에러를 반환하는데 오류가 발생한 경우에도 operation은 진행되며 모든 operation이 완료된 후에 결과가 반환된다.

인자

```
int dbmSelectArray( dbmHandle           * aHandle,
                    char                * aTableName,
                    void                * aUserData,
                    int                   aDataCount,
                    int                 * aAffectedRowCount,
                    int                   aRetArr[] )
```

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.
aDataCount	int	In	aUserData에 담긴 개수를 지정한다.
aAffectedRowCount	int	Out	전체 처리된 건수를 반환한다.
aRetArr	int	Out	각 operation의 에러코드를 순서대로 담는다.

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;
}
```

사용자가 입력한 데이터 한 개의 크기와 개수에 따라 설정해야 하는 사용자 버퍼가 잘못 설정되면 올바르지 않은 포인터가 접근할 수 있으므로 주의해서 사용해야 한다.

2.40 dbmDeleteArray

기능

N 개의 레코드를 삭제할 때 사용한다. N 개의 operation 중에 오류가 한 건 이상 발생하면 에러를 반환하는데 오류가 발생한 경우에도 operation은 진행되며 모든 operation이 완료된 후에 결과가 반환된다.

인자

```
int dbmDeleteArray( dbmHandle          * aHandle,
                    char               * aTableName,
                    void               * aUserData,
                    int                  aDataCount,
                    int                * aAffectedRowCount,
                    int                  aRetArr[] )
```

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.
aDataCount	int	In	aUserData에 담긴 개수를 지정한다.
aAffectedRowCount	int	Out	전체 처리된 건수를 반환한다.
aRetArr	int	Out	각 operation의 에러코드를 순서대로 담는다.

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;
}
```

사용자가 입력한 데이터 한 개의 크기와 개수에 따라 설정해야 하는 사용자 버퍼가 잘못 설정되면 올바르지 않은 포인터가 접근할 수 있으므로 주의해서 사용해야 한다.

2.41 dbmEnqueue

기능

사용자 데이터를 queue 형식의 테이블에 삽입한다.

- 한 개의 테이블에 여러 형태의 메시지를 이용하려면 msg-type을 임의의 숫자로 지정한다.
- Priority가 더 낮을 수록 우선순위가 높다.
- Msg-type 이나 priority 등이 설정되지 않은 경우에는 모두 0으로 설정된다.

인자

```
int dbmEnqueue( dbmHandle          * aHandle,
                char               * aTableName,
                int                  aMsgType,
                int                  aPriority,
                char               * aUserData,
                int                  aDataSize )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 queue 테이블 이름이다.
aMsgType	int	In	0 이상의 사용자 정의 type 이다.
aPriority	int	In	0 이상의 사용자 정의 priority 이다.
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",
                     101,
                     1,
                     & sData,
                     sizeof(DATA) );

}
```

2.42 dbmDequeue

기능

Queue table로부터 한 건의 사용자 데이터를 추출한다.
(MsgType, Priority)를 지정할 경우, 해당 조건에 맞는 데이터를 dequeue 한다. 조건없이 가져오려면 두 개의 필드를 모두 (-1, -1)로 설정하면 된다.
Dequeue 된 데이터는 자동으로 삭제되며 rollback이 호출된 경우 원본 데이터 형태로 그대로 삽입된다. (MsgType과 priority가 모두 유지되고 삽입 시간만 rollback time으로 변경된다.)

인자

```
int dbmDequeue( dbmHandle          * aHandle,
                char               * aTableName,
                int                  aInMsgType,
                int                  aInPriority,
                int                * aOutMsgType,
                int                * aOutPriority,
                char               * aUserData,
                int                * aDataSize,
                int                  aTimeout  )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 queue 테이블 이름이다.
aInMsgType	int	In	Dequeue를 수행할 Msg-Type을 특정할 경우 해당 값을 입력하고 그렇지 않을 경우 -1을 입력한다.
aInPriority	int	In	Dequeue를 수행할 priority를 특정할 경우 해당 값을 입력하고 그렇지 않을 경우 -1을 입력한다. (입력된 priority와 같거나 큰 대상을 반환한다.)
aOutMsgType	int *	Out	해당 메시지의 Msg-Type 정보를 반환한다. (NULL일 경우에는 반환하지 않는다.)
aOutPriority	int *	Out	해당 메시지의 우선순위 정보를 반환한다. (NULL일 경우에는 반환하지 않는다.)
aUserData	void *	Out	사용자 변수 포인터이다.
aDataSize	int *	Out	aUserData의 크기이다.
aTimeout	int	In	Queue에 데이터가 없을 경우 대기하는 시간을 지정 (ms단위) 한다.

사용 예

```
typedef struct
{
    int c1;
    int c2;
} DATA;
main()
{
    dbmHandle    * sHandle = NULL;
    DATA           sData;
    int            sDataSize = 0;
    int            sMsgType;
    int            sPriority;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmDequeue( sHandle,
                     "que1",
                     101,
                     1,
                     &sMsgType,
                     &sPriority,
                     & sData,
                     & sDataSize,
                     1000 );

}
```

- Queue에는 (MsgType, Priority, MsgID) 순으로 조합, 정렬되어 저장된다.
- MsgType과 priority가 지정되지 않으면 MsgID를 채번한 순서대로 출력된다.
- MsgType이 동일할 경우 priority가 가장 작은 순서대로, priority가 동일할 경우에는 MsgID 순서대로 출력된다.
- Priority를 입력할 경우 priority 값이 입력된 값 이상인 것들만 출력된다.

2.43 dbmGetCurrVal

기능

미리 생성되어 있는 sequence 객체의 현재값을 반환한다.

인자

```
int dbmGetCurrVal( dbmHandle          * aHandle,
                   char               * aSequenceName,
                   long long          * aCurrVal )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 queue 테이블 이름이다.
aCurrVal	long long int	Out	반환받을 포인터 (8 byte 변수 필요) 이다.

사용 예

```
main()
{
    dbmHandle     * sHandle = NULL;
    long long       sCurrVal;
    int             sDataSize = 0;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmGetCurrVal( sHandle,
                        "seq1",
                        & sCurrVal );
}
```

Sequence 객체에 대해 nextval이 호출되지 않은 상태에서 currval을 호출할 경우 오류가 발생한다.

2.44 dbmGetNextVal

기능

미리 생성되어 있는 sequence 객체의 다음 값을 반환한다.

인자

```
int dbmGetNextVal( dbmHandle          * aHandle,
                   char               * aSequenceName,
                   long long          * aNextVal )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aTableName	char *	In	대상 queue 테이블 이름이다.
aNextVal	long long int	Out	반환받을 포인터 (8 byte 변수 필요) 이다.

사용 예

```
main()
{
    dbmHandle     * sHandle = NULL;
    long long       sNextVal;
    int             sDataSize = 0;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmGetNextVal( sHandle,
                        "seq1",
                        & sNextVal);
}
```

2.45 dbmCommit

기능

사용자가 수행한 모든 트랜잭션을 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 );
}
```

2.46 dbmRollback

기능

사용자가 수행한 트랜잭션을 모두 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 );
}
```

2.47 dbmRefineSystem

기능

사용자로 하여금 instance-level, table-level에서 table/index lock을 해제하거나 index를 재구축하는 복구 기능을 실행하도록 한다.

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 필수이다.
- TableName을 지정하면 대상 테이블만 복구하고, NULL을 입력하면 instance와 관련된 모든 테이블을 복구한다.

사용 예

```
if( argc > 1 )
    {
        if( dbmRefineSystem( "demo", NULL ) != 0 )
        {
            printf( "failed to refine all\n" );
            exit(-1);
        }
    else
    {
        if( dbmRefineSystem( "demo", "t1" ) != 0 )
        {
            printf( "failed to refine t1\n" );
            exit(-1);
        }
```

```
    }
```

Instance-level에서 수행할 때 대상 테이블이 아니면 skip 한다.

2.48 dbmGetRowCount

기능

UPDATE/ DELETE/ SELECT 기능을 prepare/ execute 한 후에 대상 row의 개수를 계산한다.

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, "...", &sStmt );
    rc = dbmExecuteStmt( sHandle, sStmt );

    rc = dbmGetRowCount( sHandle, sStmt, &sRowCount );
}
```

2.49 dbmGetRowSize

기능

입력한 테이블이 생성된 시점의 row size를 반환한다.

인자

```
int dbmGetRowSize( dbmHandle          * aHandle,
                   char               * aObjectName,
                   int                * aRowSize )
```

인자 항목	타입	In/ out	비고
aHandle	dbmHandle *	In	dbmInitHandle로 처리된 변수이다.
aObjectName	char *	In	조회할 object name 이다.
aRowSize	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 );
}
```

2.50 dbmGetTableName

기능

입력된 테이블 핸들로부터 table name을 반환한다.

인자

```
const char * dbmGetTableName( dbmTableHandle           * aHandle )
```

인자 항목	타입	In/ out	비고
aHandle	dbmTableHandle *	In	dbmPrepareTableHandle 로 처리된 변수이다.

사용 예

```
main()
{
    dbmHandle     * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    char   * sTableName;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareTableHandle( &sHandle, "t1", &sTableHandle );
    sTableName = dbmGetTableHandle ( sTableHandle );
}
```

TableHandle이 성공한 상태이어야 한다.

2.51 dbmGetTableType

기능

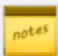
입력된 테이블 핸들로부터 table type을 반환한다.

인자

```
dbmTableType   dbmGetTableType( dbmTableHandle           * aHandle )
```

인자 항목	타입	In/ out	비고
aHandle	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,                // N/A
    DBM_TABLE_TYPE_SEQUENCE,             // Sequence Object
    DBM_TABLE_TYPE_DIRECT_TABLE,         // Direct Table
    DBM_TABLE_TYPE_DIRECT_QUEUE,         // N/A
    DBM_TABLE_TYPE_SPLAY_TABLE,          // Splay Index Table
    DBM_TABLE_TYPE_LIST_TABLE,           // N/A
    DBM_TABLE_TYPE_HASH_TABLE,           // Hash Table
    DBM_TABLE_TYPE_PERF_VIEW,
    DBM_TABLE_TYPE_MAX
} dbmTableType;
```



본 매뉴얼의 작성 시점인 3.2 버전 기준으로 사용자가 생성 가능한 유형은 다음과 같다.

- DBM_TABLE_TYPE_TABLE
- DBM_TABLE_TYPE_DIRECT_TABLE
- DBM_TABLE_TYPE_SPLAY_TABLE
- DBM_TABLE_TYPE_HASH_TABLE

사용 예

```
main()
{
    dbmHandle      * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    dbmTableType  sTableType;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareTableHandle( &sHandle, "t1", &sTableHandle );
    sTableType = dbmGetTableType ( sTableHandle );
}
```

2.52 dbmSetSplayMode4DML

기능

Splay table handle인 경우, 입력된 option에 따라 insert 할 때 tree를 splay 할 지 여부를 설정한다.
0 으로 설정하면 splay 하지 않는다.

인자

```
int dbmSetSplayMode4DML( dbmTableHandle            * aHandle,
                         int                         aMode )
```

인자 항목	타입	In/ out	비고
aHandle	dbmTableHandle *	In	dbmPrepareTableHandle 로 처리된 변수이다.
aMode	int	int	0 : no splay (default) 1 : do splay

사용 예

```
main()
{
    dbmHandle     * sHandle = NULL;
    dbmTableHandle * sTableHandle = NULL;
    rc = dbmInitHandle( &sHandle, "demo" );
    rc = dbmPrepareTableHandle( &sHandle, "t1", &sTableHandle );
    dbmSetSplayMode4DML( sTableHandle, 1 );
}
```

입력된 TableHandle은 SPLAY_TABLE_TYPE일 때만 동작한다.

2.53 dbmGetErrorData

기능

API 호출로 발생한 각 오류 코드에 대한 상세 에러 메시지를 확인한다.
DBM 내에는 에러가 stack 형태로 축적되어 있기 때문에 에러가 날 때까지 호출하면 최초 오류부터 발생 원인을 추적할 수 있다.

인자

```
int dbmGetErrorData( dbmHandle          * aHandle,
                     int                * aErrorCode,
                     char               * aErrorMsg )
```

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.

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 ) == 0 )
        {
            fprintf( stdout, "ERR-%d] %s\n", sErrCode, sErrMsg );
        }
    }
}
```

dbmGetErrorData를 통해 반환받는 에러 메시지 버퍼의 크기는 최소 512 byte 이상이어야 한다.

2.54 dbmGetErrorMsg

기능

API 호출로 발생한 각 오류 코드에 대한 상세 에러 메시지를 확인한다.
에러 stack에 저장된 에러의 원본 유형을 출력하기 때문에 상세한 부가 오류사항은 출력되지 않는다.

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) );
    }
}
```

2.55 dbmGetTableUsage

기능

특정 테이블의 segment 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	테이블의 확장된 상태를 포함하여 현재 저장 가능한 최대 row 개수이다.
aUsedSize	long *	Out	테이블에 현재 사용 중인 row 개수이다. (Commit 되지 않은 row를 포함할 수 있음)
aFreeSize	long *	Out	테이블 가용 공간의 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 );
}
```

2.56 dbmGetTableUsageByHandle

기능

주어진 table handle에 해당하는 테이블의 segment 정보를 반환한다.

Direct 테이블의 경우, 각 segment 별로 데이터를 가진 최대 slot ID를 구하여 계산되므로 중간에 빈 slot이 있을 경우에도 사용 중인 slot으로 계산될 수도 있다.

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 테이블의 핸들이다.
aMaxSize	long *	Out	테이블을 생성할 때 지정한 최대 row 개수이다.
aTotalSize	long *	Out	테이블의 확장된 상태를 포함하여 현재 저장 가능한 최대 row 개수이다.
aUsedSize	long *	Out	테이블에 현재 사용 중인 row 개수이다. (Commit 되지 않은 row를 포함할 수 있음)
aFreeSize	long *	Out	테이블 가용 공간의 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 = dbmGetTableUsage( sHandle,
                           sTableHandle,
                           &sMax,
                           &sTotal,
                           &sUsed,
                           &sFree );
}
```

해당 값은 스냅샷이므로 테이블에 삽입/ 삭제가 발생하는 도중에는 그 값이 정확하지 않다.
DIRECT TABLE은 마지막으로 입력된 최종 위치만 기록하기 때문에 중간에 삭제될 경우에는 값이 정확하지 않을 수 있다.

2.57 dbmExtendTable

기능

주어진 table handle에 해당하는 테이블에 새 segment를 추가한다.
새 segment의 크기는 CREATE TABLE 시 지정된 extend size 이다.

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 );
}
```

- Extend에 의해 확장 가능한 segment의 최대 개수는 999 개이다.
- Extend가 빈번하게 발생할 경우 삽입 성능이 저하될 수 있으므로 table 생성 시점에 init 크기를 적절

> 하게 설정해야 한다.

2.58 dbmExistDataInQue

기능

지정된 queue table에 데이터가 존재하는지 여부를 반환한다.

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 );
}
```

일반 테이블에 대해서는 사용할 수 없고 queue type 테이블만 지원한다.

2.59 dbmNow

기능

현재 시각을 반환한다.

인자

```
unsigned long dbmNow( void )
```

사용 예

```
#include <dbmUserAPI.h>
#include <common.h>
typedef struct
{
    int c1;
    unsigned long c2;
} DATA;
int main()
{
    DATA   sData;

    sData.c1 = 1;
    sData.c2 = dbmNow();
    rc = dbmInsertRow( sHandle, "t1", &sData, sizeof(sData) );
    TEST_ERR( sHandle, rc, "InsertFail" );
    sData.c1 = 1;
    rc = dbmSelectRow( sHandle, "t1", &sData );
    TEST_ERR( sHandle, rc, "SelectFail" );
    tt =sTimeVal.tv_sec;
    nowtm = localtime( &tt );
    strftime( buf, sizeof(buf), "%Y-%m-%d %H:%M:%S", nowtm);
    printf( "fetchData: c1=%d, c2=%s.%06ld\n", sData.c1, buf, sTimeVal.tv_usec );
}
```

2.60 Error Message

GOLDILOCKS LITE에 정의된 에러 메시지는 다음 표와 같다.

Error defined name	Error code	Detail message	Description
DBM_ERRCODE_INVALID_ARGS	70001	fail to validate some parameters at internal processing	-
DBM_ERRCODE_MEMORY_NOT_SUFFICIENT	70002	fail to alloc memory from OS (errno=%d)	-
DBM_ERRCODE_FAIL_TO_ALLOC_MEMORY	70003	fail to alloc memory from dbmAllocator	-
DBM_ERRCODE_NOT_IMPL	70004	not implemented	-
DBM_ERRCODE_ALREADY_SHM_EXIST	70005	a shared memory already exists	-
DBM_ERRCODE_CREATE_SHM_FAIL	70006	fail to create a shared memory segment	-
DBM_ERRCODE_INIT_SHM_FAIL	70007	fail to initialize a shared memory segment	-
DBM_ERRCODE_ATTACH_SHM_FAIL	70008	fail to attach a shared memory segment	-
DBM_ERRCODE_SHM_OPEN_FAIL	70009	fail to open a shared memory	-
DBM_ERRCODE_SHM_FSTAT_FAIL	70010	fail to get a information of shm	-
DBM_ERRCODE_SHM_INVALID_SIZE	70011	invalid segment size to attach a shm	-
DBM_ERRCODE_MMAP_FAIL	70012	fail to call a mmap to attach a shared memory segment	-
DBM_ERRCODE_DETACH_SHM_FAIL	70013	fail to detach a shared memory segment	-
DBM_ERRCODE_DROP_FAIL	70014	fail to drop a shared segment memory	-
DBM_ERRCODE_CREATE_SHM_DIR_FAIL	70015	fail to create a directory for shared-memory	-
DBM_ERRCODE_INVALID_SLOT_NO	70016	invalid slot number (SlotId=%ld)	-
DBM_ERRCODE_NO_EXIST_DIC	70017	fail to attach dictionary (execute initdb)	-
DBM_ERRCODE_NOT_DEF_INSTANCE	70018	a operation not allowed without instance	-
DBM_ERRCODE_NOT_EXIST_TABLE	70019	(%s) table not exists	-
DBM_ERRCODE_NOT_EXIST_COLUMN	70020	(%s) Column not exists	-

Error defined name	Error code	Detail message	Description
DBM_ERRCODE_MAX_SEGMENT	70021	a segment has no space to extend because of reached max_segment	-
DBM_ERRCODE_NO_SPACE	70022	a segment has no space to extend because of reached max_size	-
DBM_ERRCODE_CONNECT_FAIL	70023	fail to connect target server	-
DBM_ERRCODE_SEND_FAIL	70024	fail to send a packet	-
DBM_ERRCODE_RECV_FAIL	70025	fail to receive a packet	-
DBM_ERRCODE_HB_FAIL	70026	fail to send or receive a packet for HB	-
DBM_ERRCODE_INIT_HANDLE_FAIL	70027	fail to initialize a handle	-
DBM_ERRCODE_ALLOC_HANDLE_FAIL	70028	fail to alloc a memory for handle	-
DBM_ERRCODE_NEED_VALUE_NULL	70029	a pointer have to be set null to initialize a handle	-
DBM_ERRCODE_FREE_HANDLE_FAIL	70030	fail to free a handle	-
DBM_ERRCODE_ALLOC_STMT_FAIL	70031	fail to alloc a memory for statement	-
DBM_ERRCODE_INIT_PARSE_CTX_FAIL	70032	fail to alloc a memory for parser-context	-
DBM_ERRCODE_EXECUTE_FAIL	70033	fail to execute a statement	-
DBM_ERRCODE_INVALID_STMT_TYPE	70034	invalid stmt type	-
DBM_ERRCODE_INVALID_PLAN_TYPE	70035	invalid plan type	-
DBM_ERRCODE_INVALID_DATA_TYPE	70036	invalid data type	-
DBM_ERRCODE_INVALID_TABLE_SIZE_OPTION	70037	invalid table size option	-
DBM_ERRCODE_PREPARE_FAIL	70038	"fail to prepare a statement	-
DBM_ERRCODE_FREE_STMT_FAIL	70039	fail to finalize a stmt	-
DBM_ERRCODE_INVALID_EXPR_TYPE	70040	invalid expr type	-
DBM_ERRCODE_ALLOC_MEM_FAIL	70041	fail to alloc a memory for something	-
DBM_ERRCODE_INVALID_BUILT_FUNC	70042	invalid built-in function	-
DBM_ERRCODE_INVALID_SEGMENT	70043	invalid segment	-
DBM_ERRCODE_ALLOC_TRANS	70044	fail to alloc a trans for current-session	-

Error defined name	Error code	Detail message	Description
_FAIL			
DBM_ERRCODE_DATA_COUNT_MISMATCH	70045	the number of binding-data mismatch to target-list	-
DBM_ERRCODE_INVALID_COLUMN	70046	(%s) column not exists	-
DBM_ERRCODE_INVALID_EXPR	70047	invalid expression type	-
DBM_ERRCODE_CONVERT_DATA_FAIL	70048	fail to convert a data as invalid data-type or value-size or origin-value etc.	-
DBM_ERRCODE_BINDING_COL_FAIL	70049	fail to bind a column (%s)	-
DBM_ERRCODE_CONVERT_OVERFLOW	70050	fail to convert data as overflow	-
DBM_ERRCODE_NO_MORE_DATA	70051	no more data to fetch	-
DBM_ERRCODE_DIVIDE_BY_ZERO	70052	a operation can not be executed because of divide by zero	-
DBM_ERRCODE_INVALID_GROUP_BY	70053	invalid group-by or target-list to execute group-by	-
DBM_ERRCODE_NOT_EXIST_INDEX	70054	index not exist (%s)	-
DBM_ERRCODE_INDEX_DUPLICATED	70055	index key value duplicated (%s)	-
DBM_ERRCODE_INDEX_KEY_NOT_FOUND	70056	index key not found (%s)	-
DBM_ERRCODE_INVALID_LOG_TYPE	70057	invalid log type	-
DBM_ERRCODE_DUP_COLUMN_NAME	70058	(%s) column duplicated	-
DBM_ERRCODE_INVALID_DATA_SIZE	70059	invalid data size (Limit=%d : InputSize=%d)	-
DBM_ERRCODE_CHANGE_SCN_FAIL	70060	fail to change SCN of row (Segment=%s, SlotId=%ld)	-
DBM_ERRCODE_INVALID_SCN	70061	invalid scn (SCN=%ld)	-
DBM_ERRCODE_COMMIT_PROC_FAIL	70062	fail to process a function to commit (log=%s)	-
DBM_ERRCODE_ROLLBACK_PROC_FAIL	70063	"fail to process a function to rollback (log=%s)	-
DBM_ERRCODE_DUP_INDEX_KEY_COLUMN	70064	a index with same ordering key was already created (%s)	-
DBM_ERRCODE_DUP_COLUMN_DEFINED	70065	a column definition duplicated (%s)	-
DBM_ERRCODE_OPEN_DISK_LO			

Error defined name	Error code	Detail message	Description
G_FAIL	70066	fail to open a disk logfile (%s) (errno=%d)	-
DBM_ERRCODE_LSEEK_DISK_LOG_FAIL	70067	fail to locate a position of disk logfile (%s) (errno=%d)	-
DBM_ERRCODE_SWITCH_DISK_LOG_FAIL	70068	fail to switch a disk logfile	-
DBM_ERRCODE_WRITE_DISK_LOG_FAIL	70069	fail to write a disk logfile (errno=%d)	-
DBM_ERRCODE_FSYNC_DISK_LOG_FAIL	70070	fail to sync a disk logfile (errno=%d)	-
DBM_ERRCODE_READ_DISK_LOG_FAIL	70071	fail to read from a disk logfile (errno=%d)	-
DBM_ERRCODE_INVALID_DISK_LOG	70072	invalid disk log block	-
DBM_ERRCODE_INVALID_TABLE_TYPE	70073	a operation can not be executed on target-table (check table type)	-
DBM_ERRCODE_NOT_ALLOWED_COLUMN	70074	column(ID, INTIME) can not be specified at target-list when target table is queue-type	-
DBM_ERRCODE_INVALID_INDEX_STAT	70075	a index (%s) invalid stat, need to rebuild index	-
DBM_ERRCODE_INVALID_TRY	70076	not supported transaction	-
DBM_ERRCODE_INST_ALREADY_EXISTS	70077	a instance already exists	-
DBM_ERRCODE_INDEX_ALREADY_EXISTS	70078	a index already exists	-
DBM_ERRCODE_ALREADY_EXISTS_TABLE	70079	a table already exists	-
DBM_ERRCODE_DEAD_LOCK_DETECT	70080	a dead-lock detection	-
DBM_ERRCODE_TOO_LONG_NAME	70081	a length of object too long (max %d bytes)	-
DBM_ERRCODE_INVALID_BIND_PARAM	70082	invalid binding parameters (index or name not exist)	-
DBM_ERRCODE_MISMATCH_BIND_COL	70083	invalid binding column count	-
DBM_ERRCODE_NEED_DICT_HANDLE	70084	this operation can be executed by a dictionary handle.	-
DBM_ERRCODE_NOT_EXISTS_INST	70085	a instance not exists	-
DBM_ERRCODE_INVALID_KEY_DATA_TYPE	70086	a index key column must have a data type as (long, char, int, short)	-
DBM_ERRCODE_TIMEOUT	70087	a timeout raised on this operation	-
DBM_ERRCODE_NOT_ALLOWE		this operation not allowed at current-instanc	

Error defined name	Error code	Detail message	Description
D_OPERATION	70088	e	-
DBM_ERRCODE_TOO_BIG_ROWSIZE	70089	a total size of columns is too big to create	-
DBM_ERRCODE_NEED_COMMIT_OR_ROLLBACK	70090	fail to free a statement variable as transaction not completed	-
DBM_ERRCODE_TOO_BIG_TO_WRITE_LOG	70091	a log-size is too big to write a transaction log	-
DBM_ERRCODE_FAIL_TO_PARSE	70092	fail to parse a syntax	-
DBM_ERRCODE_NEED_INDEX	70093	a operation via API need a index	-
DBM_ERRCODE_INVALID_SEQ_OPTION	70094	a invalid number or range for sequence	-
DBM_ERRCODE_SEQUENCE_MAXVALUE	70095	a sequence reached at max-value	-
DBM_ERRCODE_SEQUENCE_NOT_DEF_CURRVAL	70096	a currval of sequence not yet defined (need to call nextval)	-
DBM_ERRCODE_NOT_ENOUGH_BUFF	70097	not enough buffer size	-
DBM_ERRCODE_INVALID_LICENSE	70098	invalid license	-
DBM_ERRCODE_INVALID_OFFSET	70099	invalid offset	-
DBM_ERRCODE_TOO_MANY_ROWS	70100	too many rows	-
DBM_ERRCODE_CHECK_DIC_FAIL	70101	fail to check dictionary"	-
DBM_ERRCODE_THREAD_FAIL	70102	fail to invoke a thread	-
DBM_ERRCODE_FILE_READ_FAIL	70103	fail to read	-
DBM_ERRCODE_FILE_WRITE_FAIL	70104	fail to write	-
DBM_ERRCODE_NOT_ACTIVE_INSTANCE	70105	a instance not active-mode	-
DBM_ERRCODE_DIRECT_INVALID_KEY_DATA_TYPE	70106	a index key column must have a data type as (long, int, short)	-
DBM_ERRCODE_DIRECT_NEED_INDEX	70107	at first, need to create a index to use a direct table	-
DBM_ERRCODE_FAIL_TO_PREPARE_DISK_LOG	70108	fail to prepare a disk logfile	-
DBM_ERRCODE_FAIL_TO_PREPARE_REPL	70109	fail to prepare replication	-
DBM_ERRCODE_FAIL_TO_PREP			

Error defined name	Error code	Detail message	Description
ARE_TABLE	70110	fail to prepare a table	-
DBM_ERRCODE_NOT_FOUND	70111	no data found	-
DBM_ERRCODE_REPL_NOT_CONNECTED	70112	a replication-session not connected	-
DBM_ERRCODE_TOO_MANY_RESULT	70113	a result-set has too many rows to process	-
DBM_ERRCODE_NOT_EXIST_PROC	70114	a procedure not found	-
DBM_ERRCODE_ALREADY_EXIST_PROC	70115	a procedure already exists	-
DBM_ERRCODE_INVALID_IDENTIFIER	70116	invalid identifier	-
DBM_ERRCODE_CASE_NOT_FOUND	70117	case not found	-
DBM_ERRCODE_CURSOR_ALREADY_OPENED	70118	a cursor already opened	-
DBM_ERRCODE_CURSOR_NOT_OPENED	70119	a cursor not opened	-
DBM_ERRCODE_EXCEPTION_DUPLICATED	70120	a exception duplicated(Line=%d,Column=%d)	-
DBM_ERRCODE_RAISE_USER_EXCEPTION	70121	a user exception raised	-
DBM_ERRCODE_UNHANDLE_EXCEPTION	70122	unhandled exceptions	-
DBM_ERRCODE_PREPARE_PROCEDURE	70123	fail to prepare a object/statement of procedure (Line=%d, Column=%d)	-
DBM_ERRCODE_EXIT_ONLY_AT_LOOP	70124	a exit/continue statement is able to be used in loop-statement	-
DBM_ERRCODE_EXECUTE_PROC_FAIL	70125	fail to execute a procedure statement (Line=%d)	-
DBM_ERRCODE_CHANGED_PLAN	70126	changed index after dbmPrepareStmt	-
DBM_ERRCODE_ALREADY_ATTACH_TID	70127	current thread-id already attached at (Trans=%d)	-
DBM_ERRCODE_TOO_MANY_SEGMENT_EXTEND	70128	a count of segment expected too many chunk. (need less than 999)	-
DBM_ERRCODE_GET_SEMAPHORE	70129	error get semaphore (id=%ld)	-
DBM_ERRCODE_CURSOR_API_ALREADY_OPENED	70130	open cursor api already executed	-
DBM_ERRCODE_CURSOR_API_ALREADY_CLOSED	70131	close cursor api already executed	-

Error defined name	Error code	Detail message	Description
DBM_ERRCODE_DDL_RAISED	70132	a handle of table re-prepared as ddl executed	-
DBM_ERRCODE_BEGIN_TRANS_STAT	70133	a operation can not be executed as other transaction (transId=%d) already began	-
DBM_ERRCODE_NEED_NO_TX_AT_DDL	70134	a operation can not be executed as previous transaction need commit or rollback	-
DBM_ERRCODE_ALREADY_EXIST_LIB	70135	a library already exists	-
DBM_ERRCODE_NOT_EXIST_LIB	70136	a function not exists	-
DBM_ERRCODE_EXECUTE_USER_FUNC_FAIL	70137	fail to execute a user function (%s:RetCode=%d)	-
DBM_ERRCODE_INVALID_TIME_OPTION	70138	invalid time option	-
DBM_ERRCODE_PORT_OUT_OF_RANGE	70139	port out of range	-
DBM_ERRCODE_CLIENT_MAX_OUT_OF_RANGE	70140	client max out of range	-
DBM_ERRCODE_PROCESS_MAX_OUT_OF_RANGE	70141	process max out of range	-
DBM_ERRCODE_PROCESS_MIN_OUT_OF_RANGE	70142	process min out of range	-
DBM_ERRCODE_PROCESS_CNT_OUT_OF_RANGE	70143	process count out of range	-
DBM_ERRCODE_QUEUE_SIZE_OUT_OF_RANGE	70144	queue size out of range	-
DBM_ERRCODE_GSB_CREATE_FAIL	70145	create gsb failed	-
DBM_ERRCODE_GSB_DROP_FAIL	70146	drop gsb failed	-
DBM_ERRCODE_INVALID_JSON_KEY_VALUE	70147	a key value has not to be json-object or array	-
DBM_ERRCODE_INVALID_JSON_VALUE	70148	invalid json key-string or valueOrType	-
DBM_ERRCODE_ALREADY_EXISTS_REPL	70149	a replication name already exists	-
DBM_ERRCODE_INVALID_DIRECT_TABLE_INDEX	70150	a column is not valid as index in direct-table	-
DBM_ERRCODE_INVALID_REPL_DIR	70151	a value of unsent_dir property is not matched between anchor-file and property-file	-
DBM_ERRCODE_INVALID_PROP	70152	a property(%s) is not found or invalid value	-
DBM_ERRCODE_NEED_JOIN_INDEX	70153	a join table need index	-

Error defined name	Error code	Detail message	Description
DBM_ERRCODE_DDL_NOT_ALLOWED_IN_REPL	70154	a DDL not allowed as a table involved in replication	-
DBM_ERRCODE_NEED_INDEX_ON_OPERATION	70155	a operation can not executed as some table need unique-index"	-
DBM_ERRCODE_ODBC_CALL_FAIL	70156	fail to call ODBC_LIB (Detail:%s)	-
DBM_ERRCODE_NOT_EXIST_DSN	70157	a dsn not exists	-
DBM_ERRCODE_ODBC_LIB_OPEN_FAIL	70158	fail to open odbc-library	-
DBM_ERRCODE_ODBC_GET_SYMBOL_FAIL	70159	fail to get a function symbol of mapping ODBC API	-
DBM_ERRCODE_INVALID_JSON_KEY_SIZE	70160	a json key is too long	-
DBM_ERRCODE_ERROR_HTTP	70161	http failed	-
DBM_ERRCODE_INVALID_LIMIT_OPTION	70162	invalid limit option	-
DBM_ERRCODE_NOT_ALLOWED_UPDATE	70163	a update operation not allowed on a record with expired-time	-

3.

Stored Procedure

3.1 개요

Stored procedure는 사용자 업무 절차를 SQL을 통해 작성하여 메모리에 저장한 후 이를 호출하여 결과를 만들어내는 방식을 의미한다.

기본적으로 원격 서버에서 한 개의 트랜잭션 내에 다수의 API를 호출하여 발생하는 network I/O 비용을 고려할 때 procedure를 통해 호출하는 방법이 유리한 경우에 stored procedure를 사용할 것을 권장한다.
지역 서버 내에서도 procedure를 호출할 수 있으나 내부적으로 SQL을 처리하는 과정에서 변수와 결과에 대한 많은 expression 처리 비용이 실제 API를 호출하는 비용보다 클 수 있기 때문에 단순한 operation에 대해서는 API에 비해 빠르지 않을 수 있음을 고려해야 한다.

기능

Procedure 내에 변수를 선언하거나 parameter를 사용할 수 있으며 다음과 같은 데이터 타입이 제공된다.

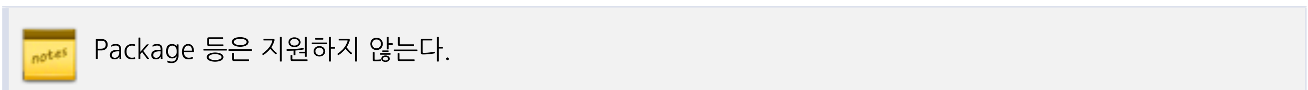
Data type	C/C++ Type	설명
int	int	4 bytes
long	long long	8 bytes
short	short	2 bytes
float	float	4 bytes
double	double	8 bytes
char	char	1 byte (MAX: 30 K)
TableName.ColumnName%TYPE VariableName%TYPE	명시된 object 타입과 동일	Procedure 내에서 사전에 선언된 variable만 사용 가능
TableName%ROWTYPE	명시된 TableName의 구조와 동일	-
EXCEPTION	사용자 exception 정의	-
CURSOR	사용자 cursor 정의	Cursor declaration은 제공하지 않음

Ref cursor, cursor variable, user defined data type 등은 지원하지 않는다.

GOLDILOCKS LITE는 다음 표와 같이 procedure 내의 기능 구문을 지원한다.

기능	지원 여부	설명
LOOP	O	-
FOR LOOP	O	-
WHILE LOOP	O	-
IF	O	-
EXIT (WHEN)	O	-

기능	지원 여부	설명
CONTINUE (WHEN)	O	-
GOTO	O	-
SIMPLE CASE	O	-
SEARCHED CASE	O	-
EXCEPTION WHEN	O	-
ASSIGN	O	-
INSERT INTO	O	Array 형태를 제공하지 않음
UPDATE SET	O	Array 형태를 제공하지 않음
DELETE FROM	O	Array 형태를 제공하지 않음
SELECT ~ INTO	O	Array 형태를 제공하지 않음
OPEN <Cursor>	O	-
FETCH <Cursor> INTO	O	Array 형태를 제공하지 않음
CLOSE <Cursor>	O	-
DBMS_OUTPUT.PUT_LINE	O	화면 로그 출력용


Package 등은 지원하지 않는다.

GOLDILOCKS LITE에서 SQL과 cursor attribute 변수를 사용하기 위해 다음 표의 변수가 자동으로 생성되어 있다.

구분	이름	초기값	설명
SQL attribute	SQLCODE	0	SQL 수행에 따른 에러가 발생하였거나 exception에 의해 에러가 발생한 경우, 에러 코드가 설정된다.
	SQLERRM	NULL	SQL 수행에 따른 에러가 발생하였거나 exception에 의해 에러가 발생한 경우, 에러 코드가 설정된다.
	SQL%ISOPEN	0	SQL 문에 어떤 결과든 한 건 이상 존재할 경우, 1로 설정된다.
	SQL%FOUND	0	SQL 문에 의해 수행된 결과가 있을 경우, 1로 설정된다.
	SQL%NOTFOUND	1	SQL 문에 의해 수행된 결과가 없을 경우, 1로 설정된다.
	SQL%ROWCOUNT	0	SQL 문을 수행한 이후에 영향을 받은 row의 개수이다.
Cursor attribute	CursorName%ISOPEN	0	CURSOR가 OPEN 된 경우 1로 설정되고, CLOSE 된 경우 0으로 설정된다.
	CursorName%FOUND	0	CURSOR 수행 시점에 데이터가 존재하는 경우, 1로 설정된다.
	CursorName%NOTFOUND	1	CURSOR 수행 시점에 데이터가 없는 경우, 1로 설정된다.
	CursorName%ROWCOUNT	0	FETCH가 호출될 때마다 1씩 증가한다.

3.2 PROCEDURE DDL

본 장에서는 procedure를 생성하는 방법에 대해 설명한다. GOLDILOCKS LITE에서는 수행 중인 procedure에 대해 DDL을 수행할 수 있는데, 이 때 기존 procedure를 실행 계획으로 하여 이미 구동된 프로세스는 변경된 procedure 내용을 감지하지 않기 때문에 관련 응용 프로그램들을 모두 정지시킨 후에 해당 DDL을 수행해야 한다.

Create [or replace] Procedure

```
create_or_replace_procedure ::= CREATE [ OR REPLACE ] PROCEDURE proc_name
                                param_declare_section
                                IS
                                declare_section
                                proc_block
                                /
                                ;
proc_name ::= Procedure 이름
param_declare_section ::= ( variable_name [IN¦OUT¦INOUT] data_type [, ... ] )
declare_section ::= ( variable_name declare_type; [ ... ] )
variable_name ::= Procedure 내에서 사용할 변수의 이름
declare_type ::=    INT
                  ¦ SHORT
                  ¦ DOUBLE
                  ¦ LONG
                  ¦ FLOAT
                  ¦ CHAR (size)
                  ¦ table_name.column_Name%TYPE
                  ¦ variable_name%TYPE
                  ¦ table_name%ROWTYPE
                  ¦ EXCEPTION
proc_block ::= BEGIN
                  proc_stmt
              END
proc_stmt ::= GOLDILOCKS LITE가 제공하는 procedure 구문들
```

다음은 procedure를 생성하는 예이다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1 (a1 in int )
IS
```

```
c1 INT;
BEGIN
   a1 := c1;
END;
/
success
dbmMetaManager(DEMO)>
```

Procedure가 정상적으로 생성되었을 경우, 다음과 같은 방법을 사용하여 dictionary instance가 있는지 여부를 확인해 볼 수 있다.

```
dbmMetaManager(DEMO)> set instance dict;
success
dbmMetaManager(DICT)> select * from dic_procedure;
--------------------------------------------------------------------------------
INST_NAME            : DEMO
PROC_NAME            : PROC1
PIECE_COUNT          : 1
--------------------------------------------------------------------------------
1 row selected
```

dbmMetaManager의 -f 옵션을 통해서만 생성할 수 있다. 또한 CREATE ~ END; / 절에서 END 와 문장의 끝을 의미하는 / 의 사이나 / 이후에는 다른 문자를 허용하지 않는다.

Drop Procedure

기존에 있던 procedure를 제거한다. Drop procedure에 의해 제거된 procedure를 수행 중인 세션이 감지할 수 없기 때문에 응용 프로그램을 중지한 후에 procedure를 제거하는 것이 좋다.

```
DROP PROCEDURE ::= DROP PROCEDURE <procedure_name>;
procedure_name := 삭제할 procedure 이름
```

```
dbmMetaManager(DEMO)> DROP PROCEDURE proc1;
success
```

Procedure View

Procedure의 정보를 확인하기 위해 다음의 두 view들을 참조할 수 있다.

dbm_procedure

Procedure의 기본 정보를 담고 있다.

Column name	Type	설명
INST_NAME	CHAR	Procedure가 속한 instance name
PROC_NAME	CHAR	Procedure name
PIECE_COUNT	CHAR	dbm_procedure_text에는 구문 전체가 나뉘어 저장되는데 이 때 나뉘어진 조각의 개수

dbm_procedure_text

Procedure를 생성한 시점에 사용된 구문 전체를 저장한다.

Column name	Type	설명
INST_NAME	CHAR	Procedure가 속한 instance name
PROC_NAME	CHAR	Procedure name
PIECE_ID	INT	저장된 조각의 순서 번호
PROC_TEXT	CHAR	Procedure syntax

3.3 PROCEDURE Language Elements

본 장에서는 procedure에서 제공되는 각 구성 요소와 기능에 대해 설명한다.

Declare Section

Procedure의 block 단위에서 사용할 수 있는 element들의 선언부이다.

Variable

SCALAR TYPE

GOLDILOCKS LITE에서 사용되는 변수 중에 scalar data type에 대한 변수를 선언한다.

```
scalar typed variable ::= variable_name data_type ( := init_value_expr );
variable_name := 선언하려는 변수의 이름 (동일 block 내에 중복된 이름을 허용하지 않음)
data_type :=   INT
             ¦ SHORT
             ¦ LONG
             ¦ FLOAT
             ¦ DOUBLE
             ¦ CHAR (size)
init_value_expr := 초기값을 지정할 경우 상수나 수식을 기술할 수 있다.
```

Data type	Size	C type 참고
INT	4 bytes	int
SHORT	2 bytes	short
FLOAT	4 bytes	float
DOUBLE	8 bytes	double
CHAR(size)	Size 만큼 할당 (byte 단위)	char
LONG	8 bytes	long long

Init_value가 지정되지 않을 경우 변수는 실행되기 전에 NULL로 초기화 된다. 따라서 숫자형 변수들은 모두 0으로 초기화되고 문자형 변수는 0x00 으로 표현된다.

%TYPE

이미 생성된 테이블의 column이나 앞서 선언된 변수의 타입을 그대로 사용하기 위해 사용된다.

```
Typed variable ::=   table_name.column_name % TYPE
                   ¦ variable_name % TYPE
                   ¦ variable_name.field_name %TYPE
                   ;
```

다음과 같이 사용할 수 있는데 테이블이나 변수의 타입을 사용하려면 해당 테이블이나 변수가 미리 생성되어 있거나 선언되어 있는 상태여야 한다.

```
dbmMetaManager(DEMO)> CREATE TABLE T1 (C1 INT, C2 INT);
success
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE PROC1
IS
  V1     INT;
  V2     V1%TYPE;
  V3     T1.C1%TYPE;
BEGIN
  NULL;
END;
/
success
```

%TYPE의 초기값은 참조되는 변수의 초기값을 따라가지 않으므로 필요할 경우 초기값을 설정해야 한다.

%ROWTYPE

하나 이상의 필드를 갖는 변수를 선언하기 위해 사용하는데 이 때 필드는 테이블의 구조와 동일하게 구성된다.

```
Row typed variable ::= table_name % ROWTYPE;
```

다음은 테이블 구조와 동일한 ROWTYPE 변수를 선언하는 예이다.

```
dbmMetaManager(DEMO)> CREATE TABLE T1 (C1 INT, C2 INT);
success
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE PROC1
IS
  V1     T1%ROWTYPE;
BEGIN
  NULL;
```

```
END;
/
success
```

위의 예제에서 V1 변수는 각각 두 개의 field (C1, C2)를 가진 대표 변수로 선언되며 각 field는 다음과 같이 표기하여 사용할 수 있다.

```
dbmMetaManager(DEMO)> CREATE TABLE T1 (C1 INT, C2 INT);
success
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE PROC1
IS
  V1    T1%ROWTYPE;
BEGIN
  V1.C1 := V1.C2;
END;
/
success
```

%ROWTYPE 변수들의 개별 필드가 1 : 1 호환 가능한 타입 구조일 경우 해당 변수들 간에 대표명으로 assign 할 수도 있다.

```
dbmMetaManager(DEMO)> CREATE TABLE T1 (C1 INT, C2 INT);
success
dbmMetaManager(DEMO)> CREATE TABLE T2 (C1 INT, C2 INT);
success
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE PROC1
IS
  V1    T1%ROWTYPE;
  V2    T1%ROWTYPE;
BEGIN
  V2 := V1;
END;
/
success
```

%ROWTYPE으로 선언된 변수에는 초기값을 설정할 수 없으며 실행 전에 모든 필드가 0x00 으로 초기화 되어 수행된다.

변수의 유효 범위

변수의 유효 범위는 선언된 block 내부이다. 상위 block에 선언된 변수는 하위 block에서도 참조할 수 있지만 하위 block의 변수는 상위 block에서 참조할 수 없다. 또한, 동일한 변수명이 여러 개 존재할 경우 먼저 탐색된 block 내의 변수를 참조하므로 다른 상위 block의 변수를 이용하려면 해당 block의 label을 통해 접근해야 한다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1
IS
  V1 INT;
BEGIN
     DECLARE
         V1 INT;
     BEGIN
         V1 := 1;
         DBMS_OUTPUT.PUT_LINE( 'SCOPE1: V1 = ' || V1 );
     END;
     DBMS_OUTPUT.PUT_LINE( 'SCOPE2: V1 = ' || V1 );
END;
/
success
dbmMetaManager(DEMO)> exec proc1
SCOPE1: V1 = 1
SCOPE2: V1 = 0
success
dbmMetaManager(DEMO)>
```

위의 예제에서 V1은 가장 안쪽 block 내에서만 유효하며 해당 block 밖으로 나와 두 번째로 출력될 때는 상위 block을 참조하기 때문에 0으로 출력된다.

Cursor

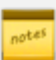
GOLDILOCKS LITE에서 cursor는 snapshot의 개념이다. 실행 시점에 획득한 SCN 값을 기준으로 자신이 접근 가능한 record에 대해 결과 집합을 만든다. 별도의 isolation level은 제공하지 않는다.

DECLARE

```
Cursor_declaration ::= CURSOR cursor_name [ ( param_list ) ]  IS select_statement ;
cursor_name := Cursor 이름
param_list := Cursor 선언 내의 select 문에 사용될 parameter가 있을 경우 기술한다. 없으면
생략한다.
select_statement := 결과 집합에 대한 질의문
```

다음과 같이 cursor 관련 구문을 활용하여 실행할 수 있다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1()
IS
  v1 INT;
  v2 CHAR(20);
  v3 DOUBLE;
  v4 t1%ROWTYPE;
  CURSOR c1 (a1 INT, a2 DOUBLE )
     IS select * from t1 where c1 >= a1 and c3 > a2;
BEGIN
  OPEN c1 (2, 3);
  LOOP
    FETCH c1 INTO v1, v2, v3;
    EXIT WHEN c1%NOTFOUND;
    dbms_output.put_line( 'v1 = ' || v1 || ', v2 = ' || v2 || ', v3 = ' || v3 || ', RowCount='
|| c1%rowcount );
  END LOOP;
  CLOSE c1;
  OPEN c1 (2, 3);
  LOOP
    FETCH c1 INTO v4;
    EXIT WHEN c1%notfound;
    dbms_output.put_line( 'c1 = ' || v4.c1 || ', c2 = ' || v4.c2 || ', c3 = ' || v4.c3 || ',
RowCount=' || c1%rowcount );
  END LOOP;
END;
/
success
dbmMetaManager(DEMO)> exec proc1
V1 = 3, V2 = c, V3 = 3.500000, ROWCOUNT=1
V1 = 4, V2 = d, V3 = 4.500000, ROWCOUNT=2
V1 = 5, V2 = e, V3 = 5.500000, ROWCOUNT=3
C1 = 3, C2 = c, C3 = 3.500000, ROWCOUNT=1
C1 = 4, C2 = d, C3 = 4.500000, ROWCOUNT=2
C1 = 5, C2 = e, C3 = 5.500000, ROWCOUNT=3
success
```



Cursor declare 절에 parameter가 기술된 경우, cursor open을 실행할 때 선언된 parameter 개수와 동일한 type과 개수의 데이터를 입력해야 한다.

User Exception

Procedure를 실행하는 과정에서 사용자가 예외 처리할 사항을 정의한다.

```
User_exception ::= exception_name EXCEPTION ;
exception_name := 사용자가 정의하려는 exception name
```

다음은 user exception을 사용한 exception handler의 예이다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1
IS
  user1 EXCEPTION;
BEGIN
  BEGIN
    RAISE user1;
    EXCEPTION WHEN case_not_found THEN dbms_output.put_line('fatal execution');
  END;
EXCEPTION WHEN user1 THEN dbms_output.put_line( 'SQLCODE=' || SQLCODE || ',SQLERRM=' ||
SQLERRM );
END;
/
success
dbmMetaManager(DEMO)> exec proc1
SQLCODE=70121,SQLERRM=a user exception raised
success
```

GOLDILOCKS LITE에 미리 정의된 내부 exception은 다음 표와 같다.

EXCEPTION NAME	Error code	설명
CASE_NOT_FOUND	70117	Simple/ searched case 구문에서 모든 경우의 수에 해당되지 않고 else 구문도 정의되지 않은 경우에 발생한다.
CURSOR_ALREADY_OPEN	70118	Cursor가 이미 열려 있는 경우에 발생한다.
DUP_VAL_ON_INDEX	70055	Insert 문에서 key가 중복되는 경우에 발생한다.
NO_DATA_FOUND	70111	Select 문에서 대상 데이터를 찾지 못한 경우에 발생한다.
TOO_MANY_ROWS	70113	Select into 문에서 두 건 이상 반환되는 경우에 발생한다.
VALUE_ERROR	70047	(divide_by_zero를 포함하여) 수식에 오류가 있는 경우에 발생한다.

Assign Statement

Assign statement는 procedure 내의 변수에 사용자가 지정한 값을 설정할 때 사용한다.

```
Assign_statement ::=   Variable := Expression ;
variable ::= Procedure내에 선언된 변수 이름
expression := 수식
```

A := B의 수식에서 A가 RowType과 같은 필드로 구성된 변수 타입인 경우, B도 A와 동일한 타입 또는 필드의 구성 타입과 개수가 동일한 타입의 변수이어야 assign을 수행할 수 있다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1 ()
IS
v1 t1%ROWTYPE;
v2 t1%ROWTYPE;
BEGIN
    v1.c1 := 100;
    v1.c2 := 200;
    dbms_output.put_line( 'v1.c1 = ' || v1.c1 );
    dbms_output.put_line( 'v1.c2 = ' || v1.c2 );
    v2 := v1;
    dbms_output.put_line( 'v2.c1 = ' || v2.c1 );
    dbms_output.put_line( 'v2.c2 = ' || v2.c2 );
END;
/
success
dbmMetaManager(DEMO)> exec proc1
V1.C1 = 100
V1.C2 = 200
V2.C1 = 100
V2.C2 = 200
success
```

Procedure assign의 target이 parameter일 경우 BindingMode가 IN이면 값을 설정할 수 없다.

Control Statements

Procedure 수행 과정에서 조건에 따라 반복, 점프, 분기 등을 수행하기 위한 구문을 제공한다.

IF

조건절을 만족할 경우, 하위 statement들을 수행한다.

```
IF_statement ::=  IF condition_expression THEN statements
                  [ ( ELSIF condition_expression THEN Statements ) ... ]
                  ¦
                  [ ELSE statements ]
                  END IF ;
condition_expression ::= 수행 여부를 판단하는 조건 수식
statements ::= Procedure statement의 정의
```

condition_expression에서 두 개 이상의 조건을 판단해야 할 경우, AND 또는 OR로 나열하여 기술한다.
다음 예제를 참조한다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1( a1 int )
IS
c1 INT;
BEGIN
  c1 := a1;
  IF c1 = 1 THEN
      dbms_output.put_line( 'C1(cond1) = ' ¦¦ c1 );
  END IF;
  IF c1 = 1 THEN
      dbms_output.put_line( 'C1(cond2.1) = ' ¦¦ c1 );
  ELSE
      dbms_output.put_line( 'C1(cond2.2) = ' ¦¦ c1 );
  END IF;
  IF c1 = 1 THEN
      dbms_output.put_line( 'C1(cond3.1) = ' ¦¦ c1 );
  ELSIF c1 = 2 THEN
      dbms_output.put_line( 'C1(cond3.2) = ' ¦¦ c1 );
  ELSIF c1 = 3 THEN
      dbms_output.put_line( 'C1(cond3.3) = ' ¦¦ c1 );
  ELSE
      dbms_output.put_line( 'C1(cond3.4) = ' ¦¦ c1 );
```

```
  END IF;
END;
/
dbmMetaManager(DEMO)> exec proc1( 1 )
C1(COND1) = 1
C1(COND2.1) = 1
C1(COND3.1) = 1
success
dbmMetaManager(DEMO)> exec proc1( 2 )
C1(COND2.2) = 2
C1(COND3.2) = 2
success
dbmMetaManager(DEMO)> exec proc1( 3 )
C1(COND2.2) = 3
C1(COND3.3) = 3
success
dbmMetaManager(DEMO)> exec proc1( 4 )
C1(COND2.2) = 4
C1(COND3.4) = 4
success
```

Simple Case

Case 절에 기술한 값이 WHEN 절의 수식 결과값과 일치할 경우, 해당 WHEN 절에 기술된 statement들을 수행한다.

```
simple_case ::= CASE value_expression
                [ WHEN value_expression THEN statements ; ( ... ) ]
                ( ELSE statements ; )
                END CASE;
value_expression ::= 수식
statements ::= Procedure에서 실행 가능한 구문
```

다음 예제를 참조한다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1( a1 int )
IS
  c1 INT;
BEGIN
  c1 := a1;
  dbms_output.put_line( 'NoElsePart' );
  CASE a1
```

```
        WHEN 1 THEN dbms_output.put_line( '1. c1 = ' || c1 );
        WHEN 2 THEN dbms_output.put_line( '2. c1 = ' || c1 );
        WHEN 3 THEN dbms_output.put_line( '3. c1 = ' || c1 );
    END CASE;
    dbms_output.put_line( 'ElsePart' );
    CASE a1
        WHEN 1 THEN dbms_output.put_line( '1. c1 = ' || c1 );
        WHEN 2 THEN dbms_output.put_line( '2. c1 = ' || c1 );
        ELSE dbms_output.put_line( 'else c1 = ' || c1 );
    END CASE;
END;
/
success
dbmMetaManager(DEMO)> exec proc1(1)
NOELSEPART
1. C1 = 1
ELSEPART
1. C1 = 1
success
dbmMetaManager(DEMO)> exec proc1(2)
NOELSEPART
2. C1 = 2
ELSEPART
2. C1 = 2
success
dbmMetaManager(DEMO)> exec proc1(3)
NOELSEPART
3. C1 = 3
ELSEPART
ELSE C1 = 3
success
```

어떤 조건도 만족하지 않고 ELSE 절도 생략된 경우, CASE_NOT_FOUND 에러가 발생한다.

Searched Case

CASE WHEN 절에 기술된 수식이 참 (TRUE)인 경우, 해당 WHEN 절의 statement들을 수행한다.

```
searched_case::= CASE [ WHEN condition_expression THEN statements ; ( ... ) ]
                  ( ELSE statements ; )
                   END CASE;
condition_expression ::= 조건 수식
statements ::= Procedure에서 실행 가능한 구문
```

다음 예제를 참조한다.

```
dbmMetaManager(demo)> CREATE OR REPLACE PROCEDURE proc1( a1 int )
IS
  c1 INT;
BEGIN
  c1 := a1;
  dbms_output.put_line( 'NoElsePart' );
  CASE
      WHEN a1 = 1 THEN dbms_output.put_line( '1. c1 = ' || c1 );
      WHEN a1 = 2 THEN dbms_output.put_line( '2. c1 = ' || c1 );
      WHEN a1 = 3 THEN dbms_output.put_line( '3. c1 = ' || c1 );
  END CASE;
  dbms_output.put_line( 'ElsePart' );
  CASE
      WHEN a1 = 1 THEN dbms_output.put_line( '1. c1 = ' || c1 );
      WHEN a1 = 2 THEN dbms_output.put_line( '2. c1 = ' || c1 );
      ELSE dbms_output.put_line( 'else c1 = ' || c1 );
  END CASE;
END;
/
success
dbmMetaManager(DEMO)> exec proc1(1)
NOELSEPART
1. C1 = 1
ELSEPART
1. C1 = 1
success
dbmMetaManager(DEMO)> exec proc1(2)
NOELSEPART
2. C1 = 2
ELSEPART
2. C1 = 2
success
dbmMetaManager(DEMO)> exec proc1(3)
```

```
NOELSEPART
3. C1 = 3
ELSEPART
ELSE C1 = 3
success
```

어떤 조건도 만족하지 않고 ELSE 절도 생략된 경우, CASE_NOT_FOUND 에러가 발생한다.

GOTO

특정 label 위치로 jump 한다. 일반적인 C 언어의 goto 문과 같은 개념으로 사용한다. 단, IF 절의 안쪽으로는 진입할 수 없다.

```
GOTO_statement ::= GOTO label;
label ::= Procedure 내에 명시된 label name
```

다음 예제를 참조한다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1()
IS
  c1 INT := 0;
BEGIN
    << AA >>
    BEGIN
       dbms_output.put_line( 'seoul(c1=' || c1 || ')' );
       << BB >>
       BEGIN
           c1 := c1 + 1;
           dbms_output.put_line( 'paris(c1=' || c1 || ')' );
           IF c1 < 3
           THEN
              GOTO aa;
           ELSIF c1 >= 3 AND c1 < 5 THEN
              GOTO bb;
           ELSE
              GOTO cc;
           END IF;
           dbms_output.put_line( 'never print' );
           BEGIN
               dbms_output.put_line( 'brazil(c1=' || c1 || ')' );
```

```
                BEGIN
                  << CC >>
                  dbms_output.put_line( 'sanghai(c1=' || c1 || ')' );
                  IF c1 = 5 THEN
                     GOTO aa;
                  ELSIF c1 = 6 THEN
                     GOTO bb;
                  END IF;
                END;
            END;
            dbms_output.put_line( 'taipei(c1=' || c1 || ')' );
        END;
    END;
END;
/
success
dbmMetaManager(DEMO)> exec proc1
SEOUL(C1=0)
PARIS(C1=1)
SEOUL(C1=1)
PARIS(C1=2)
SEOUL(C1=2)
PARIS(C1=3)
PARIS(C1=4)
PARIS(C1=5)
SANGHAI(C1=5)
SEOUL(C1=5)
PARIS(C1=6)
SANGHAI(C1=6)
PARIS(C1=7)
SANGHAI(C1=7)
TAIPEI(C1=7)
success
```

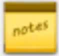
EXIT (WHEN)

LOOP를 수행하는 도중에 멈추고 해당 scope를 빠져나가기 위해 사용한다.

```
EXIT_statement ::= EXIT ( WHEN condition_expression ) ;
condition_expression ::= 조건을 지정하고자 할 경우 조건 수식을 기술
```

조건없이 EXIT를 조건 없이 기술할 경우에는 즉시 scope를 빠져나가고, 조건절이 기술된 경우에는 조건을 만족했을 때 scope를 벗어난다.
다음 예제를 참조한다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1()
IS
  c1 INT;
BEGIN
    dbms_output.put_line( 'first_loop' );
    c1 := 0;
    LOOP
        c1 := c1 + 1;
        dbms_output.put_line( 'c1 = ' || c1 );
        EXIT;
    END LOOP;
    dbms_output.put_line( 'second_loop' );
    c1 := 0;
    LOOP
        c1 := c1 + 1;
        dbms_output.put_line( 'c1 = ' || c1 );
        EXIT WHEN c1 = 10;
    END LOOP;
END;
/
success
dbmMetaManager(DEMO)> exec proc1
FIRST_LOOP
C1 = 1
SECOND_LOOP
C1 = 1
C1 = 2
C1 = 3
C1 = 4
C1 = 5
C1 = 6
C1 = 7
C1 = 8
C1 = 9
C1 = 10
success
```

 EXIT 구문은 LOOP 계열 구문에서만 사용할 수 있다.

CONTINUE (WHEN)

LOOP를 수행하는 도중에 조건을 만족할 경우, LOOP scope 내의 시작 위치로 이동하여 수행을 시작한다.

```
CONTINUE_statement ::= CONTINUE ( WHEN condition_expression ) ;
condition_expression ::= 조건을 지정하고자 할 경우 조건 수식을 기술
```

조건없이 CONTINUE를 기술할 경우에는 LOOP 내의 첫 번째 statement로 이동하고, 조건절이 기술된 경우에는 조건을 만족할 경우에 이동한다.
다음 예제를 참조한다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1()
IS
  c1 INT;
BEGIN
    c1 := 0;
    LOOP
        c1 := c1 + 1;
        IF c1 < 5 THEN
            CONTINUE;
        END IF;
        CONTINUE WHEN c1 < 10;
        dbms_output.put_line( 'c1 = ' || c1 );
        EXIT;
    END LOOP;
END;
/
success
dbmMetaManager(DEMO)> exec proc1
C1 = 10
success
```

CONTINUE 구문은 LOOP 계열 구문에서만 사용할 수 있다.

RAISE_APPLICATION_ERROR

사용자가 특정 에러 코드와 에러 메시지를 설정하여 exception을 발생시키고자 할 경우에 사용된다.
GOLDILOCKS LITE에서 사용자 에러 코드는 음수로만 정의할 수 있다.

```
raise_application_error_statement ::= RAISE_APPLICATION_ERROR( errorCode, errorMessage);
errorCode ::= 음수 범위 내의 정수형 값
errorMessage ::= 사용자가 지정한 에러 메시지 (512 byte 이내로 설정 가능)
```

다음과 같이 exception을 발생시키는데 exception handler에 의해 처리되지 않을 경우, 상위로 에러를 전달한다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1
IS
 user1 EXCEPTION;
 PRAGMA EXCEPTION_INIT( user1, -100);
BEGIN
  RAISE_APPLICATION_ERROR( -100, 'user1 raise' );
EXCEPTION WHEN user1 THEN dbms_output.put_line( 'SQLCODE=' || SQLCODE || ', SQLERRM=' ||
SQLERRM );
END;
/
success
dbmMetaManager(DEMO)> exec proc1
SQLCODE=-100, SQLERRM=USER1 RAISE
success
```

User exception의 에러 코드가 일치하지 않을 경우 unhandled exception으로 처리되어 상위로 전달된다.

Loop Statements

Procedure 내에서 반복되는 명령문을 처리하기 위한 구문들이다.

SIMPLE LOOP

LOOP와 END LOOP 구문 사이에 기술된 구문들을 반복적으로 수행한다. Loop 구간을 빠져나가려면 EXIT 구문을 사용한다.

```
LOOP Statement ::=   LOOP
                        < statements >
                     END LOOP
statements ::= Procedure 내에서 사용 가능한 구문들
```

```
dbmMetaManager(demo)> CREATE OR REPLACE PROCEDURE proc1()
IS
  c1 INT;
BEGIN
    dbms_output.put_line( 'first_loop' );
    c1 := 0;
    LOOP
        c1 := c1 + 1;
        dbms_output.put_line( 'c1 = ' || c1 );
        EXIT;
    END LOOP;
    dbms_output.put_line( 'second_loop' );
    c1 := 0;
    LOOP
        c1 := c1 + 1;
        dbms_output.put_line( 'c1 = ' || c1 );
        EXIT WHEN c1 = 10;
    END LOOP;
END;
/
success
dbmMetaManager(DEMO)> exec proc1
FIRST_LOOP
C1 = 1
SECOND_LOOP
C1 = 1
C1 = 2
C1 = 3
C1 = 4
C1 = 5
C1 = 6
C1 = 7
C1 = 8
C1 = 9
C1 = 10
success
```

FOR LOOP

FOR LOOP는 FOR 구문에 기술된 수식을 증가/ 감소시키면서 최종값에 도달할 때까지 LOOP ~ END LOOP 구문 사이에 기술된 procedure 구문들을 수행한다.

```
FOR LOOP Statement ::= FOR <identifier> IN (REVERSE) <first_expr> .. <last_expr>
                       LOOP
                            <statements>
                       END LOOP
identifier := FOR LOOP 내에서 사용될 LOOP 범위 연산의 변수명
REVERSE := 생략할 경우 First_expr을 시작으로 Last_expr 값까지 진행, 기술되면 역순으로 진행
first_expr := FOR LOOP 내의 identifier가 가질 첫 번째 수식 값, REVERSE가 기술된 경우에는
마지막 값
last_expr := FOR LOOP 내의 identifier가 가질 마지막 수식 값, REVERSE가 기술된 경우에는 첫 번째
값
statements := FOR LOOP 내에서 수행될 procedure 구문들
```

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1()
IS
  c1 INT;
BEGIN
  FOR i IN 1 .. 5
  LOOP
    dbms_output.put_line( 'i = ' || i );
  END LOOP;
END;
/
success
dbmMetaManager(DEMO)> EXEC proc1()
I = 1
I = 2
I = 3
I = 4
I = 5
success
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1()
IS
  c1 INT;
BEGIN
  FOR i IN REVERSE 1 .. 5
  LOOP
```

```
    dbms_output.put_line( 'i = ' || i );
  END LOOP;
END;
/
success
dbmMetaManager(DEMO)> EXEC proc1()
I = 5
I = 4
I = 3
I = 2
I = 1
success
```

FOR LOOP 내의 identifier는 declare 절에 미리 기술되어 있지 않아도 되지만 이 경우, FOR LOOP ~ END LOOP 구문 사이의 범위 내에서만 참조할 수 있다. 해당 scope를 벗어날 경우 참조할 수 없다.

WHILE LOOP

WHILE LOOP는 WHILE 절에 기술된 조건이 참인 경우에만 LOOP 내의 procedure 구문들을 수행하는 방식이다.

```
WHILE LOOP Statement ::= WHILE  <cond_expr >
                         LOOP
                            <statements>
                         END LOOP
```

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1()
IS
  c1 INT := 0;
BEGIN
    dbms_output.put_line( 'start: c1 = ' || c1 );
    WHILE c1 < 5
    LOOP
        c1 := c1 + 1;
    END LOOP;
    dbms_output.put_line( 'last c1 = ' || c1 );
END;
/
success
dbmMetaManager(DEMO)> EXEC proc1
START: C1 = 0
```

```
LAST C1 = 5
success
```

CURSOR Statements

SELECT 문을 사용하여 두 건 이상의 데이터를 가져오고자 할 경우에 사용한다. GOLDILOCKS LITE의 cursor는 실행 시점의 snapshot을 이용하며 committed-read mode만 보장한다.
Cursor를 사용하기 위해서는 cursor 정의를 먼저 수행해야 하는데 이는 declare section의 **Cursor** 부분을 참조한다.

OPEN

Declare section에 정의된 cursor 구문을 실행한다.

```
OPEN statement ::= OPEN <cursor_name>;
cursor_name := Declare section에 정의된 cursor 이름
```

```
dbmMetaManager(DEMO)> CREATE TABLE t1 (c1 INT, c2 CHAR(20), c3 DOUBLE)
success
dbmMetaManager(DEMO)> CREATE UNIQUE INDEX idx_t1 ON t1 (c1)
success
dbmMetaManager(DEMO)> INSERT INTO t1 VALUES (1, 'a', 1.5 )
success
dbmMetaManager(DEMO)> INSERT INTO t1 VALUES (2, 'b', 2.5 )
success
dbmMetaManager(DEMO)> INSERT INTO t1 VALUES (3, 'c', 3.5 )
success
dbmMetaManager(DEMO)> INSERT INTO t1 VALUES (4, 'd', 4.5 )
success
dbmMetaManager(DEMO)> INSERT INTO t1 VALUES (5, 'e', 5.5 )
success
dbmMetaManager(DEMO)> COMMIT
success
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1()
IS
  v1 INT;
  v2 CHAR(20);
  v3 DOUBLE;
  v4 t1%ROWTYPE;
CURSOR c1 (a1 INT, a2 DOUBLE )
```

```
 IS select * from t1 where c1 >= a1 and c3 > a2;
BEGIN
  OPEN c1 (2, 3);
  LOOP
    FETCH c1 INTO v1, v2, v3;
    EXIT WHEN c1%NOTFOUND;
    dbms_output.put_line( 'v1 = ' || v1 || ', v2 = ' || v2 || ', v3 = ' || v3 || ', RowCount='
|| c1%rowcount );
  END LOOP;
  CLOSE c1;
  OPEN c1 (2, 3);
  LOOP
    FETCH c1 INTO v4;
    EXIT WHEN c1%NOTFOUND;
    dbms_output.put_line( 'c1 = ' || v4.c1 || ', c2 = ' || v4.c2 || ', c3 = ' || v4.c3 || ',
RowCount=' || c1%rowcount );
  END LOOP;
END;
/
success
dbmMetaManager(DEMO)> EXEC proc1
V1 = 3, V2 = c, V3 = 3.500000, ROWCOUNT=1
V1 = 4, V2 = d, V3 = 4.500000, ROWCOUNT=2
V1 = 5, V2 = e, V3 = 5.500000, ROWCOUNT=3
C1 = 3, C2 = c, C3 = 3.500000, ROWCOUNT=1
C1 = 4, C2 = d, C3 = 4.500000, ROWCOUNT=2
C1 = 5, C2 = e, C3 = 5.500000, ROWCOUNT=3
success
```

FETCH

OPEN 구문으로 열린 cursor에 한해 record를 한 건씩 읽어들여 INTO 절에 기술된 변수로 값을 복사한다.

```
FETCH Statement ::=  FETCH <cursor_name>  INTO < variable_name (, ... ) >
cursor_name := Open 구문으로 정상 처리된 cursor name
variable_name := Fetch 된 결과를 저장할 변수 이름
```

Scalar와 ROWTYPE 변수 둘 다 INTO 절에 기술할 수 있다. 변수를 기술하는데 제약 사항은 없지만 cursor결과를 가진 target 절의 개수와 INTO 절에 기술된 변수와의 DataType 호환에 제약이 있을 경우 fetch를 수행하는 도중에 오류가 발생할 수 있다.

CLOSE

OPEN 구문에 의해 열린 cursor를 정리한다.

```
CLOSE statement ::= CLOSE <cursor_name>
cursor_name := Open 구문에 의해 정상적으로 열린 cursor 이름
```

SQL Statements

Procedure 내에 일반 DML을 사용할 수 있는데 이에 대한 구문을 설명한다.

INSERT

한 건의 record를 지정한 테이블에 삽입한다.

```
INSERT Statement  ::= INSERT INTO <target_table> ( column_name (, ...) >
                      VALUES ( value_expr ( ,...) );
target_table := 대상 테이블
column_name := 테이블 내의 특정 column을 나열하고자 할 경우
value_expr := Procedure 변수를 포함한 value list
```

Value 절에는 procedure 내의 RowType 변수를 사용할 수 있다.

UPDATE

테이블에서 한 건 이상의 레코드의 지정된 column 값을 갱신한다.

```
UPDATE Statement ::= UPDATE <target_table>
                     SET  <column_name> = <value_expr> (, ...)
                     ( WHERE cond_expr );
target_table := 대상 테이블
column_name := 값을 변경하고자 하는 대상 column 이름
value_expr := 변수등을 포함하는 value expression
cond_expr := 특정 레코드를 탐색할 경우 해당하는 조건절
```

DELETE

테이블에서 한 건 이상의 레코드를 삭제한다.

```
DELETE Statement ::= DELETE FROM <target_table>
                     (WHERE cond_expr);
target_table := 대상 테이블
cond_expr := 특정 레코드를 탐색할 경우 조건을 기술
```

SELECT INTO

테이블에서 한 건의 데이터를 탐색하여 procedure의 변수에 값을 저장하고자 할 경우에 사용한다. Cursor와 달리 한 건만 fetch 할 수 있다.

```
SELECT INTO statement ::= SELECT <target_list> INTO <variable_list>
                                    FROM <source_table>
                                   (WHERE cond_expr);
target_list := 테이블에서 가지고 올 column 이름 ( *를 사용할 경우, 모든 column을 가지고 옴)
variable_list := Procedure 내의 변수
source_table := 대상 테이블
cond_expr := 조건절
```

INTO 절에는 procedure의 RowType 변수를 사용할 수 있다.

COMMIT

현재 세션에서 진행한 트랜잭션을 영구적으로 반영한다.
다음 예제와 같이 procedure 내에서 COMMIT을 사용하여 트랜잭션을 영구적으로 반영한다. 자신의 세션에서 조회할 경우 commit 하기 전이라도 세션 변경사항을 볼 수 있지만 다른 세션에서 조회할 경우에는 변경 전의 committed -image를 조회한다.

```
dbmMetaManager(DEMO)> CREATE TABLE t1 (c1 INT, c2 INT)
success
dbmMetaManager(DEMO)> INSERT INTO t1 VALUES (2, 2)
success
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1
IS
  V1 INT := 100;
BEGIN
    INSERT INTO T1 VALUES (1, 1);
```

```
    COMMIT;
    UPDATE T1 SET C2 = 100 WHERE C1 = 1;
    DBMS_OUTPUT.PUT_LINE( 'v1 = ' || v1 );
END;
/
success
dbmMetaManager(DEMO)> EXEC proc1
V1 = 100
success
dbmMetaManager(DEMO)> SELECT * FROM T1
--------------------------------------------------------------------------------
C1                    : 2
C2                    : 2
--------------------------------------------------------------------------------
C1                    : 1
C2                    : 100
--------------------------------------------------------------------------------
2 row selected
```

다른 세션에서 조회할 경우 다음과 같이 변경전 이미지를 조회한다.

```
[2nd_proc] dbmMetaManager(DEMO)> set instance demo
success
[2nd_proc] dbmMetaManager(DEMO)> select * from t1
--------------------------------------------------------------------------------
C1                    : 2
C2                    : 2
--------------------------------------------------------------------------------
C1                    : 1
C2                    : 1
--------------------------------------------------------------------------------
2 row selected
```

ROLLBACK

현재의 세션에서 진행한 트랜잭션을 철회한다.
다음 예제와 같이 갱신된 데이터에 rollback을 수행하여 트랜잭션을 이전의 원래 데이터로 복구한다.

```
dbmMetaManager(DEMO)> CREATE TABLE t1 (c1 INT, c2 INT)
success
dbmMetaManager(DEMO)> INSERT INTO t1 VALUES (2, 2)
```

```
success
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc1
IS
  V1 INT := 200;
BEGIN
    INSERT INTO T1 VALUES (1, 1);
    UPDATE T1 SET C2 = 100 WHERE C1 = 1;
    COMMIT;
    UPDATE T1 SET C2 = 200 WHERE C1 = 1;
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
END;
/
success
dbmMetaManager(DEMO)> EXEC proc1
V1 = 200
success
dbmMetaManager(DEMO)> SELECT * FROM T1
---------------------------------------------------------------------------------
C1                    : 2
C2                    : 2
---------------------------------------------------------------------------------
C1                    : 1
C2                    : 100
---------------------------------------------------------------------------------
2 row selected
```

Exception Handler

Exception handler는 procedure를 수행하는 도중에 발생하는 오류를 감지하여 사용자가 지정한 코드를 수행하고 해당 블럭을 종료하는 procedure의 exception 코드들이다.

이러한 예외 처리들은 미리 정의된 구문이나 사용자 정의에 의해 구분된다. 사용자 정의 exception에 대한 자세한 앞서 설명한 **Declare Section**을 참조한다.

PreDefined Exception

EXCEPTION NAME	Error code	설명

EXCEPTION NAME	Error code	설명
CASE_NOT_FOUND	70117	Simple/ searched case 구문에서 모든 경우의 수에 해당되지 않고 else 구문도 정의되지 않은 경우에 발생한다.
CURSOR_ALREADY_OPEN	70118	Cursor가 이미 열려 있는 경우에 발생한다.
DUP_VAL_ON_INDEX	70055	Insert 문에서 key가 중복되는 경우에 발생한다.
NO_DATA_FOUND	70111	Select 문에서 대상 데이터를 찾지 못한 경우에 발생한다.
TOO_MANY_ROWS	70113	Select into 문에서 두 건 이상 반환되는 경우에 발생한다.
VALUE_ERROR	70047	(divide_by_zero를 포함하여) 수식에 오류가 있는 경우에 발생한다.

PreDefined exception은 여타의 DBMS 구문 및 오류상황과 다를 수 있다.

Exception Handler 정의

```
Exception Handler ::= EXCEPTION
                      ( WHEN <exception_name> THEN <statements>; [, ...] )
exception_name := PreDefined exception 또는 사용자 정의 exception name
statements := Procedure 구문들
```

다음과 같은 방식으로 사용할 수 있다. 다음은 select 문에 의해 NOT_FOUND 에러가 발생한 경우, OTHERS exception handler를 통해 처리되는 부분과 exception scope에 대한 예이다.

```
dbmMetaManager(DEMO)> CREATE OR REPLACE PROCEDURE proc2 ()
IS
v1 INT;
BEGIN
    dbms_output.put_line( 'start' );
    BEGIN
        BEGIN
            select c1 into v1 from t1 where c1 = 1;
            dbms_output.put_line( 'first line' );
        EXCEPTION
        WHEN others THEN dbms_output.put_line( 'exception raise' );
        END;
        dbms_output.put_line( 'middle line' );
    EXCEPTION
    WHEN others THEN dbms_output.put_line( 'exception raise' );
    END;
```

```
    dbms_output.put_line( 'last line' );
END;
/
success
dbmMetaManager(DEMO)> EXEC proc2
START
EXCEPTION RAISE
MIDDLE LINE
LAST LINE
success
```

Raise Statement

Procedure를 수행하는 도중에 사용자가 임의로 exception 상태를 발생시키려고 할 경우에 사용한다.

```
RAISE Statement ::= RAISE ( <exception_name> );
exception_name ::= 사용자 정의 exception 이름
```

RAISE 문이 exception handler에서 사용될 경우에는 exception 이름을 지정할 수 없다. 즉, 현재 발생한 exception 정보만 상위 블록으로 전달하는데 이용한다. (지정되더라도 동작하지 않는다.)

Attribute Variable

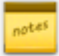
SQL 수행 상태 또는 cursor의 수행 상태와 정보를 가진 변수들을 의미하며 사용자 선언 없이 내부적으로 할당되어 존재한다.

SQL Attribute Variable

SQL 문의 수행 상태에 대한 정보를 담고 있다. GOLDILOCKS LITE의 상태/ 초기값은 다른 DBMS들에서 쓰이는 것과 다를 수 있다.

Cursor Attribute Variable

CURSOR 문의 수행 상태에 대한 정보를 담고 있다. GOLDILOCKS LITE의 상태/ 초기값은 다른 DBMS들에서 쓰이는 것과 다를 수 있다. SQL attribute 변수들과 의미는 동일하며 cursor에 대해서만 사용할 수 있다.



Godlilocks Lite에서 attribute 변수들의 초기값은 다른 DBMS의 설정값과 다르기 때문에 호환되는지 여부에 주의해야 한다. 예를 들어 FOUND, NOTFOUND의 경우에는 (True/False)의 개념이 아닌 (0/1)의 개념이 적용되어 있다.

SQLCODE

Procedure를 수행 중에 발생하는 오류들 중에 가장 마지막 오류의 에러 코드를 저장한다. Exception handler의 첫 구문이 정상적으로 처리되면 에러 코드는 0으로 설정된다.

SQLERRM

Procedure 수행 중에 발생하는 오류들 중에 가장 마지막 오류의 에러 메시지를 저장한다. Exception handler의 첫 구문이 정상적으로 처리되면 오류는 초기화 된다.

4.

V2_V3 변환 가이드

4.1 V3 신규 기능

- dbmMetaManager 내에서 일부 SQL 문을 사용할 수 있다.
- Information view를 통한 session, transaction 등의 상태를 SQL 형태로 조회할 수 있다.

4.2 Package 변경사항

- 모든 binary utility 이름의 prefix를 dbm으로 사용하도록 변경 (예: metaManager → dbmMetaManager)
- dbmAPI.h → dbmUserAPI.h 로 명칭 변경
- libdbm.so → libdbmCore.so 로 Shared Library Object 명칭 변경
- 기존 WAL → wal 경로로 변경

4.3 DB Object Conversion Guide

SQL 조건절 강화

- Ver2 대비 SQL 형태의 질의 처리가 가능하며 조건절에 AND, OR과 같은 논리 연산이 가능하다.
- (>, < , =, !=, NOT, IN)과 같은 연산을 지원한다.
- 제공되는 built-in 함수를 통한 질의 처리가 가능하다.
- Full-scan을 통한 질의 처리가 가능하다.

INDEX

V2 버전에서는 unique index의 기본값을 다음과 같이 설정하여 생성한다.
- create index: Unique index로 설정
- create non unique: Non unique index로 설정

V3 버전에서는 SQL 구문과 동일하게 기본값을 non-unique로 설정한다.
- create unique index: Unique index로 설정
- create index: Non unique index로 설정

기타 제한은 V2와 V3 모두 동일하다.

DROP INSTANCE

V2에서는 하위 object를 제거할 수 없었지만 V3에서는 하위 object를 모두 제거하므로 사용할 때 주의해야 한다.

CREATE INSTANCE

- Dictionanry instance (DICT)에서만 수행할 수 있다.
- V2에서는 CREATE INSTANCE를 수행할 때 자동으로 전환되었지만 V3에서는 반드시 SET INSTANCE를명시한 경우에만 전환된다.

JOIN 처리 (두 개 테이블로 제한)

```
dbmMetaManager(DEMO)> select * from t1 a, t2 b
where a.c1 = b.c1;
--------------------------------------------------------------------------------
C1                    : 1
C2                    : 1
C1                    : 1
C2                    : 11
--------------------------------------------------------------------------------
C1                    : 2
C2                    : 2
C1                    : 2
C2                    : 22
--------------------------------------------------------------------------------
C1                    : 3
C2                    : 3
C1                    : 3
C2                    : 33
--------------------------------------------------------------------------------
C1                    : 4
C2                    : 4
C1                    : 4
C2                    : 44
```

```
--------------------------------------------------------------------------------
C1                   : 5
C2                   : 5
C1                   : 5
C2                   : 55
--------------------------------------------------------------------------------
5 row selected
dbmMetaManager(DEMO)> select a.c1, a.c2, b.c2 from t1 a, t2 b
where a.c1 > 2
and a.c1 = b.c1;
--------------------------------------------------------------------------------
C1                   : 3
C2                   : 3
C2                   : 33
--------------------------------------------------------------------------------
C1                   : 4
C2                   : 4
C2                   : 44
--------------------------------------------------------------------------------
C1                   : 5
C2                   : 5
C2                   : 55
--------------------------------------------------------------------------------
3 row selected
```

- 제한된 조건 내에서 두 개 테이블에 한해 JOIN 방식을 사용할 수 있다.
- API 방식으로 사용할 경우 dbmPrepareStmt/dbmExecuteStmt/dbmFetchStmt API를 통해 사용할 수 있다.

Procedure 지원

- SQL을 이용한 절차형 개발 방식을 제공한다.
- Function은 지원하지 않는다.

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

```
success
dbmMetaManager(DEMO)> create table t1 (c1 int, c2 char(20), c3 double)
success
dbmMetaManager(DEMO)> create unique index idx_t1 on t1 (c1)
success
dbmMetaManager(DEMO)> insert into t1 values (1, 'a', 1.5 )
success
dbmMetaManager(DEMO)> insert into t1 values (2, 'b', 2.5 )
success
dbmMetaManager(DEMO)> insert into t1 values (3, 'c', 3.5 )
success
dbmMetaManager(DEMO)> insert into t1 values (4, 'd', 4.5 )
success
dbmMetaManager(DEMO)> insert into t1 values (5, 'e', 5.5 )
success
dbmMetaManager(DEMO)> commit
success
dbmMetaManager(DEMO)> create or replace procedure proc1()
is
  v1 int;
  v2 char(20);
  v3 double;
  v4 t1%rowtype;
cursor c1 (a1 int, a2 double )
 is select * from t1 where c1 >= a1 and c3 > a2;
begin
  open c1 (2, 3);
  loop
    fetch c1 into v1, v2, v3;
    exit when c1%notfound;
    dbms_output.put_line( 'v1 = ' || v1 || ', v2 = ' || v2 || ', v3 = ' || v3 || ', RowCount='
|| c1%rowcount );
  end loop;
  close c1;
  open c1 (2, 3);
  loop
    fetch c1 into v4;
    exit when c1%notfound;
    dbms_output.put_line( 'c1 = ' || v4.c1 || ', c2 = ' || v4.c2 || ', c3 = ' || v4.c3 || ',
RowCount=' || c1%rowcount );
  end loop;
```

```
end;
/
success
dbmMetaManager(DEMO)> exec proc1
v1 = 3, v2 = c, v3 = 3.500000, RowCount=1
v1 = 4, v2 = d, v3 = 4.500000, RowCount=2
v1 = 5, v2 = e, v3 = 5.500000, RowCount=3
c1 = 3, c2 = c, c3 = 3.500000, RowCount=1
c1 = 4, c2 = d, c3 = 4.500000, RowCount=2
c1 = 5, c2 = e, c3 = 5.500000, RowCount=3
success
dbmMetaManager(DEMO)> quit
```

기타 변경사항

- Object 이름들에서 더이상 대/소문자를 구별하지 않음
- 레코드 한 개당 최대 저장 크기가 30 K로 확장
- DBM_DISK_LOG_BUFFER_SIZE 항목을 제거 (메모리가 허용하는 한도로 변경하였다.)

4.4 API Conversion Guide

이 장에서는 V2 버전에서 V3 버전으로 변환할 때 참고할 사항들을 설명한다.

dbmHandle 처리

V2 코드

```
dbmHandle    sHandle;
dbmInitHandle( &sHandle, "DEMO" );
```

V3 코드

```
dbmHandle    * sHandle = NULL;
dbmInitHandle( &sHandle, "DEMO" );
```

- dbmHandle 변수가 pointer 형태로 변경된다.
- 해당 변수는 반드시 NULL로 초기화 된 상태에서 호출되어야 한다.
- Instance name을 기술하는 위치에서 더이상 대/소문자를 구별하지 않게 되었다.
- Instance name을 NULL로 기술할 경우, 자동적으로 사용자 환경변수 (DBM_INSTANCE)를 사용한다.
- 일반 API에서 EX로 동작하던 기능들을 포함하여 API_Ex로 제공되는 API들이 모두 제거되었다.

dbmPrepareTable

호환성을 위해 제공은 하지만 더 이상 호출은 하지 않고 최초 API에서 필요한 경우에 자동으로 처리된다.
다만 성능을 위해 별도로 제공되는 dbmPrepareTableHandle API 사용을 권장한다.

dbmInsertRow

- 삽입된 레코드의 slot ID를 반환하는 인자가 추가되었다.
- 대상 테이블 이름에서 더 이상 대/소문자를 구별하지 않게 되었다.

dbmUpdateRow

V2 코드

```
dbmHandle     sHandle;
UserData      sData;
dbmUpdateRow( &sHandle, "T1", &sData );
```

V3 코드

```
dbmHandle     * sHandle = NULL;
UserData        sData;
int             sAffectedCount;
dbmUpdateRow( sHandle, "t1", &sData, &sAffectedCount );
```

- 대상 테이블 이름에서 더이상 대/소문자를 구분하지 않게 되었다.
- (Non unique index를 지정한 경우) 마지막 인자에 update 된 대상 row의 개수를 반환한다.

dbmDeleteRow

V2 코드

```
dbmHandle     sHandle;
UserData      sData;
dbmDeleteRow( &sHandle, "T1", &sData );
```

V3 코드

```
dbmHandle     * sHandle = NULL;
UserData        sData;
int             sAffectedCount;
dbmDeleteRow( sHandle, "t1", &sData, &sAffectedCount );
```

- 대상 테이블 이름에서 더이상 대/소문자 구별을 하지 않게 되었다.
- (Non unique index를 지정한 경우) 마지막 인자에 delete 된 대상 row의 개수를 반환한다.

dbmOpenCursor(GT/LT)

V2 코드

```
dbmHandle     sHandle;
UserData      sData;
dbmOpenCursor( &sHandle, "T1", &sData );
dbmFetchRow( &sHandle, "T1", &sData );
```

V3 코드

```
dbmHandle     * sHandle = NULL;
UserData        sData;
int             sAffectedCount;
dbmSelectRow( sHandle, "t1", &sData );   // dbmSelectRowGT, dbmSelectRowLT
dbmFetchNextGT( sHandle, "t1", &sData );
```

- 대상 테이블 이름에서 더이상 대/소문자 구분을 하지 않게 되었다.
- OpenCursor는 non-unique index에 대해 동일한 키를 가진 데이터를 조회하는 용도로 사용되지만 V3에서는 dbmSelectRow를 통해 첫 번째로 일치하는 데이터를 찾은 후에 dbmFetchNextGT/LT 함수들을 통해 다음 데이터를 탐색할 수 있다.

단순한 조회만 목적으로 할 경우, 다음과 같이 SQL문을 활용하여 조회할 수도 있다.

```
dbmInitHandle    * sHandle = NULL;
dbmStmt          * sStmt = NULL;
int                v1;
int                v2;
dbmPrepareStmt( sHandle, "select * from t1 where c1 > 10 and c1 < 20", &sStmt );
dbmBindCol( sHandle, sStmt, &v1 );
dbmBindCol( sHandle, sStmt, &v2 );
dbmExecuteStmt( sHandle, sStmt );
while( !dbmFetchStmt( sHandle, sStmt ) )
{
    printf( "v1 = %d, v2 = %d\n", v1, v2 );
}
```

dbmBindCol는 조회하기 위해 fetch된 column을 저장할 변수를 연결하는 과정이며 select 된 target절의 타입과 동일한 순서로 기술한다.

dbmSelectRowDirtyRead 계열 함수

V3에서는 V2와 달리 commit 하는 도중에 Multi Versioning Concurrency Control (MVCC) 기법을 통해 read consistency를 보장하므로 더 이상 해당 API는 필요하지 않아 모두 제거된다. 동일한 방법으로 dbmSelectRow 함수를 통해 read를 보장한다.

dbmDeferCommit, dbmDeferSync API 제거

V1에서 제공되던 일부 API들은 더 이상 제공되지 않는다.

5.

High Availability (HA) 구성 가이드

5.1 High Availability (HA) 구성 가이드

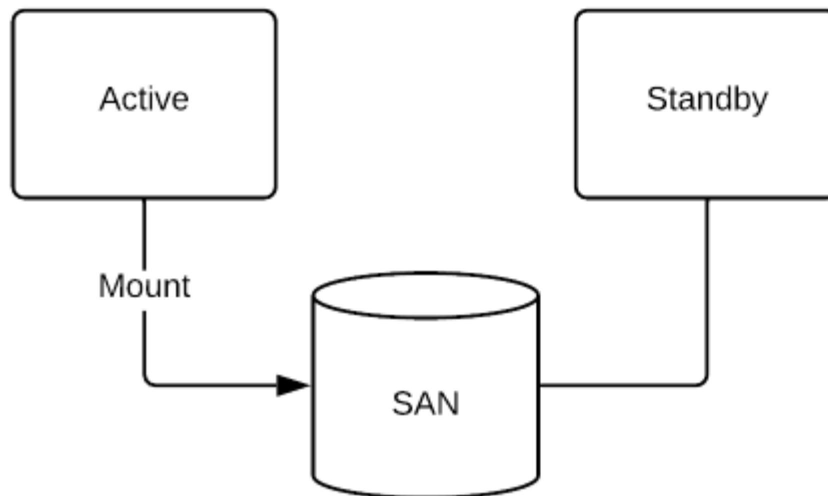
GOLDILOCKS LITE는 공유메모리 기반의 빠른 성능을 목표로 개발되었다. 따라서 사용자 운용환경에 맞춰 데이터 안정성 보장과 고가용성을 위해 디스크 로깅 방식을 선택적으로 사용할 수 있다.

모든 데이터가 공유메모리에 보존되므로 기본적인 환경에서는 데이터가 유실되지 않는다. 그러나 OS fatal, memory fault, 정전과 같은 하드웨어 장애가 발생할 경우 데이터가 유실될 수 있으므로 이에 대비하여 아래에서 설명하는 방식으로 운영할 수 있다. 단, 해당 방식으로 운영할 경우 기본적인 메모리 운영방식에 비해 성능이 저하될 수 있다는 점을 감안해야 한다.

5.2 디스크 로깅 방식

일반적인 RDBMS들과 유사하게 메모리에 변경되는 모든 사항들을 디스크 로깅 형태로 저장하고 주기적인 체크포인트를 통해 데이터 파일을 생성/ 관리하여 데이터를 복구할 수 있도록 운영하는 방법이다.

그림 1 LITE_SAN



고가용성을 위해 디스크 로깅 방식으로 운영할 경우 active/ standby 형태로 서버를 구성하고 디스크를 공유하는 방식을 권장한다. 평소에는 active에서 운영하면서 장애가 발생할 경우 공유된 디스크를 standby로 mount하여 데이터를 복구하고 서비스를 재개하는 형식의 운영 방법이다.

설정 방법

다음의 프로퍼티를 설정해야 한다.

Property name	설명	Value
DBM_DISK_LOG_ENABLE	디스크 로깅 활성화 여부	true
DBM_DISK_LOG_DIR	디스크 로그파일이 생성될 위치	절대경로
DBM_DISK_LOG_FILE_SIZE	디스크 로그파일 한 개당 최대 크기	최소 100M 이상 권장
DBM_DISK_DATA_FILE_DIR	데이터 파일이 생성될 위치	절대경로

위 프로퍼티들을 설정하면 각 세션당 로그 파일이 생성된다. I/O가 발생할 때 세션간 트랜잭션 순서에 대한 disk I/O 경합이 최소화 되도록 병렬 로깅과 순서에 영향을 받지 않는 복구 메커니즘을 통해 체크포인트 시점마다 생성된 로그 파일을 데이터 파일에 반영한다.

기본 로깅 설정은 세션이 commit을 호출할 때마다 I/O을 발생시킨다. 위와 같은 형태는 로깅에 대한 경합을 최소화 할 수 있지만 I/O 호출을 빈번하게 발생시킨다.

따라서 만일 다음과 같이 설정할 경우 이를 공유 메모리에 로깅하고 주기적으로 디스크로 flush 시키는 형태로 운영할 수도 있다.

Property name	설명	Value
DBM_LOG_CACHE_MODE	Cache mode 설정 여부이다.	1: NVDIMM을 이용한다. 2: 공유메모리를 이용한다.
DBM_LOG_CACHE_COUNT	로그를 기록할 메모리 segment를 몇 개로 운용할지 설정한다.	NVDIMM의 경우 해당 NVDIMM의 개수
DBM_LOG_CACHE_SIZE	로그를 기록할 메모리 segment 한 개당 최대 크기이다.	NVDIMM의 경우 해당 NVDIMM의 크기
DBM_LOG_CACHE_FLUSH_INTERVAL	메모리로부터 강제로 디스크로 flush를 수행하는 주기이다.	최소 3초 이상
DBM_LOG_CACHE_EMPTY_INTERVAL	메모리에서 빈 로그 블록을 만날 경우 얼마만큼 대기한 후에 이 빈 블록을 무시할지를 설정한다.	최소 3초 이상

위와 같이 cache mode를 설정하면 세션이 commit 시점에 로그를 파일에 바로 기록하지 않고 공유 메모리나 NVDIMM 장치에 기록한다. 이 과정에서 자신의 로그를 기록할 위치를 할당받는 순간에 경합이 발생할 수 있지만 기본적으로 디스크에 기록하는 기본 설정에 비해 성능이 더 나을 수 있다.
NVDIMM은 비 휘발성 메모리를 의미하며 정전이 되더라도 자체 배터리와 SSD 조합을 통해 데이터를 보존할 수 있다. 사용자는 NVDIMM 장치를 사용하여 성능과 안정성을 최대화 할 수 있다.

Cache segment에 기록된 트랜잭션 로그들은 dbmLogFlusher라는 process에 의해 디스크의 로그 파일 형태로 기록이 된다. 따라서 dbmLogFlusher의 동작이 느리거나 cache segment size를 너무 작게 설정하면 cache에 로그를 기록할 공간을 할당하기 위해 트랜잭션 대기가 발생할 수 있으므로 cache segment 크기와 dbmLogFlusher의 동작 주기를 적절하게 설정해야 한다.

추가적으로 사용자가 로그에 대해 archiving 동작을 설정하려면 다음 프로퍼티들을 설정해야 한다.

Property name	설명	Value
DBM_ARCHIVE_ENABLE	archive 운영 설정 여부	true
DBM_ARCHIVE_PATH	archiving 될 로그가 저장될 경로	절대경로

Archive mode를 설정하면 dbmCkpt라는 process에 의해 체크포인트가 동작할 때 반영 완료된 로그파일이 DBM_ARCHIVE_PATH로 이동된다. 만일 이 설정이 false로 되어 있으면 해당 로그파일은 자동으로 삭제된다.

관련 Process

디스크 로깅 방식으로 운영하려면 항상 다음과 같은 프로세스들이 구동되어야 한다.

Process name	설명
dbmCkpt	체크포인트를 수행하여 기록이 완료된 로그 파일을 정리하고 데이터 파일을 생성한다.

Process name	설명
dbmLogFlusher	Log cache mode를 설정할 때 메모리의 로그를 디스크의 로그 파일 형태로 기록한다. (Cache mode가 아니면 구동하지 않는다.)

제약사항

다른 모든 메모리 DBMS와 마찬가지로 트랜잭션 로그를 메모리에 기록하고 주기적인 디스크 I/O를 통해 로그 파일을 만든다. 그러나 이 동작 주기 중간에 장애가 발생하면 직전 동작이 완료된 직후 발생한 메모리 상의 모든 트랜잭션 로그는 유실된다.
이를 방지하기 위해 성능이 매우 느려지는 것을 감수하고 DBM_DISK_COMMIT_WAIT 라는 프로퍼티를 제공한다. 이 프로퍼티는 로그를 기록할 때마다 해당 로그가 디스크로 완전히 flush 된다는 것을 보장한다. 이 설정은 디스크 기반의 DBMS와 비교하여 변경 연산을 처리할 때 성능 차이가 없으므로 고성능이 목적인 환경에서는 권장하지 않는다.

고려사항

응용 프로그램 코드 변경

메모리 기반인지 디스크 기반인지에 따라 코드를 변경할 필요는 없다.

HW 용량

다음 사항들을 고려해야 한다.

HW 고려 대상	설명
Memory size	Cache mode로 운영할 때 segment의 합계만큼 할당한다.
LogFile 보관 용량	체크포인트 동작 주기에 따라 필요한 보관 용량이다.
DataFile 보관 용량	체크포인트에 의해 생성되는 데이터 파일을 보관하는데 필요한 용량이다.
Archive 보관 용량	체크포인트가 archive file을 보관할 때 필요한 용량이다.

Log cache mode에서 공유 메모리를 설정하여 운영할 경우 기존 데이터 저장공간 외에 별도의 로그 저장 공간이 생성되는데 이 공간은 dbmLogFlusher에 의해 비워질 때까지 유지되므로 공간을 충분하게 할당해야 한다.

dbmCkpt에 의해 체크포인트가 발생하면 해당 시점에 대량의 I/O가 발생한다. 이는 system 성능에 영향을 미칠 수 있으므로 유휴 시간에 처리하는 방식을 고려할 수 있는데 이 경우 로그 파일이 지속적으로 증가하게 되므로 이를 보관할 수 있는 용량을 확보해야 한다.

장애 복구

디스크 로깅 방식에서 장애 복구는 다음 절차대로 진행된다. 여기서 각 예제는 active 장애가 발생하여 standby로 전환하는 경우에 대해 설명한다. 반대의 경우도 동일한 절차를 통해 failback 할 수 있다.

1. 디스크를 standby로 전환한다.
2. Standby에서 dbmCkpt를 통해 강제로 체크포인트를 수행한다.
3. Standby에서 "startup" 명령을 통해 데이터를 복구한다.
4. 데이터를 확인한 후에 서비스를 재개한다.

2번 단계는 직전 체크포인트 시점까지의 로그 파일만 데이터 파일에 반영된 상태이므로 이후에 기록된 로그 파일 전부를 디스크로 반영하려면 강제로 체크포인트를 수행하여 데이터 파일을 완성해야 한다. 그렇지 않으면 마지막 체크포인트 시점의 로그 파일까지만 반영되므로 데이터 정합성이 깨진다.

3번 단계는 dbmMetaManager를 구동한 후 내부에서 "startup" 명령을 수행하여 전체 instance/ table의 모든 데이터를 복구하는 과정이다. 따라서 standby는 초기화된 상태이어야 한다.

장애 복구의 과정들은 Linux의 failover를 담당하는 제품 내 failover script 상에 2, 3 단계별 명령어를 각각 등록하여 자동화할 수 있다.

5.3 REPLICATION 방식

GOLDILOCKS LITE는 네트워크를 이용한 실시간 이중화 방식을 제공한다.
SYNC와 ASYNC라는 두 가지 유형으로 동작을 설정할 수 있다.

- SYNC 방식은 트랜잭션이 커밋되는 시점에 standby 측에 데이터가 반영된 후 client가 응답을 받는 구조이다.
- ASYNC 방식은 트랜잭션이 커밋되는 시점에 buffer에 누적하고 client가 응답을 받는 구조이다. Buffer에 누적된 동기화 데이터는 다음과 같이 특정 조건에 맞을 경우 전송된다.
    - 이중화 버퍼를 모두 기록하여 더 이상 기록할 수 없는 경우 전송
    - 이중화 버퍼가 기록된 이후 1초 이내에 추가적 기록이 없는 경우 전송

이중화 운영을 위한 환경설정

각각의 master/ slave 측에서는 다음과 같은 환경 변수 등이 설정 되어야 한다.

[MASTER 측]
DBM_REPL_ENABLE=1
DBM_REPL_TARGET_PRIMARY_IP=xxx.xxx.xxx.xxx
DBM_REPL_TARGET_PORT=29001

[SLAVE 측]
DBM_REPL_LISTEN_PORT = 29001

이중화 대상 테이블 등록 및 dbmReplica 구동

이중화 대상은 전송 측인 master에서 등록한다.

```
dbmMetaManager(DEMO)> CREATE REPLICATION TABLE t2, t3;
success
dbmMetaManager(DEMO)> SELECT  * FROM dic_repl_table;
--------------------------------------------------------------------------------
INST_NAME  : DEMO
TABLE_NAME : T2
--------------------------------------------------------------------------------
INST_NAME  : DEMO
TABLE_NAME : T3
--------------------------------------------------------------------------------
2 row selected
```

이중화 수신 측인 slave에서는 dbmReplica를 구동한다.

```
shell> dbmReplica -i demo
```

사용자 프로그램의 동작

응용 프로그램에서는 dbmInitHandle을 수행하는 시점에 이중화를 준비한다.
이 과정에서 개별 프로그램은 내부적으로 이중화 관련 전송 로그 반영 여부에 대한 Ack 수신 thread를 생성한다.
dbmPrepareTable을 수행하는 과정에서 테이블이 이중화 대상인지 판별한다.
dbmCommit 시점에서 이전에 수행된 DML 중에 테이블이 이중화 대상인 것들만 모아 slave로 전송한다.

이중화 동기화 mode	Description
ASYNC	commit 시점에 slave에 반영 여부를 확인하지 않는다.
SYNC	commit 시점에 slave에 반영된 결과를 확인한다.

이중화 동기화 mode는 성능과 관련된 설정이다. Local transaction의 성능이 중요한 경우 async로 설정하고, 데이터 동기화 수준이 더 중요한 경우라면 sync로 설정한다. 이 설정은 환경 변수로도 제어할 수 있으므로 프로그램마다 동기화 수준을 개별로 제어할 수 있다.

미전송 로그 반영

네트워크 장애가 발생했을 때 이중화로 전송이 불가능한 시점에는 master 측에서 각 세션들이 직접 미전송 로그를 기록한다.
장애가 복구된 후 사용자가 직접 이중화 로그 전송 처리 명령을 수행하면 데이터는 slave에 반영된다.
만일 미전송 로그를 공유 디스크에 기록하도록 설정하면 slave측에서는 미전송 로그를 직접 반영한 후 서비스를 재개할 수 있도록 하는 기능을 제공한다.

6.

연동가이드

6.1 연동가이드

주의사항

- 본 챕터에서는 ODBC/JDBC 연동에 관하여 설명한다.
- ODBC API 방식은 Linux 기반의 unixODBC driver manager를 통해 연동하거나 shared library의 fuction을 직접 호출하여 사용할 수 있다.
- JDBC API 방식은 type1 (Jdbc-Odbc-Bridge) 방식만 제공한다.
- ODBC/JDBC APIs에 대해서는 일부 spec만 제공한다.
- LITE는 AUTO_COMMIT mode를 제공하지 않는다. 다만, 일부 tools과의 호환을 위해 SetAutoCommit이 성공으로 반환되도록 동작한다.

DATA TYPE Mapping

GOLDILOCKS Lite의 source data type과 ODBC/JDBC의 타입은 다음과 같이 설정된다.

SOURCE DATA TYPE	ODBC TYPE	JDBC TYPE	C TYPE
short	SQL_SMALLINT	short	SQL_C_SSHORT
integer	SQL_INTEGER	int	SQL_C_SLONG
char	SQL_CHAR	java.lang.String	SQL_CHAR
long	SQL_BIGINT	bigint	SQL_C_DOUBLE
float	SQL_DOUBLE	double	SQL_C_DOUBLE
double	SQL_DOUBLE	double	SQL_C_DOUBLE
date	SQL_TIMESTAMP	java.sql.Timestamp	SQL_C_TIMESTAMP

ODBC APIs

현재 버전에서 제공되는 ODBC APIs는 아래 표와 같다. 자세한 API spec은 ODBC specification의 sql.h를 참고한다.

ODBC API	설명
SQLAllocHandle	-
SQLConnect	-
SQLDriverConnect	-
SQLGetInfo	SQL_DRIVER_ODBC_VER SQL_ODBC_VER SQL_SCROLL_OPTIONS SQL_FETCH_DIRECTION SQL_CURSOR_COMMIT_BEHAVIOR SQL_CURSOR_ROLLBACK_BEHAVIOR
SQLGetTypeInfo	SQL_ALL_TYPES
SQLGetConnectAttr	SQL_ATTR_AUTOCOMMIT SQL_ATTR_ODBC_VERSION
SQLSetConnectAttr	SQL_ATTR_AUTOCOMMIT: 성공으로 반환되지만 상태를 바꾸지는 않는다. (Non AutoCommit mode만 가능하다.)
SQLPrepare	-
SQLExecute	-
SQLExecuteDirect	-
SQLBindParameter	-
SQLBindCol	-
SQLFetch	-
SQLNumResultCols	-
SQLColAttribute	SQL_DESC_CATALOG_NAME SQL_COLUMN_NAME SQL_DESC_NAME SQL_DESC_LABEL SQL_DESC_DISPLAY_SIZE SQL_DESC_LENGTH SQL_COLUMN_LENGTH SQL_DESC_OCTET_LENGTH SQL_DESC_CONCISE_TYPE SQL_DESC_TYPE_NAME SQL_DESC_SCHEMA_NAME SQL_DESC_UNSIGNED SQL_COLUMN_NULLABLE
SQLRowCount	-
SQLGetData	-
SQLFreeHandle	-

ODBC API	설명
SQLDisconnect	-
SQLEndTran	-
SQLTransAct	-
SQLGetDiagRec	-
SQLError	-
SQLDescribeCol	-
SQLNumParams	-
SQLDescribeParam	-
SQLParamData	-
SQLPutData	-
SQLTables	-
SQLColumns	-
SQLPrimaryKeys	-

6.2 ODBC 설정

GOLDILOCKS Lite의 ODBC/JDBC 동작을 위해서는 먼저 unixODBC driver manager가 정상적으로 설치되어야 한다.

unixODBC Driver Manager 설정

unixODBC driver manager 설치

시스템에 unixODBC driver manager가 설치되지 않은 경우 다음 명령으로 설치한다.

```
root> yum install unixODBC unixODBC-devel
```

DSN 파일 설정

unixODBC driver manager를 통해 접속할 정보를 DSN 파일에 기록한다.

```
[LITE]
HOST = 127.0.0.1
PORT = 27584
```

```
INSTANCE = demo
DA_MODE = true
DRIVER = /mnt/md1/ssd_home/lim272/new_lite/pkg/lib/libdbmCore.so
```

항목	설명
HOST	원격지의 IP를 기술한다.
PORT	원격지 dbmListener 의 port를 기술한다.
INSTANCE	접속할 instance name을 기술한다
DA_MODE	TRUE일 경우 DA 방식으로 접근한다. 원격지일 경우 FALSE로 설정한다.
DRIVER	libdbmCore.so 의 절대경로를 기술한다.

위의 DSN 파일을 명시적으로 ODBCINI 환경변수에 설정한다.

```
export ODBCINI=${HOME}/.odbc.ini
```

다음의 unixODBC driver manager에서 제공하는 "isql"을 이용하여 접속 테스트를 수행한다.

```
[11:09:15 lim272@lympc /mnt/md1/ssd_home/lim272/new_lite/pkg/sample]
$ isql lite
+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| quit                                  |
|                                       |
+---------------------------------------+
SQL> select table_name from dic_table limit 10;
+-----------------------------------------------------------------+
| TABLE_NAME                                                      |
+-----------------------------------------------------------------+
| DIC_INST                                                        |
| DIC_TABLE                                                       |
| DIC_COLUMN                                                      |
| DIC_INDEX                                                       |
| DIC_INDEX_COLUMN                                                |
| DIC_SEQUENCE                                                    |
| DIC_REPL_INST                                                   |
| DIC_REPL_TABLE                                                  |
| DIC_CAPTURE_HOST                                                |
| DIC_CAPTURE_TABLE                                               |
```

```
+------------------------------------------------------------------+
SQLRowCount returns 10
10 rows fetched
SQL>
```

6.3 JDBC 설정

Jdbc-odbc-bridge jar

JDBC type1에 대해서는 JRE 1.7까지만 제공되지만 JRE 상위버전에서는 jar파일을 이용하여 사용할 수 있다.

다음 링크를 통해서도 다운로드 할 수 있다.
https://github.com/dbeaver/jdbc-odbc-bridge-jre7

Java 환경에서는 ${JAVA_HOME}/lib/ext 경로에 위의 bridge jar 파일을 위치시킨다.
다음은 jdbc.jar 이름으로 설치하는 예이다.

```
$ ls -lrt /etc/alternatives/jre_1.8.0_openjdk/lib/ext/jdbc.jar
-rw-r--r-- 1 root root 1447077  8월 23 10:57 /etc/alternatives/jre_1.8.0_openjdk/lib/ext/jdbc.
jar
```

Class 및 URL 설정

ODBC 설정을 참고하여 DSN 파일을 지정하고 ODBCINI가 해당 파일을 가리키도록 설정한 경우, 다음 예제를 통해 j
ava에서도 사용할 수 있다.

```
try
{
        Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
        System.out.println( "find class" );
        con = DriverManager.getConnection( "jdbc:odbc:lite", "test", "test" );
        System.out.println( "connect ok" );
} catch(Exception e)
{
}
```

CLASS는 "sun.jdbc.odbc.JdbcOdbcDriver" 으로 기술한다.
DSN의 section을 "lite"라고 기술한 경우, URL은 "jdbc:odbc:section-name" 으로 기술한다.

bridge jar 파일은 tool에 연동해야 할 경우 jre/lib/ext 가 아닌 add jar file 방식을 통해서만 동작하는 경우도 있다.

6.4 PYTHON3 연동가이드

python3.6 이상에서 연동 가능하며 사전에 pyodbc를 설치해야 한다.
다음 방식으로 설치할 수 있다.

```
shell> pip3 install pyodbc
```

- 테스트를 위한 사전 DDL 수행 예제

```
dbmMetaManager> initdb;
dbmMetaManager> set instance dict;
dbmMetaManager> create instance demo;
dbmMetaManager> set instance demo;
dbmMetaManager> create table t1 (c1 int, c2 float, c3 char(20), c4 long, c5 double, c6 date, c
7 short);
```

- python 예제 코드

```
import pyodbc;
import datetime;
## DSN 방식을 통해 연결
conn = pyodbc.connect( 'dsn=lite', ansi=True, autocommit=False )
conn.setdecoding(pyodbc.SQL_CHAR, encoding='utf8')
conn.setencoding(encoding='utf8', ctype=pyodbc.SQL_CHAR)
## DDL 예제
cursor = conn.cursor()
cursor.execute( "truncate table t1" );
conn.commit()
## Direct Execute 예제
cursor.execute( "insert into t1 values (1, 2.45, 'y', 100, 300.67891, sysdate, 22)")
```

```
conn.commit()
## Prepare Execute 예제
a1 = 10
a2 = 20.23
a3 = "c"
a4 = 999992913
a5 = 200.12345
#a6 = datetime.datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S.%f')
a6 = datetime.datetime.now()
a7 = 32;
print( "a6 = " , a6)
#cursor.execute( "insert into t1 values (?, ?, ?, ?, ? ,to_date(?, 'yyyy-mm-dd hh:mi:ss.s6'),
? )", (a1, a2, a3, a4, a5, a6, a7))
for i in range(3000):
  a1 = i
  a2 = a2 + i;
  a4 = a4 + i;
  a5 = a5 + i;
  cursor.execute( "insert into t1 values (?, ?, ?, ?, ? ,?, ?)", (a1, a2, a3, a4, a5, a6, a7))
  conn.commit()
## Table Description 예제
for row in cursor.columns(schema='DEMO', table='T1'):
  print(row.table_schem, "," , row.table_name, "," , row.column_name , "," , row.data_type,
",", row.type_name, ",", row.column_size, "," , row.sql_data_type )
## Select Fetch 예제
result = cursor.execute( "select * from t1" );
for row in result:
  print( row.C1, "," , format(row.C2, '.3f'), "," , row.C3, "," , row.C4 , "," , format(row.C
5, 'f'), "," , row.C6, "," , row.C7 )
cursor.close()
conn.close()
## 수행결과 예시
[11:43:28 lim272@lympc /mnt/md1/ssd_home/lim272/new_lite/pkg/sample]
$ python3 pytest.py
a6 =  2024-08-30 11:47:27.366667
 , T1 , C1 , 4 , integer , 38 , 4
 , T1 , C2 , 8 , float , 38 , 8
 , T1 , C3 , 1 , char , 20 , 1
 , T1 , C4 , -5 , long , 38 , -5
 , T1 , C5 , 8 , double , 38 , 8
 , T1 , C6 , 11 , timestamp , 16 , 11
```

```
 , T1 , C7 , 5 , short , 38 , 5
1 , 2.450 , y , 100 , 300.678910 , 2024-08-30 11:47:27.000366 , 22
0 , 20.230 , c , 999992913 , 200.123450 , 2024-08-30 11:47:27 , 32
1 , 21.230 , c , 999992914 , 201.123450 , 2024-08-30 11:47:27 , 32
......
2997 , 4492523.000 , c , 1004485416 , 4492703.123450 , 2024-08-30 11:47:27 , 32
2998 , 4495521.000 , c , 1004488414 , 4495701.123450 , 2024-08-30 11:47:27 , 32
2999 , 4498520.000 , c , 1004491413 , 4498700.123450 , 2024-08-30 11:47:27 , 32
[11:47:28 lim272@lympc /mnt/md1/ssd_home/lim272/new_lite/pkg/sample]
$
```

6.5 JDBC 연동가이드

jdbc에서 연동할 경우 앞서 설명된 사전 ODBC 설정이 필요하다.

JDBC 예제코드

- 테스트를 위한 사전 DDL 수행 예제

```
dbmMetaManager> initdb;
dbmMetaManager> set instance dict;
dbmMetaManager> create instance demo;
dbmMetaManager> set instance demo;
dbmMetaManager> create table t1 (c1 int, c2 float, c3 char(20), c4 long, c5 double, c6 date, c
7 short);
```

- jdbc 예제코드

```
import java.sql.*;
public class jdbcSample {
  public static void main( String[] args ) {
    String url = "jdbc:odbc:lite";
    String query = "select * from t1";
    Connection con;
    PreparedStatement pstmt;
    Statement stmt;
```

```
    ResultSet rs;
    try
    {
        System.out.println( "start" );
        Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
        System.out.println( "find class" );
        con = DriverManager.getConnection( url, "test", "test" );
        System.out.println( "connect ok" );
        stmt = con.createStatement();
        System.out.println( "create stmt ok" );
        stmt.execute( "truncate table t1" );
      //pstmt = con.prepareStatement( "insert into t1 values (?, ?, ?, ?, ?, to_date(?, 'yyyy-
mm-dd hh:mi:ss.s3'))" );
        pstmt = con.prepareStatement( "insert into t1 values (?, ?, ?, ?, ?, ?, ?)" );
        pstmt.setInt( 1, 100 );
        pstmt.setDouble( 2, 200.1234 );
        pstmt.setString( 3, "xyz" );
        pstmt.setLong( 4, 123456789 );
        pstmt.setDouble( 5, 50.678901 );
        pstmt.setTimestamp( 6, new java.sql.Timestamp(System.currentTimeMillis()));
        pstmt.setShort( 7, (short)22 );
        pstmt.executeUpdate();
        pstmt.setInt( 1, 200 );
        pstmt.setDouble( 2, 300.1234 );
        pstmt.setString( 3, "abc" );
        pstmt.setLong( 4, 923456789 );
        pstmt.setDouble( 5, 150.678901 );
        pstmt.setTimestamp( 6, new java.sql.Timestamp(System.currentTimeMillis()));
        pstmt.setShort( 7, (short)32 );
        pstmt.executeUpdate();
        con.commit();
        rs = stmt.executeQuery( "select * from t1" );
        System.out.println( "execute select ok" );
        while( rs.next() )
        {
            System.out.println( rs.getInt(1) + ", " + rs.getFloat(2) + ", " +  rs.getString(3)
+ ", " + rs.getLong(4) + ", " + rs.getDouble(5) + ", " + rs.getTimestamp(6) + ", " + rs.
getShort(7) );
        }
    }
    catch( Exception ex )
```

```
    {
        System.out.println( "err: " + ex.getMessage() );
        ex.printStackTrace();
    }
  }
}
```

6.6 DBEAVER 연동가이드

dbeaver는 database management를 위한 open-source tool이며 본 문서에서는 community version을 이용한 연동에 대해 설명한다. (테스트는 21.x 버전을 기준으로 설명한다.)
dbeaver의 설치는 해당 사이트를 참조하도록 한다.

- GOLDILOCKS Lite는 window library를 제공하지 않기 때문에 Linux에서 사용할 경우 x-window 환경으로 구동 가능해야 한다.
- GOLDILOCKS Lite의 경우 dbeaver의 unique index 목록에서 모든 index 정보를 보여준다.
- 외부키, 참조키 등은 지원하지 않는다.
- DDL에서 출력된 구문은 범용 DDL 구문으로써 LITE에서는 호환되지 않는다.

일부 환경에서 x-window 화면 깜빡임이 심할 경우에는 다음의 환경변수를 설정하고 구동한다.
export SWT_GTK3=0

드라이버 등록

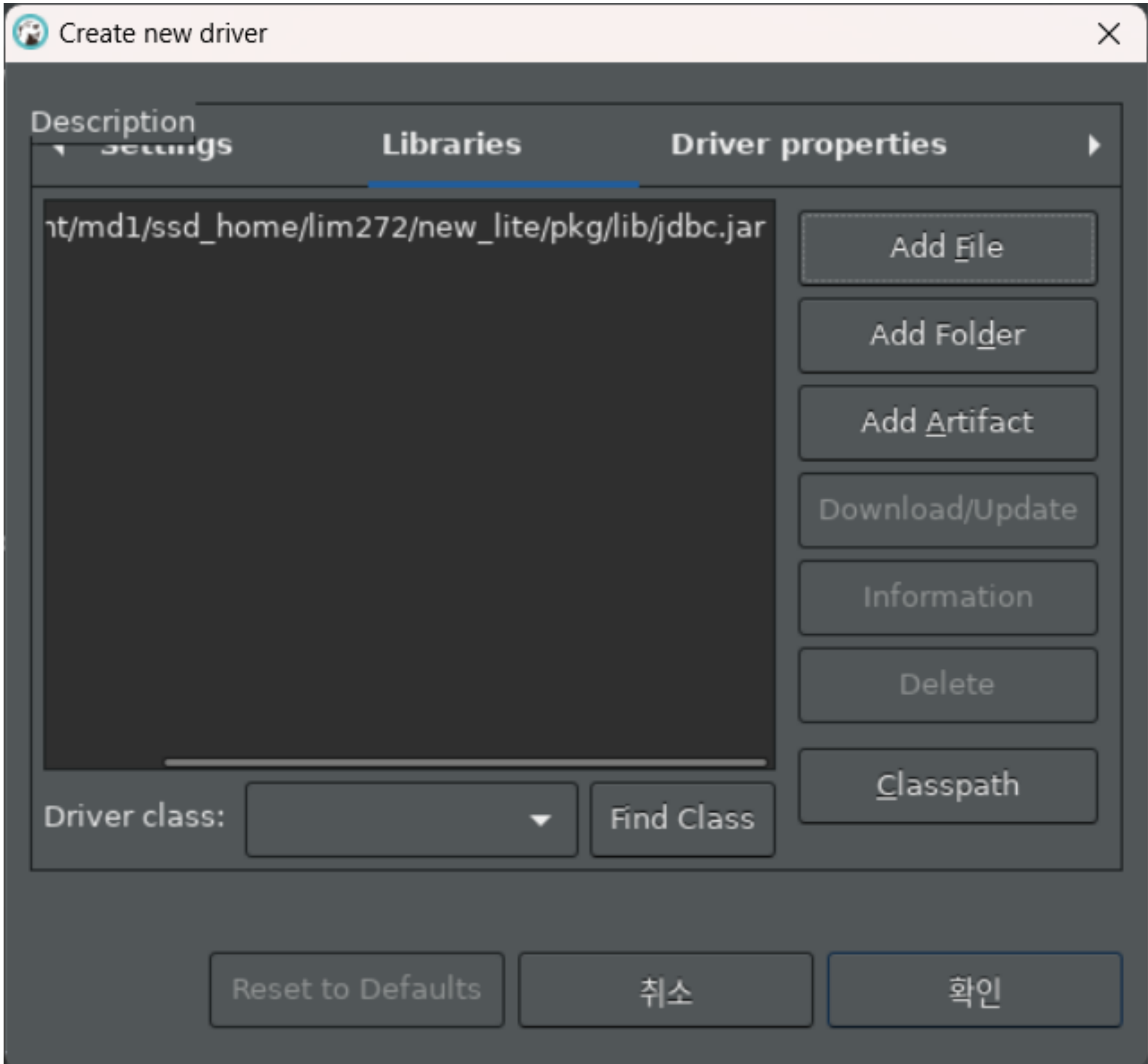
데이터베이스 → 드라이버 관리자 메뉴를 열고 NEW를 선택한 후 아래의 내용을 참고하여 등록한다.

다음의 bridge jar 파일도 등록한다.

그림 1 Create New Driver

jdbc-odbc-bridge.jar 파일을 등록한다.

그림 2 add jar file



새 데이터베이스 연결

데이터베이스 → **새 데이터베이스 연결**을 선택하고 앞서 등록한 "LITE"을 이용하여 연결을 생성한다.

그림 3 New Database

Connect to a database

Select your database

새로운 Connection을 만듭니다. 아래 목록에 해당하는 데이터베이스가 없다면 드라이버 매니저를 통해 새로운 드라이버를 등록할 수 있습니다.

Type part of database/driver name to filter

Sort by: Title Score

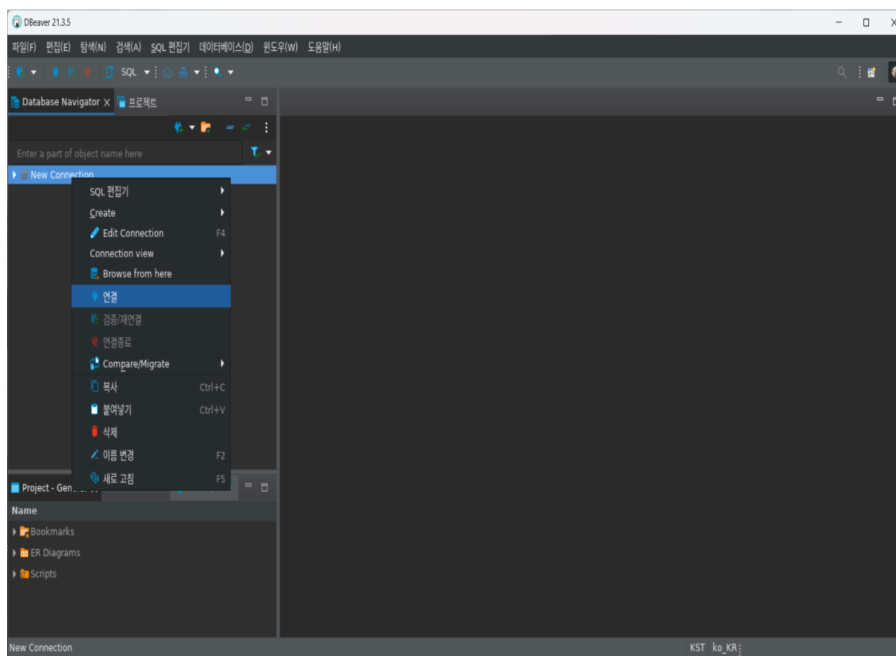
All
Popular
SQL
NoSQL
Analytical
Timeseries
Embedded
Hadoop / BigData
Full-text search
Graph databases

H2GIS Server
HANA
HSQL Embedded
HSQL Server
Informix
Ingres
InterSystems IRIS
Jennifer
Kognitio
LITE
Machbase
MaxDB
Mimer SQL
MonetDB
MS Access (UCanAccess)
MySQL (Old)
MySQL 5
NDB Cluster
Neo4j
Netezza
NuoDB
OceanBase
Ocient
OmniSci (formerly
Open Distro Elasticsearch
OpenEdge
OrientDB
Pervasive SQL
PostgreSQL (Old)
PrestoDB
PrestoSQL
Redshift

Test Connection ...
< 이전(B) 다음(N) > 취소 완료(F)

화면 왼쪽의 **New connection**에서 오른쪽 마우스 클릭을 통해 **연결**을 수행한다.

그림 4 Connect



연결이 정상적으로 수행되었다면 다음과 같이 사용할 수 있다.

그림 5 navigation

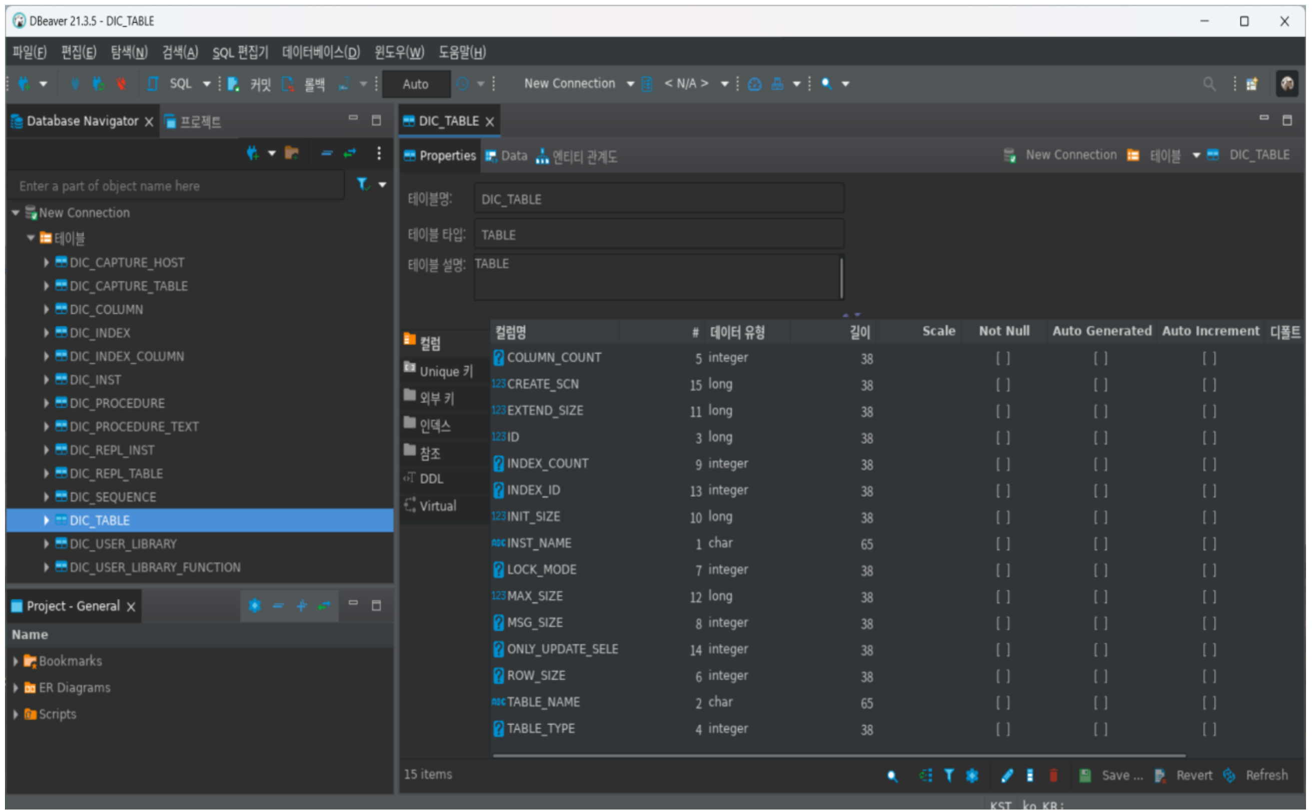
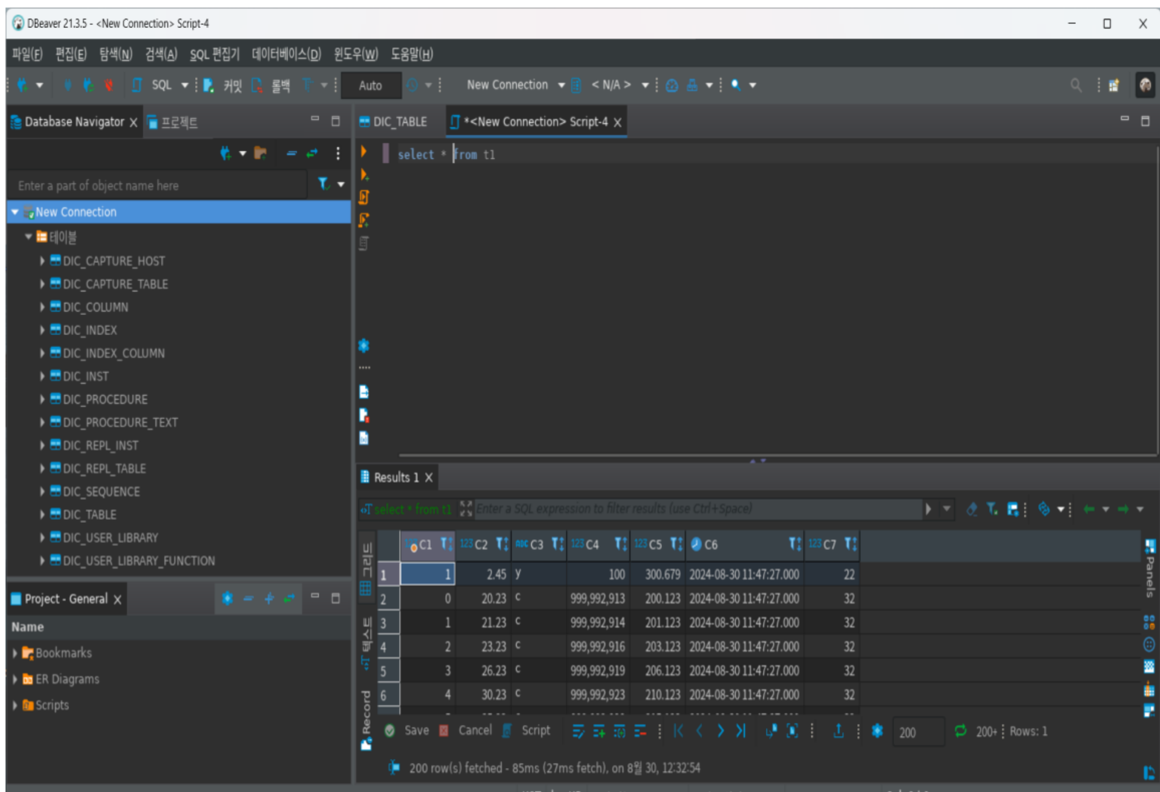


그림 6 SQL



6.7 Grafana 연동가이드

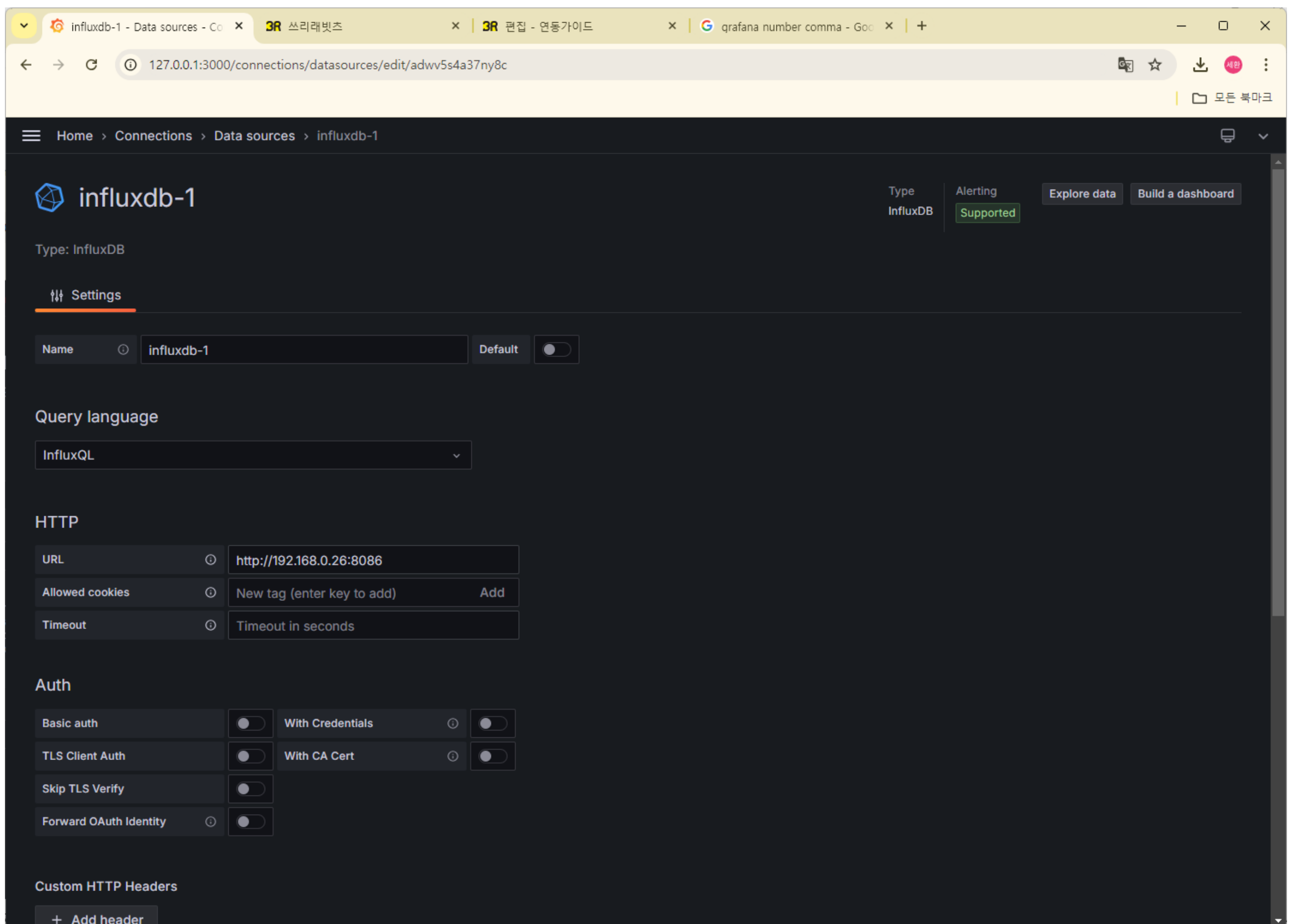
Grafana는 monitoring 데이터를 직관적으로 시각화 할 수 있는 기능을 제공하는 open source tool 이다. 본 절에서는 GOLDILOCKS Lite를 모니터링할 수 있도록 grafana를 설정하는 방법을 설명한다.

- grafana는 visualization tool이며 설치는 다음을 참고한다. (**https://grafana.com/**)
- influxdb는 monitoring data에 대한 시계열 정보를 저장하고 grafana와 연동되는 repository이며 설치는 다음을 참고한다. (**https://www.influxdata.com/**)
- 선재소프트는 위 두 개 제품의 설치/ 운영/ 문제에 대한 제반 사항을 지원하지 않는다.

influxdb DataSource 설정

Import 할 DataSource에는 "influxdb-1" 이란 이름의 data source를 사용하고 있다. 이를 위해 datasource를 미리 만들어야 한다.

그림 7 datasource



Import Dash-Board json

grafana가 설치되고 influxdb가 운영 중인 것을 전제로 하고 먼저 grafana에 GOLDILOCKS Lite용 dash board를 import한다.
dashboard json file의 위치는 ${DBM_HOME}/sample/grafana/LITE_MON_DASHBOARD.json 파일이다.
다음 화면처럼 grafana import 메뉴를 구동하고 파일을 업로드한다.

그림 8 Import

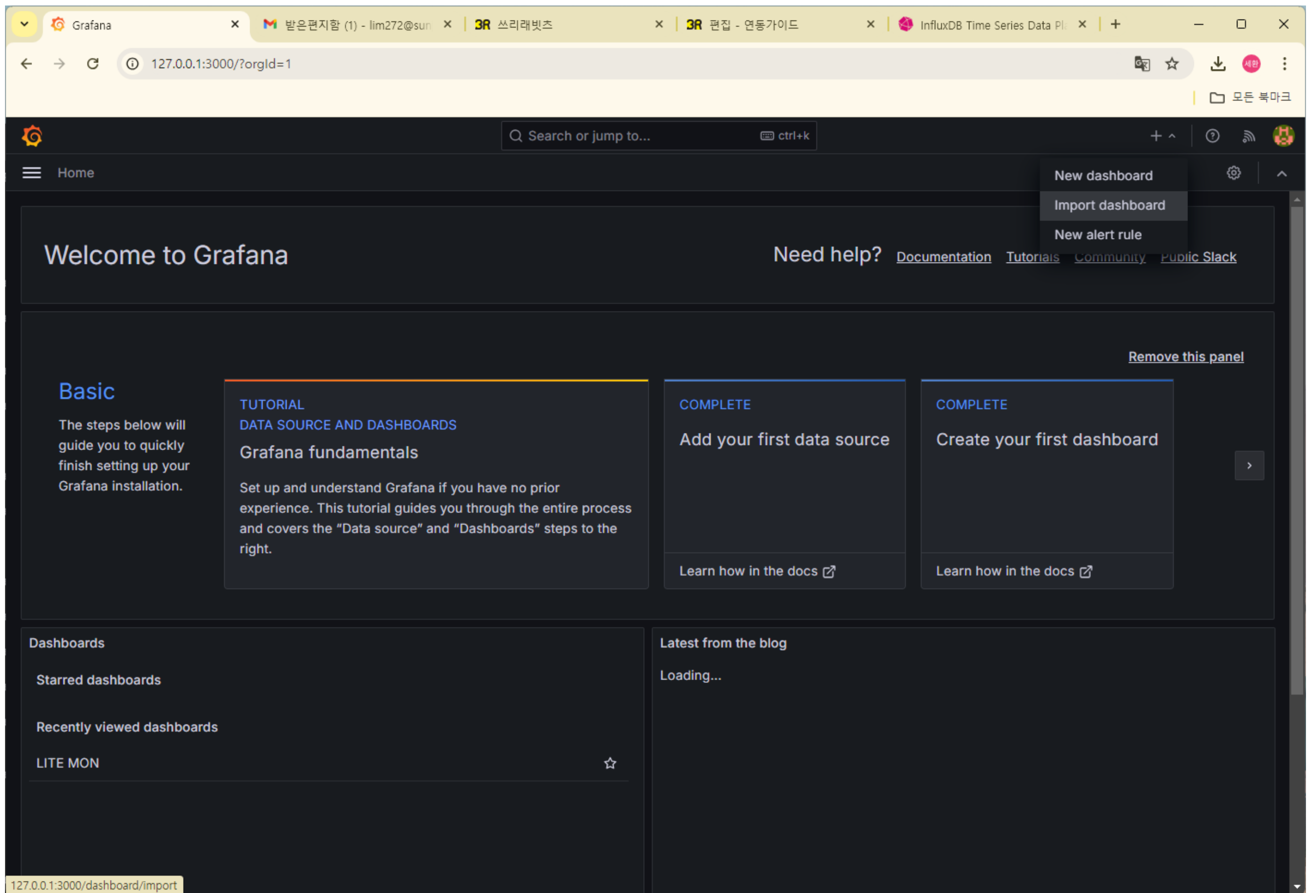
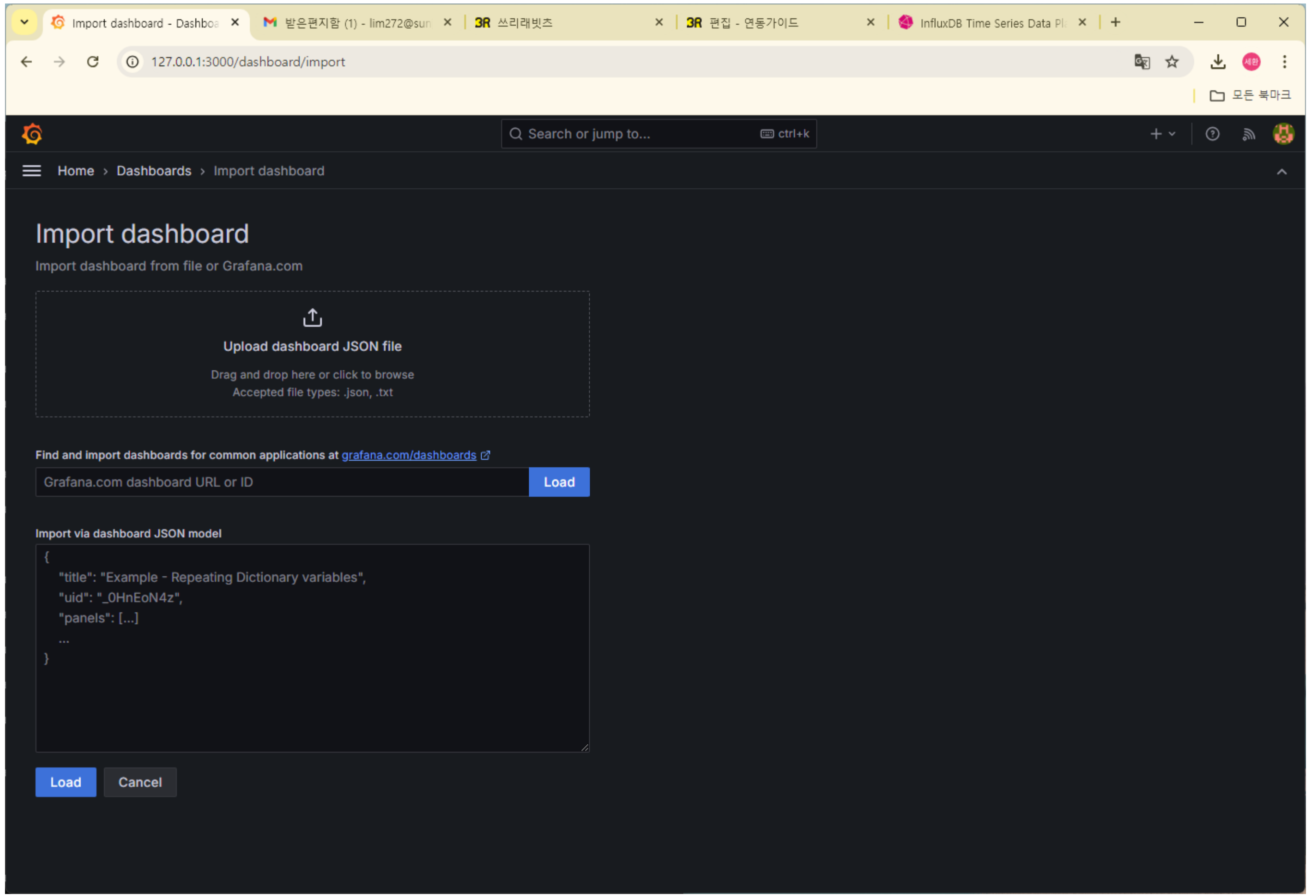
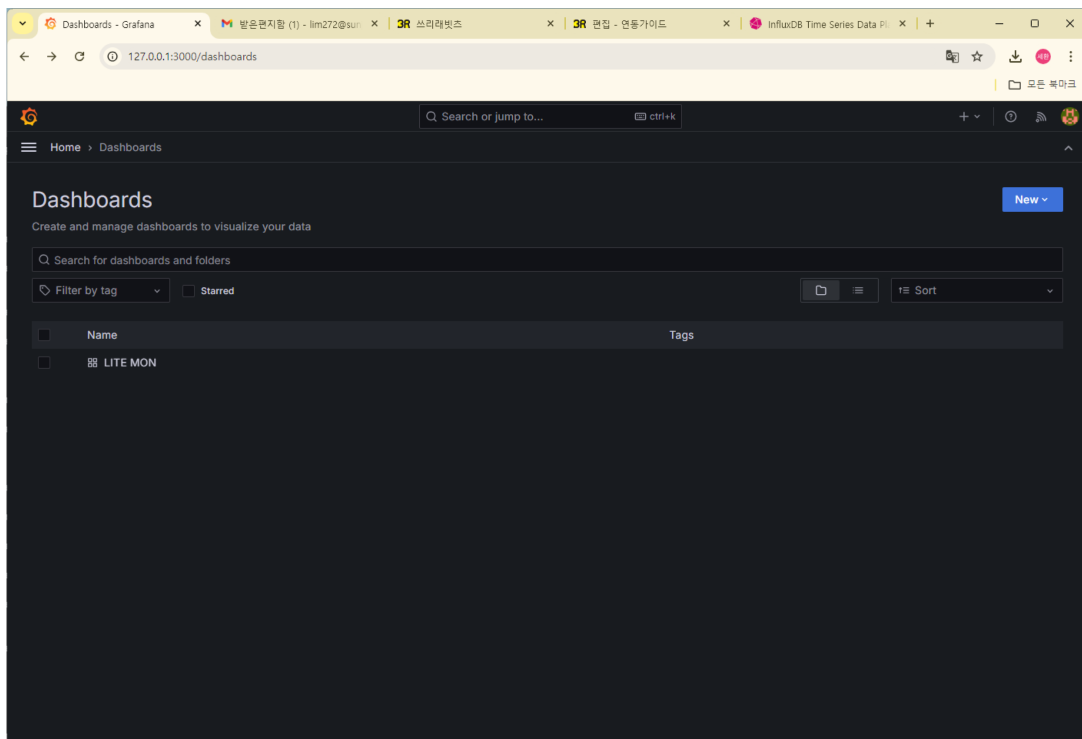


그림 9 upload json



정상적으로 upload 되면 다음과 같이 dash board가 추가된다.

그림 10 dashboard list



Monitoring Data 생성

아직은 monitoring data가 존재하지 않기 때문에 dash board 내에 출력되는 데이터는 없다. 이를 위해 ${DBM_HOME}/pkg/sample/liteMon.sh 을 주기적으로 수행하거나 background로 실행하도록 한다. (이 때 influxdb는 구동된 상태이어야 한다.)

liteMon.sh 에서 사용자는 다음의 사항을 수정해야 한다.

```
$ cat liteMon.sh
#################################
## CONFIG
#################################
INFLUX_IP="192.168.0.26"          # influxdb 가 운영되는 IP
INFLUX_PORT=8086                  # influxdb 의 Listen Port (default:8086)
INFLUX_DB="dev1"                  # influxdb 에 생성한 database name
INTERVAL=5                        # second (데이터 수집 주기)
```

수정 후 다음 명령으로 수행할 수 있다.

```
shell> sh liteMon.sh once      ## test를 위한 1회성 동작
shell> nohup sh liteMon.sh &   ## 반복수행
혹은, crontab에 동작주기를 설정하여 "liteMon.sh once" 로 실행하는 방법도 가능
```

influx에 정상적으로 데이터가 적재되면 다음 예시와 같은 화면을 확인할 수 있다.

그림 11 sample