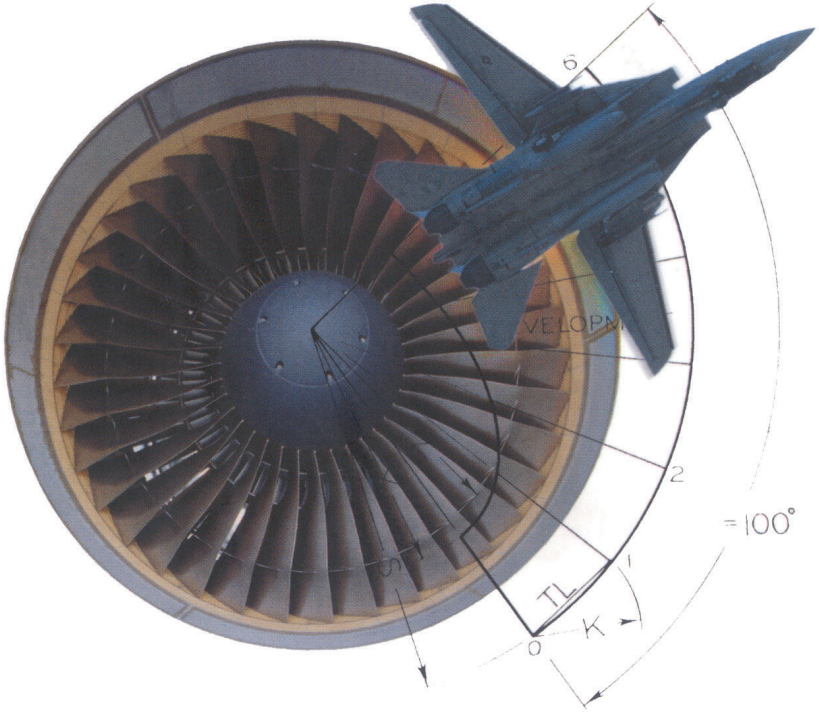
최신의 완벽한 Informix 데이터베이스 관리자 안내서

INFORMIX POWER REFERENCE



- Informix 종합 안내서
- 성능 및 신뢰성 극대화
- 아키텍처, 개발, 관리의 모든 정보
- 실용적인 Informix-SQL™ 및 Informix-4GL™

ART TAYLOR

Inoformix Power Reference

Art Taylor

목차

개정판 서문 xxi

감사의 말씀 xxii

제 1 장 Informix 제품 개요 1

Informix 제품의 역사 1
C-ISAM 2
관계형 데이터베이스와 Informix-RDS 3
Informix-SQL Engine(SE) 3
Informix Dynamic Server7(IDS7) 5
Informix Dynamic Server9(IDS9) 6
자료형 확장성 6
상속 7
사용자 정의 함수 7
Extended Parallel Server(XPS) 7
Informix 도구 8
Informix 모니터링 도구 8
Informix 언어 도구 9

Informix 인터넷 개발 도구 9
Informix 제품 아키텍처 10
디스크 입출력 10
메모리(버퍼) 11
스레드(프로세스) 12
통신 12
병렬 데이터베이스 작업 12
요약 13
다음장 미리보기 13

제 2 장 Informix의 기본개념 및 서버 아키텍처 15

IDS 기본개념 15
공유 메모리 버퍼 15
다중 스레드와 가상 프로세서 16
스레드 작동 18
Informix 데이터베이스 로깅 20
물리로그 20
논리로그 22
체크포인트 23
청크 및 DBSPACE 24
데이터 스토리지 구조 24
Informix 자료형 25
Smallint 25
Integer 26
Smallfloat 26
Decimal 26
Money 28
Float 28

Date 28
Datetime 29
Interval 30
Character 31
Varchar 31
특수 자료형 31
미러링 32
데이터 복제 33
시스템 예약 페이지 34
PAGE_PZERO 35
PAGE_CONFIG 35
PAGE_CKPT 35
PAGE_DSP 36
PAGE_PCHUNK 36
PAGE_MCHUNK 36
PAGE_ARCH 36
root DBSPACE 37
시스템 카탈로그 37
sysmaster 데이터베이스 38
Informix 서버 아키텍처 40
IDS 아키텍처 : 디스크 입출력 41
IDS 아키텍처 : 데이터베이스 메모리 42
공유 메모리의 상주 분할영역 43
공유 메모리 버퍼풀 46
논리 로그 버퍼 46
물리 로그 버퍼 47
데이터 복제 버퍼 48
LRU 관리 48
공유 메모리의 가상 분할 영역 49
IDS 아키텍처 : 스레드(프로세스) 53
IDS 아키텍처 : 통신 54
엔진을 통해 쿼리 추적 55

쿼리 결과 처리 57
병렬 데이터 쿼리 및 PDQ 우선 순위 58
IDS 옵션 및 확장 기능 59
XPS의 아키텍처 59
XPS에서의 쿼리 처리 60
IDS9 아키텍처 61
IDS9에서의 상속 62
요약 62
다음장 미리보기 63

제 3 장 Informix Dynamic Server 설치 및 구성 65

IDS 구성 65
설치 준비 66
IDS 설치 67
설치 프로그램 실행 68
IDS 환경 변수 68
ONCONFIG 70
INFORMIXSERVER 71
TERMINFO 71
PATH 71
DBANSIWARN 72
DBCENTURY 72
DBDELIMITER 73
DBEDIT 73
DBLANG 73
DBFLTMASK 74
DBMONEY 74
DBPATH 74
DBPRINT 75

DBSPACETEMP 76
DBTEMP 76
DBTIME 76
FET_BUF_SIZE 76
INFORMIXSHMBASE 77
INFORMIXSQLHOSTS 77
INFORMIXSTACKSIZE 78
INFORMIXTERM 78
PDQPRIORITY 78
PSORT_DBTEMP 79
PSORT_NPROCS 79
TERM 79
TERMCAP 79
onconfig 파일 80
ROOTAME 83
ROOTPATH 84
ROOTOFFSET 84
ROOTSIZE 84
MIRROR 84
MIRRORPATH 85
MIRROROFFSET 85
PHYDBS 85
PHYSFILE 86
LOGFILES 86
LOGSIZE 86
MSGPATH 87
CONSOLE 87
ALARMPROGRAM 88
TAPEDEV 88
TAPEBLK 88
TAPESIZE 88
LTAPEDEV 89

LTAPEBLK 89
LTAPESIZE 89
STAGEBLOB 90
SERVERNUM 90
DBSERVERNAME 90
DBSERVERALIASES 90
NETTYPE 91
DEADLOCK_TIMEOUT 91
RESIDENT 92
MULTIPROCESSOR 92
NUMCPUVPS 92
SINGLE_CPU_VP 92
NOAGE 93
AFF_SPROC 93
AFF_NPROC 93
LOCKS 94
BUFFERS 94
NUMAIOVPS 94
PHYSBUFF 94
LOGBUFF 95
LOGSMAX 95
CLEANERS 95
SHMBASE 95
SHMVIRTSIZE 95
SHMADD 96
SHMTOTAL 96
CKPTINTVL 97
LRUS 97
LRU_MAX_DIRTY 97
LRU_MIN_DIRTY 97
LTXHWM 98
LTXEHWM 98
TXTIMEOUT 98

STACKSIZE 99
OFF_RECVRY_THREADS 99
ON_RECVRY_THREADS 99
DRAUTO 99
DRINTERVAL 100
DRTIMEOUT 100
DRLOSTANDFOUND 100
CDR_LOGBUFFERS 101
CDR_EVALTHREADS 101
CDR_DSLOCKWAIT 101
CDR_QUEMEM 101
BAR_ACT_LOG 101
BAR_MAX_BACKUP 101
BAR_RETRY 102
BAR_NB_XPORT_COUNT 102
BAR_XFER_BUF_SIZE 102
BAR_BSALIB_PATH 103
RA_PAGES 103
RA_THRESHOLD 103
DBSPACETEMP 103
DUMPDIR 103
DUMPSHMEM 104
DUMPGCORE 104
DUMPCORE 104
DUMPCNT 104
FILEFACTOR 105
USEOSTIME 105
MAX_PDQPRIORITY 105
DS_MAX_QUERIES 105
DS_TOTAL_MEMORY 106
DS_MAX_SCANS 106
DATASKIP 106
OPTCOMPIND 107

ONDBSPACEDOWN 107
LBU_PRESERVE 108
OPCACHEMAX 108
요약 108
다음장 미리보기 109

제 4 장 IDS 디스크 관리 111

IDS의 디스크 보기 111
물리 디스크 레이아웃 112
논리 디스크 레이아웃 113
DBSPACE 유지보수 114
DBSPACE 작성 및 삭제 115
DBSPACE에 청크 추가 116
DBSPACE 미러링 사용 117
청크 삭제 118
소프트 링크를 사용하여 청크 지정 118
데이터베이스 테이블 작성 119
테이블의 로크 단위 124
분할화 125
분할화 방법 개발 127
보관을 위한 분할화 128
성능을 위한 분할화 128
보관/복구를 위한 분할화 방법 132
기타 분할화 요인 133
데이터 분산 사용 133
분산 정보를 사용하여 분할화 방법 결정 138
시간에 따라 다양한 데이터 분산 140
디스크 사용도 모니터링 141
요약 146

다음장 미리보기 147

제 5 장 IDS 엔진 모니터링 및 튜닝 149

IDS를 위한 튜닝 방법 149
onstat 유틸리티를 사용하여 IDS 모니터링 151
onstat 유틸리티를 사용하여 IDS 모니터링 151
onstat 유틸리티를 사용하여 자원 병목 현상 식별 155
onstat -g gio를 사용하여 CPU 사용 모니터링 155
onstat VP 출력의 해석 158
운영 체제 자원 모니터의 사용 159
CPU 자원 문제의 해결 159
IDS 메모리 사용 모니터링 160
IDS 모니터를 위한 onstat 옵션의 반복 162
메모리 문제 해결 163
디스크 활동 모니터링 164
디스크 문제의 해결 166
IDS 엔진의 로그 활동 167
로그 성능 문제의 해결 168
물리 로그 활동의 평가 및 튜닝 168
물리 로그 위치 169
물리 로그 크기 169
LRU 큐의 평가 170
LRU 성능의 평가 171
LRU 관리 튜닝 174
가중된 업데이트 로드에 대한 LRU 매개변수 설정 176
경미한 업데이트 로드에 대한 LRU 매개변수 설정 176
의사 결정 지원 시스템에 대한 LRU 설정 177

LRU 튜닝을 위한 반복 벤치마킹 177
체크포인트 평가 및 튜닝 178
쿼리 및 응용프로그램의 튜닝 179
onstat -g ses를 사용하여 쿼리 세션 추적 180
onstat -g ses를 사용하여 세션 활동 상태 보기 180
onstat -g sql 명령 183
onstat -g act를 사용하여 활성 스레드 모니터링 184
onstat -g iof를 사용하여 비동기 입출력의 모니터링 185
시스템 자원 문제 해결 186
CPU VP의 추가 187
쿼리 튜닝 188
동일한 디스크로 분할화 190
필터와 조각 제거 191
PDQPRIORITY 설정 192
PDQPRIORITY의 설정 및 제어 195
OPTCOMPIND 매개변수를 사용하여 최적화에 미친 영향 197
인덱스 읽기 대 연속 읽기 197
미리 읽기 옵션 198
쿼리 및 쿼리 계획 199
액세스 계획 200
조인 계획 200
해시 조인 201
평가 순서 201
비용 기반의 옵티마이저 및 set optimization 명령 201
쿼리 튜닝 202
전체 테이블 스캔의 식별 및 문제 해결 202
전체 테이블 스캔을 승인할 수 있는 경우 203
쿼리 튜닝을 위해 테이블 분할화 사용 206
분할화 스키마 206
inv_date별 분할 207
inv_flag별 분할 208

ship_to_code별 분할 209
분할화와 RAID 210
요약 210
다음장 미리보기 211

제 6 장 Informix Dynamic Server의 데이터베이스 관리 213

데이터베이스의 시작 및 중지 213
엔진 운영 모드 214
시작 문제 215
온라인 엔진 수정 216
엔진을 오프라인 모드로 가져오기 217
단일 사용자 모드로 엔진 가져오기 218
온라인(다중 사용자) 모드로 엔진 가져오기 219
CPU 자원의 추가 및 제거 220
DSS 매개변수 변경 221
공유 메모리 매개변수의 변경에 onmode 사용 222
onmode -a를 사용하여 가상 공유 메모리 추가 223
onmode -r을 사용하여 공유 메모리 버퍼를 상주 상태로 설정 223
기타 onmode 명령 224
onmode -l 명령 225
onmode -Z <transaction id> 명령 225
ommode -z <pid> 명령 225
onmode -R 명령 226
onmode -d 명령 226
IDS 데이터베이스 보안 226
데이터베이스 레벨 사용 권한 227
데이터베이스 connect 권한 227
데이터베이스 resource 권한 228
데이터베이스 dba 권한 228

사용자 역할의 작성 및 사용 229
테이블 사용 권한 230
with grant option 명령문 231
revoke 명령으로 사용 권한 제거 232
IDS 로그 관리 232
논리 로그 모니터링 233
로그 파일 백업 234
물리 로그의 크기 및 위치 변경 235
onconfig 파일의 변경 237
onparams 명령 사용 237
논리 로그의 추가 237
논리 로그 삭제 238
보관 및 복구 239
보관 레벨 239
보관 수행 240
데이터 복구 241
전체 복구 및 선택 복구 242
데이터베이스 로깅 상태의 변경 242
요약 243
다음장 미리보기 244

제 7 장 관계형 데이터베이스와 Informix-SQL 245

관계형 데이터베이스의 역사 246
관계형 데이터베이스 개념 246
엔티티, 속성 및 관계 246
정규화 247
IDS의 구조화된 쿼리 언어 248
SQL문 유형 248
확장 SQL문 249

데이터 스토리지 작성 - DDL문 250
데이터베이스 작성 250
create table 명령문 251
테이블 이름, 컬럼 이름 및 자료형 251
IDS 분할화 정의 252
데이터베이스 뷰 작성 255
뷰를 사용하여 보안 시행 256
뷰에 키워드 사용 257
전체 쿼리 참조 258
데이터베이스 테이블 삭제 259
데이터베이스 뷰 삭제 259
데이터베이스 삭제 260
alter table 명령문 260
rename column 명령문 262
rename table 명령문 263
컬럼 무결성 및 참조 제한 조건 264
테이블 레벨의 기본 키와 외부 키 정의 265
고유 컬럼 작성 266
제한 조건 이름 지정 267
데이터 액세스 - DML문 268
select 명령문 268
select 목록 269
from 절 269
where 절 270
표시 레이블 지정 271
정렬 순서 식별 271
리턴되는 최대 행 수 지정 272
레코드 그룹화 273
테이블 별칭 지정 274
서브쿼리 사용 275
반복적 쿼리 277
외부 조인 사용 279

SQL 합집합 280
union절을 사용하여 값 세그먼트화 281
insert 명령문 283
update 명령문 284
delete 명령문 285
기타 SQL 286
SQL 복사 작업 286
load 및 unload 명령문 287
임시 테이블 사용 288
Informix 내장 프로시저 언어(SPL) 289
if 명령문 291
return 명령문과 returning 절 293
foreach 명령문 294
call 명령문 295
while 루프 296
for 명령문 297
begin/end 명령문 블록 299
continue 명령문 300
raise exception 명령문 301
let 명령문 302
system 명령문 304
on exception 명령문 305
define 명령문 306
SPL에서 전역 범위 변수 사용 307
exit 명령문 308
trace 명령문을 사용하여 디버그 결과 생성 309
프로시저에서 dbinfo 함수 사용 309
IDS 데이터베이스 트리거 311
create trigger 명령문 312
삽입 트리거 312
삭제 트리거 314
업데이트 트리거 316

트리거의 유용성 318
트리거 내에서 내장 프로시저 실행 318
트리거에서 when절 사용 319
사용자 정의 복제에 트리거 및 내장 프로시저 사용 320
IDS 문자열 처리 함수 323
trim 함수 324
substring 함수 324
substr 함수 325
replace 함수 325
lpad 함수 326
rpad 함수 326
대소문자 변환 함수 327
새로운 버전 7.3 기능 327
case 명령문 328
decode 함수 329
nvl 함수 330
메모리 상주 테이블 및 인덱스 작성 331
옵티마이저 지시문 사용 332
요약 334
다음장 미리보기 335

제 8 장 Informix-4GL 소개 337

Informix-4GL 언어 337
명령문 및 연산자 339
프록램 블록 339
언어 연산자 339
메모리 제어 339
자료형 지정 340
데이터베이스 상호 작용 340

4GL 응용프로그램의 파트 340
Informix-4GL과의 사용자 상호 작용 342
보고서 작성 및 데이터 필터링 342
Informix-4GL을 사용하여 모듈 프로그래밍 342
Informix 자료형 및 연산자 343
Informix-4GL 자료형 343
Smallint 344
Integer 344
Smallfloat 345
Decimal 345
Money 346
Float 347
Date 347
Datetime 347
Interval 348
Character 349
콤플렉스 자료형 350
배열 350
레코드 350
데이터베이스 바인딩 351
NULL 데이터 값 352
지정문의 데이터 변환 353
표현식에서의 지정 및 변환 356
문자 변환 356
산술 연산자와 숫자 변환 규칙 357
초기화 358
언어 상수 358
Informix-4GL의 흐름 제어문 359
for 명령문 360
Informix-4GL의 조건식 361
연결 연산자 362

Informix-4GL 응용프로그램에서 SQL 사용 363
SQL문을 사용하여 유형 변환 365
Informix-4GL에서의 데이터베이스 바인딩 366
스키마 1 368
스키마 2 368
prepare 명령문 369
prepare 명령문을 사용하여 최적화 370
SQL 처리 : 프론트엔드 대 백엔드 활동 370
prepare문과 declare문을 사용하여 최적화 371
Informix-4GL의 커서 374
foreach 명령문 375
SQL문에서 사용된 데이터 변수 375
Informix-4GL의 커서 376
커서 유형 377
스크롤 커서 378
스크롤 커서의 문제 380
업데이트 커서 382
삽입 커서 383
커서 닫기 384
뷰 385
모듈과 SQL 387
요약 389
다음 장 미리보기 389

제 9 장 화면 폼 및 보고서 391

화면 크기 관리 391
화면 폼 프로그래밍 392
화면 폼에서 데이터베이스 바인딩 392
formonly 화면 393
Informix-4GL로 화면 폼 처리 394

화면 폼 식별자 395
모듈성과 화면 폼 상호 작용 395
다중 윈도우 관리 396
현재 윈도우 396
display 명령문 401
display 명령문과 화면 폼 상호 작용 403
input 명령문 405
입력 필드 통과 406
현재 필드 강조 표시 408
termcap 문제 409
기능 키 재정의 410
input array에서 insert key 및 delete key 해제 410
termcap 파일에서 기능 키 추가 410
interrupt key 412
Informix-4GL 보고서 413
보고서 명령문 414
보고서 선언 415
기본 보고서 생성 416
output 섹션 418
order by 섹션 419
format 섹션 419
first page header 섹션 419
보고서에서 동적 컬럼 위치 작성 421
format 섹션 - before group of/after group of 절 422
보고서의 배치 처리 423
print 명령문 426
보고서의 텍스트 필드 관리 427
페이지 구분 427
모듈식 동적 보고서 430
보고서 문제 432
보고서의 모듈성 435
보고서 동기화 437
construct 명령문 중심의 동적 보고서 438

qry_acct 함수 439
rpt_drv 함수 440
acct_totals 보고서 441
테이블 중심의 동적 보고서 446
main 프로그램 블록과 rpt_drv 함수 448
proc_line 함수 450
sql_rpt 보고서 451
on every row 보고서 블록 454
요약 457
다음 장 미리보기 457

제 10 장 Informix-4GL의 함수 및 오류 처리 459

함수 매개변수 460
구현 460
표현식으로서의 함수 461
함수 내에서 변수 사용 462
매개변수의 변수 수를 함수로 전달 464
Informix-4GL의 반복 기능 468
반복의 예 - 부분 분해 문제 469
Informix-4GL의 오류 처리 및 도움말 메시지 475
whenever error 명령문 476
Informix-4GL을 사용하여 오류 로깅 478
whenever any error를 사용하여 폼 및 표현식 오류 트래핑 480
오류 메시지 482
일반화된 오류 처리 루틴 483
오류 프롬프트 함수 484
Informix-4GL의 도움말 메시지 485
sqlca 레코드 487
sqlerrd 배열 488

sqlwarn 문자열 489
요약 490

색인 493

개정판 서문

데이터베이스는 응용프로그램의 핵심이자 중심이며, 응용프로그램 아키텍처의 중요 구성 요소입니다. 중요한 응용프로그램 정보가 이 구성 요소를 통해 들어오고 나갑니다. 데이터베이스가 작동되지 않거나 데이터베이스에 결함이 발생하면 응용프로그램 전체에 중대한 영향을 미칠 수 있습니다.

데이터베이스는 이와 같이 중요한 부분이기 때문에 데이터베이스를 제대로 관리하는 것이 무엇보다도 중요합니다. 사용하는 데이터베이스가 올바르고 효율적으로 작동되어야 응용프로그램이 어떠한 문제도 발생하지 않고 실행될 수 있습니다. 데이터베이스가 원활하게 작동되기 위해서는 관리를 잘 해야 할 뿐만 아니라 데이터베이스의 논리적 설계와 물리적 설계가 적절하게 맞아야 합니다. 즉, 데이터베이스는 데이터베이스를 사용하는 응용프로그램의 필요에 맞도록 설계되어야 합니다.

Informix 데이터베이스의 관리자는, 엔진이 머신의 자원(CPU 및 디스크)을 어떻게 사용하는 지에서부터 데이터의 무결성이 로깅 메커니즘, 보관 및 복구 기능과 더불어 가장 중요한 작업인 데이터베이스 모니터링과 튜닝을 통해 어떻게 유지되는지에 이르기까지 엔진이 작동되는 전 과정을 이해해야 합니다. 이 책은 처음에 기본 적인 용어와 개념에서 시작하여 엔진 아키텍처를 설명하는 순서로 각 주제에 대해 차례로 다룹니다.

Informix 응용프로그램의 개발자가 응용프로그램을 올바로 설계하기 위해서는 데이터베이스의 작동 과정을 이해해야 합니다. 정적 데이터베이스 분할화를 통한 유연한 데이터 구성은 Informix 데이터베이스의 중요한 기능이며, 올바로 이해하고 사용 될 경우 다양한 응용프로그램의 성능을 향상시킬 수 있습니다. 이 책에서는 Informix 분할화를 자세하게 다루며, 분할화라는 중요한 기능을 사용하여 데이터 검색 성능을 개선하고 데이터를 특정 디스크 위치에 배치시킴

으로써 데이터 관리를 용이하게 하는 방법에 대해 설명합니다.

Informix 응용프로그램 개발자나 데이터베이스 관리자는 Informix의 구조화된 쿼리 언어(SQL)를 사용하여 Informix 데이터베이스와 상호 작용합니다. Informix-SQL을 올바로 이해하고 현명하게 사용하면 개발과 관리에 들이는 노력이 현격히 줄어들 것입니다. 이 책에는 데이터 처리 및 데이터 정의에 가장 일반적으로 사용되는 Informix-SQL문이 다양한 예와 함께 설명되어 있습니다.

가장 지속성이 강한 제4세대 언어 가운데 하나가 Informix-4GL입니다. Informix-4GL 개발 언어는 프로시저 및 비프로시저 프로그래밍 명령문을 잘 결합시켜 놓은 언어로, 이 언어를 사용하여 완전한 응용프로그램을 쉽게 개발할 수 있습니다. 4GL은 보고서 개발, 데이터베이스 관리 및 데이터 웨어하우스와 데이터 마트에서 일반적으로 볼 수 있는 응용프로그램 변환과 필터링에 뛰어납니다. 이 책에서 이 4GL에 대해 자세하게 다루겠습니다.

컴퓨터라는 말을 사용하기는 하지만, 일반적으로 컴퓨터에 대한 프로그램을 작성하고 컴퓨터를 관리하는 주체는 컴퓨터가 아니라 사람입니다. 올바른 정보에 기초한 신중한 설계, 프로그래밍 및 관리는 컴퓨터 응용프로그램이 작성 시의 기본사항입니다. 이를 위해서는 사용하는 개발 도구를 확실히 잘 알고 있어야 합니다. Informix의 개발자 또는 관리자인 여러분은 이 책을 통해 책에 제시된 Informix 제품들에 대한 확고한 지식 기반을 갖출 수 있을 것입니다.

감사의 말씀

이 책을 쓰도록 도움을 주신 모든 분들께 감사를 드립니다. 특히 많은 조언을 해주었으며 응원해 주고 이해해 준 가족들 Carolyn, Hannah, Eric 그 가운데서도 특히 Carolyn에게 고마운 마음을 전합니다.

Harold Davis와 Seth Grimes 등 기술 담당 편집자들에게도 모두 감사를 드립니다. 특히 이 책이 나오기까지 함께 해 주었으며 예리한 기술 관련 전문 지식으로 책의 내용이 간결하고 주제를 벗어나지 않도록 해 준 Paul Gancz와 Dave Kosenko에게 고마움을 표합니다.

이 작업을 함께 한 Informix 출판사의 Judy Bowman에게도 적극적 지원과 도움에 대해 특별한 감사를 드립니다.

그리고 마지막으로, Prentice-Hall 제작 팀의 Ralph Moore와 Kerry Reardon에게 고마움을 전합니다. 그들의 도움으로 이 책에 수록된 모든 정보가 간결하고 명확하며 보기 좋은 모양새를 갖출 수 있었습니다.

1장

Informix 제품 개요

Informix Software의 제품은 Informix가 1980년에 Relational Database Systems라는 이름으로 창립된 이래 양적, 질적으로 계속해서 성장해 왔습니다. 초기 Informix Software는, 데이터에 대한 인덱스 순차 액세스 방식(ISAM)을 제공하는 C 언어 함수 세트인 C-ISAM을 시작으로 지속적인 발전을 거듭해 왔습니다. 이 제품의 핵심은 엔터프라이즈의 데이터 관리 필요성을 지원하도록 구성된 강력한 데이터베이스 서버라는 점입니다. 최근에는 일련의 서버 옵션이 추가되어 객체-관계형 데이터 액세스 및 확장 자료형을 지원하며, 대량 병렬 및 클러스터 서버를 지원하는 또 다른 옵션도 추가되었습니다. 다양해진 개발 도구 세트와 확장된 보관 및 복구 도구 세트를 통해 제품의 완결성이 한층 높아졌습니다.

Informix Software의 방향을 이해하기 위해서는 현재까지의 모든 과정을 이해하는 것이 도움이 됩니다. 다음 절에서는 Informix 제품이 어떻게 시작하여 현재의 제품까지 이르렀는지에 대한 설명 등 Informix 제품의 역사에 대해 알아보겠습니다. 마지막 절에서는 현재 제품 세트를 구성하는 제품에 대해 설명합니다.

Informix 제품의 역사

Informix Software의 창립 이래 계속된 컴퓨터 산업의 성장 및 발전과 함께 Informix의 제품도 많은 성장과 발전을 거듭했습니다. 1980년에 당시 Relational Database Systems라는 이름의 Informix

는 단일 제품 출시를 시작으로, 판매 수요가 높은 소프트웨어와 관련된 비즈니스 모델을 개발하여 당시 부상하던 UNIX 서버 시장에 내놓았습니다. 당시, 많은 컴퓨터 산업 분석가들이 개방 시스템 컴퓨터가 IBM, Digital Computer 및 기타 회사에서 제공하는 독점 운영 체제와 경쟁하지 못하리라고 생각했습니다. 그러나 Informix의 창립자인 Roger Sippl은 UNIX의 운영 체제에 대한 확고한 믿음 즉, UNIX의 운영 체제가 앞으로 많은 수익을 가져오리라는 믿음을 버리지 않았습니다.

Informix 제품군은 표 1-1에서 알 수 있듯이 독점 응용프로그램 프로그래밍 인터페이스(API)와 쿼리 언어를 기반으로 하는 일련의 제품에서 출발했습니다. 그러나 곧 개방 표준형으로 전환하였고 이는 이후에도 이것은 Informix의 주요 전략이 되었습니다.

표 1-1 Informix 제품군

제품	아키텍처	릴리스 시기	특징1
C-ISAM/RDS	단일 프로세스아키텍처	1985	독점 쿼리 언어
ISQL	이중 프로세스 아키텍처	1986	개방 표준 쿼리 언어(SQL) 및 UNIX 파일 시스템 사용
Informix-4GL	이중 프로세스 아키텍처	1986	4세대 개발 언어
Turbo	이중 프로세스 아키텍처	1987	내부 파일 시스템(원시 디스크 사용), SQL
Online	이중 프로세스 아키텍처	1989	Turbo와 동일
DSDA	다중 스레드	1992	병렬 처리
IDS	다중 스레드	1997	복제, 고성능 로더
XPS	다중 스레드	1996	MPP 및 클러스터 서버 환경 지원 및 최적화
IDS9	다중 스레드	1996	객체 - 관계형 기능 지원 : 자료형 확장, 사용자 정의 함수, 테이블 및 row 자료형 상속

C-ISAM

Informix의 첫 번째 제품인 C-ISAM은 온라인 트랜잭션 처리(OLTP) 데이터베이스 서버와 4GL 이전에 나온 제품으로 rich 자료형을 지원하지 않았습니다. 하지만 C-ISAM은 ISAM을 사용하여 데이터를 작성하고 액세스하는 C 언어 루틴의 컬렉션으로 지금도 사용

하고 있습니다. C-ISAM을 구성하는 기능은 파일의 데이터를 처리하는 파일 설명자와 포인터를 사용하여 파일처럼 데이터베이스 테이블에 액세스합니다. 인덱스를 사용하여 데이터 검색 성능을 향상시킬 수 있지만 C-ISAM은 이러한 인덱스에 명시적으로 액세스해야 합니다. SQL 데이터베이스 액세스의 경우 암시적으로는 인덱스를 사용할 수 없습니다. (실제로 Informix 데이터베이스는 이러한 초기 기능에서 생성된 루틴을 사용하여 데이터베이스에서 테이블을 읽고 액세스합니다.)

관계형 데이터베이스와 Informix-RDS

1980년대 초기에는 계층형, 네트워크형, 관계형 등 데이터베이스 유형이 여러 가지 있었습니다. 일련의 관련 테이블에 데이터를 저장하는 관계형 데이터베이스는 이러한 유형 가운데 하나에 지나지 않았습니다. 관계형 데이터베이스의 이론적 개념이 견고하고 타당하다고 생각하기는 했지만, 기존의 데이터베이스에 좀더 익숙한 다수의 사람들은 관계형 데이터베이스가 기존 데이터베이스의 레벨보다 수행 능력이 떨어질 것이며 두 개 이상의 테이블을 조인하는 데 필요한 추가의 디스크 액세스로 인해 상당히 비용이 많이 들 것이라고 생각했습니다.

그러나 Informix의 경영진은 관계형 데이터베이스의 개념에 대한 신념을 갖고 C-ISAM의 후속 제품으로 C-ISAM 기능 라이브러리를 기초로 한 관계형 데이터베이스 제품을 개발했습니다. RDS(Relational Database System)이라는 이름의 이 제품에는 관계형 데이터베이스, 독점 쿼리 언어, 데이터베이스의 데이터 작성과 액세스에 사용되는 메뉴 구동 도구 및 ACE라는 보고서 작성 도구가 포함되었습니다.

Informix-SQL Engine(SE)

Informix-RDS가 성공을 거두기는 했지만 관계형 데이터베이스 산업에 대한 늘어나는 관심은 표준 쿼리 언어로 향하고 있었습니다. 당시는 처음 IBM에서 개발한 구조화된 쿼리 언어(SQL)가 산업 표준이 되어가는 과정이었습니다. 이러한 개발 움직임이 일어나는 상황에서 Informix는 1985년에 데이터베이스 쿼리에 산업 표준 SQL 언어를 사용하는 이식 가능한 데이터베이스 서버 프로그램을 개발했습니다. 이 제품이 Informix-SQL Engine이며 Informix-Standard

Engine(SE)이라고도 합니다.

이전의 제품들은 단일 프로세스 아키텍처를 사용했지만 Informix-SE 제품은 이중 프로세스 아키텍처를 사용했습니다. 이중 프로세스 아키텍처란 데이터에 액세스하는 클라이언트 프로세스와 데이터베이스 액세스를 관리하는 서버 프로세스로 구성된 아키텍처를 의미합니다(그림 1-1 참조).

또한 이러한 프로세스를 프론트엔드와 백엔드 프로세스라고도 합니다. 클라이언트 프로그램은 사용자와 인터페이스로 접속할 수 있으며 사용자의 액세스를 더욱 간편하게 하는 방식으로 개발이 가능하므로 서버 프로세스를 데이터 액세스 제어, 데이터 검색 최적화 및 데이터 무결성 제공 등의 특정 기능에 집중시킬 수 있습니다.

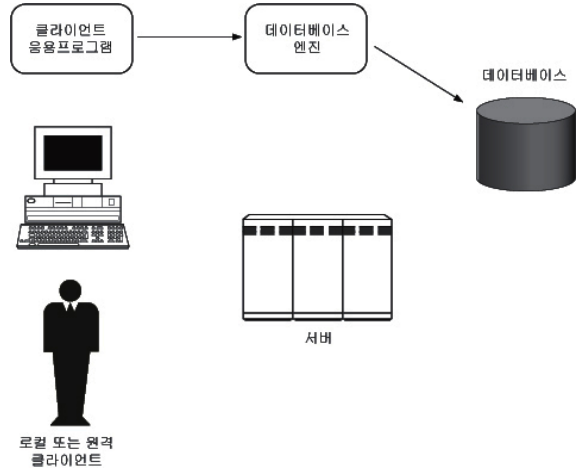


그림 1-1 Informix-SE 이중 프로세스 아키텍처

이중 프로세스 아키텍처의 클라이언트 프로그램인 프론트엔드는 프로세스 간 통신(IPC: interprocess communication)을 사용하여 백엔드 프로그램과 통신합니다. UNIX 소켓과 공유 메모리 연결은 가장 일반적으로 사용되는 IPC 액세스 방법입니다.

이 이중 아키텍처를 사용하여 얻은 개발 효과는 아주 컸습니다. 이중 프로세스 아키텍처 구현을 통해, C-ISAM과 RDS의 단일 프로세서 아키텍처를 사용했을 때 단단하게 밀착되어 있던 응용프로그램의 클라이언트 및 서버 부분을 효과적으로 분리할 수 있었고 클라이언트 및 서버 응용프로그램을 개별적으로 개발하고 최적화할 수 있었습니다. 그리고 중요한 점은 클라이언트와 서버 프로그램이 동

일한 머신에 상주하지 않아도 된다는 점입니다. 그러나 초기 UNIX 컴퓨팅 시기에는 종종 동일한 머신에 상주하기도 했습니다.

클라이언트 프로그램은 별도의 서버나 PC에서 실행할 수 있고 TCP/IP(Transmission Control Protocol/Internet Protocol) 소켓을 IPC 메커니즘으로 사용하여 원격 머신에서 실행되는 데이터베이스 서버 프로그램과 통신할 수 있으며 데이터베이스가 동일한 머신에 있는 것처럼 데이터베이스의 데이터를 원격으로 검색하고 처리할 수 있었습니다. 바로 이것이 클라이언트/서버 컴퓨팅의 시작입니다.

Informix Dynamic Server7(IDS7)

4.0 버전부터 Informix-Turbo의 이름을 Informix-Online으로 변경하고 이 릴리스에서 사용 가능한 많은 변경사항을 반영하였습니다. 반영된 변경사항을 통해 온라인 백업, 동일 머신의 다중 인스턴스, Informix-Online 인스턴스 간 통신과 향상된 옵티마이저 및 다양한 성능 최적화 기능을 지원할 수 있게 되었습니다.

후속 플랫폼상의 5.x와 6.0 버전에서 Informix Dynamic Server 7(IDS7)이 출현하였습니다. Informix Dynamic Server7(IDS7)은 다중 스레드와 정적 분할화를 사용했고 부분적으로 레벨의 복제가 가능했습니다. 내부 다중 스레드를 통해 프론트엔드 서비스에 필요한 프로세스 수를 줄일 수 있었기 때문에 Informix 서버 실행에 필요한 Unix 자원을 감소할 수 있었습니다. 메모리를 할당하고 관리하는 운영 체제 호출을 제거하고 이러한 작업을 엔진에 지정하기 위한 코드를 작성함으로써, 엔진 코드에 대한 제어 범위가 대폭 향상되고 이식 가능성이 향상되었습니다.

내부 다중 스레드를 사용함으로써 작업이 수행될 시기에 대한 제어 기능이 향상되고 엔진 자원에 대한 경쟁을 줄이는 방법이 마련되었습니다. 가장 중요한 것은 병렬 작업을 제어할 때의 세분화 정도가 높아졌다는 점이며, 이는 내부 다중 스레드를 도입한 주요 이유이기도 했습니다.

Informix Dynamic Server7(IDS7)이 확고한 제품이기는 하지만 빠른 속도로 변하는 데이터베이스 소프트웨어 시장에서 정지된 상태로 머물러 있지 않았습니다. Informix는 데이터베이스 엔진 제품에 적극적으로 기능을 추가했습니다. 6.0 이후의 서버 릴리스에서는 추가 병렬 작업, 쿼리 우선 순위 지정, 조인 작업이 가능해졌습니다. 추가 데이터 복제 기능, 감사 기능 및 이벤트 경보 기능도 추가되었습니다. 무엇보다도 중요한 사항은 객체-관계형, 대량 병렬 및 클러스터를 지원하기 위한 몇 가지 옵션이 추가되었다는 점이며 이러한

사항에 대해서는 다음 절에서 설명합니다.

Informix Dynamic Server9(IDS9)

Informix 데이터베이스 서버에 추가된 기능 중 가장 중요한 기능 가운데 하나가 Informix Dynamic Server9(IDS9) 엔진에 있는 객체-관계형 기능 세트입니다. 1995년에 Informix Software는 Illustra Software를 매입하여 Informix에서 Illustra 객체-관계형 데이터베이스를 액세스할 수 있었습니다. Illustra의 제품에는 사용자 정의 자료형, 테이블, row 자료형 상속, Datablades를 통한 확장성 등 아주 훌륭한 객체-관계형 기능이 다수 있었습니다. 하지만 이 데이터베이스에는 Informix-Online에 존재하는 기능이면서 엔진의 엔터프라이즈 클래스 사용에 필요한 기능인 충분한 확장성 및 데이터 무결성은 지원하지 않았습니다. Informix 개발자의 목표는 Informix Dynamic Server7(IDS7)을 Illustra 서버와 병합하여 뛰어난 객체-관계형 기능은 물론 확장성과 데이터 무결성 기능을 갖춘 서버를 구축하는 것이었습니다. 이러한 노력의 결과가 IDS9입니다.

IDS9 제품에는 다수의 중요한 객체-관계형 기능이 있습니다. 가장 중요한 기능은 다음과 같습니다.

- 상속
- 자료형 확장성
- 사용자 정의 함수

이러한 구성 요소는 다음 절에 설명되어 있습니다.

자료형 확장성

자료형 확장성 기능을 통해 사용자는 자료형을 데이터베이스에 추가할 수 있습니다. 사용자는 이제 대부분의 관계형 데이터베이스 공급자가 제공하는 ANSI SQL 자료형 이외의 자료형도 사용할 수 있게 되었습니다. 자료형 확장성에는 데이터베이스 행을 정의한 후 또 다른 테이블을 생성하는 등 효과적으로 row 자료형을 자료형으로 처리할 수 있는 row 자료형 정의와 같은 기능이 제공됩니다.

또한 사용자 정의 자료형을 생성하여 opaque 데이터 객체(엔진은 데이터 객체의 구성에 대해서는 전혀 알지 못함)가 데이터베이스에 존재하도록 할 수 있습니다. 그런 후 사용자 정의 서버 함수를 작성하여 이러한 자료형에서 작동되도록 합니다.

상속

상속은 코드 재사용의 주요 구성 요소 중 하나입니다. 데이터베이스 테이블과 행 정의에 적용될 경우에는, 사전에 정의된 테이블이나 row 자료형이 정의되고 있는 행이나 테이블의 기준이라는 것을 나타내는 기능에 해당됩니다.

사용자 정의 함수

사용자 정의 함수 작성은 IDS9의 가장 강력한 기능 중 하나입니다. 사용자는 C에서 Java에 이르는 다양한 언어로 함수를 정의할 수 있습니다. 이러한 함수에는 엔진 버퍼 캐시의 데이터를 처리할 수 있는 함수가 포함되며, IPC를 통해 데이터를 검색하는 데 드는 오버헤드, 대부분의 미들웨어 솔루션에서 발생하는 오버헤드를 피할 수 있습니다.

Extended Parallel Server(XPS)

XPS(확장 병렬: extended parallel) 서버는 단일의 고속 네트워크로 연결된 병렬 시스템인, 대량 병렬 처리(MPP) 머신에 확장 가능한 성능을 제공할 수 있도록 설계되었습니다. (많은 시스템에서 이러한 고속 네트워크의 데이터 전송 속도는 머신 버스 속도에 근접하므로 노드 간 상호 통신에 병목 현상은 거의 발생하지 않습니다.) 이러한 서버는 대형 의사 결정 지원 시스템(DSS)에서와 마찬가지로 대량의 데이터를 신속하게 스캔해야 할 경우에 매우 바람직합니다.

MPP는 노드의 컬렉션입니다. XPS를 통해 IDS의 인스턴스가 노드에서 실행됩니다. 이러한 인스턴스는 코서버(coserver)로 알려져 있습니다. 각각의 코서버는 고유한 자원 세트를 관리합니다. 요청 관리자가 각 코서버에 존재하며 각 데이터 요청을 코서버 전체에 어떻게 분배해야 할지를 결정합니다. 그러면 쿼리 옵티마이저, 데이터 사전 관리자 및 스케줄러가 각 노드에서 함께 작동되어 요청을 처리합니다. XPS 서버는 정적 분할화의 개념을 확장하여(데이터를 여러 개의 디스크에 분산) 다중 코서버에 있는 디스크에서의 분할화를 지원함으로써 테이블이 별도의 시스템에서 실행되는 다중 IDS 인스턴스에 분산되도록 합니다. 이러한 환경에서 병렬 처리를 통해 별도의 머신에 있는 코서버의 자원(디스크, CPU, 메모리)을 병렬로 사용할

수 있습니다. 코서버 간 공유하는 자원이 없으므로, 경쟁이 거의 없으며 코서버를 추가할 때 XPS의 확장성은 거의 일직선으로 증가합니다. 코서버를 추가하면 성능이 거의 일직선으로 증가하므로, 제공된 작업에 코서버를 네 개 사용할 경우 두 개의 코서버를 사용하는 것에 비해 속도가 거의 두 배로 빨라진다는 것을 의미합니다.

Informix 도구

Informix는 코어 데이터베이스 제품을 보완하기 위해 수년 간에 걸쳐 개발 도구 제품을 개발했습니다. Informix는 대체적으로 Informix-4GL을 사용한 문자 기반 개발부터, Data Director 제품을 사용한 클라이언트/서버 GUI 개발 및 Informix 웹 통합 옵션을 사용하여 Informix 데이터베이스를 전적으로 활용한 웹 사이트 작성에 이르기까지 광범위한 데이터 관리 도구를 제공합니다. 이러한 도구는 Informix 데이터베이스 서버에 중요한 부속 기능입니다.

Informix 모니터링 도구

Informix는 Informix 서버를 유지보수하고 모니터링 할 수 있는 여러 도구를 제공합니다. Informix 서버는 한 때 문자 기반 개발이 일반적이었던 UNIX에 확고히 기반을 두고 있었으므로 이러한 도구 중 다수가 문자 기반이라는 것은 당연합니다. `onmonitor` 도구는 Informix 서버 인스턴스를 작성하고 유지보수하는 데 사용하며, `dbaccess`는 서버 인스턴스 내의 테이블과 데이터베이스를 작성하고 쿼리를 실행하며 결과를 리턴하는 데 사용됩니다.

그러나 업계의 추세가 GUI 클라이언트/서버 도구쪽으로 향하는 것에 부응하여 Informix는 기존의 문자 기반 응용프로그램과 기능이 동일한 완전한 GUI 응용프로그램 세트를 개발했습니다. IECC(Informix Enterprise Command Center)는 여러 Informix 서버 인스턴스를 작성하고 모니터링하는 데 사용합니다. Informix SQL Editor는 클라이언트/서버 GUI 응용프로그램을 사용하여 쿼리를 실행하고 결과를 검토하는 데 사용합니다. ROM(Relational Object Manager)은 SQL Editor와 동일한 기능을 수행하지만 IDS9 엔진에서 사용할 수 있는 rich 자료형을 인식합니다. Schema Knowledge 도구는 관계형 데이터베이스의 구성 요소를 작성하고 관리하는 데 사용합니다.

백업 및 복구는 `ontape`과 `onbar` 도구를 사용하여 관리합니다. `on`

tape 도구는 문자 기반 사용자 인터페이스를 제공하여 Informix 데이터베이스와 서버 인스턴스를 백업하며, 표준 테이프 장치를 지원합니다. onbar 유틸리티는 표준 테이프 장치와 스토리지 관리자를 모두 지원합니다.

Informix 언어 도구

Informix는 다양한 내장 언어 도구를 제공합니다. 가장 널리 사용되는 언어 도구 중 하나가 ESQL/C(Embedded SQL for C)입니다. Informix가 제공하는 다른 내장 언어에 비해 ESQL/C가 가장 널리 사용되고 있습니다. ESQL/C 제품을 사용하면 ANSI SQL문과 Informix 확장 기능을 C 언어 프로그램에 직접 내장할 수 있습니다.

Informix-4GL이 UNIX 4GL 시장의 대략 40%를 차지한 적도 있었습니다. Informix-4GL은 C 언어의 구조와 간결성은 유지하면서도 안전성과 추가 비프로시저 구문이 포함되어 간편하고 액세스가 쉽습니다. Informix-4GL에는 흐름 제어 프로시저와 조건문, 문자 기반 입력 화면을 표시하기 위한 비프로시저 명령문, 제어 중단과 출력 포맷 기능을 제공하는 보고서 작성 명령문이 모두 포함되어 있습니다. Informix-4GL은 오래 전에 개발되었음에도 불구하고 여전히 복잡한 데이터베이스 업데이트, 데이터 정리 및 변환, 문자 기반 보고 등에 대단히 유용하다는 것이 계속해서 입증되고 있습니다.

Informix 인터넷 개발 도구

1990년대 이루어진 가장 중요한 개발 중 하나는 월드 와이드 웹(WWW)과 씬 클라이언트 응용프로그램을 개발한 것입니다. 씬 클라이언트 패러다임을 이용하여, 응용프로그램을 개발한 후 실질적으로 전개 비용을 전혀 들이지 않고 전개할 수 있습니다. 사용자의 데스크탑에 호환 가능한 웹 브라우저만 있으면 됩니다. 개발이 즉각적으로 이루어집니다. 이는 개발 커뮤니티에 상당한 매력으로 다가오는 패러다임입니다.

Informix Software는 웹의 중요성을 인정하고 여러 가지 중요한 웹 도구를 도입했습니다. Informix 웹 통합 옵션으로 HTML(Hyper Text Markup Language) 확장자를 사용하여 Informix 데이터베이스에서 데이터를 검색하는 HTML 페이지를 작성할 수 있습니다. 데이터베이스 인식 목록 상자나 확인란 및 기타 HTML 형식 요소를 HTML 페이지의 일부로 작성할 수 있습니다. 데이터베이스 중심의

웹 개발로 웹 페이지 스토리지를 위한 관계형 데이터베이스의 데이터 무결성이 보장되고 트랜잭션 제어가 가능해짐으로써 사이트 관리자는 한 층 강화된 보안과 제어의 혜택을 누리게 되었습니다.

Informix 제품 아키텍처

Informix 엔진이 수행하는 작업은 간략히 말해 여러 사용자에게 데이터베이스 서비스를 제공하는 것입니다. Informix 엔진은 작업에 사용 가능한 자원을 바로 할당함으로써 이러한 작업을 수행합니다. 그러나 경쟁이 계속되는 데이터베이스 서버 산업에서, 그 목적은 자원을 효과적으로 할당하는 것뿐만 아니라 데이터베이스 성능을 최적화하고 가능한 한 CPU 자원을 거의 사용하지 않도록 하는 데에 있습니다. 이는 사용 가능한 자원을 효과적으로 관리함으로써 가능해집니다. 이러한 맥락에서 Informix 서버의 아키텍처를 자원 관리적 측면에서 보는 것이 좋습니다. 즉, 서버의 개념적 아키텍처는 다음과 같이 네 가지 기본 구성 요소로 구성됩니다(그림 1-2 참조).

- 디스크 입출력
- 메모리(버퍼)
- 스레드(프로세스)
- 통신

이러한 네 가지 구성 요소가 엔진의 모든 기능을 구성합니다. 엔진은 요청을 수신하면 요청에 대한 서비스를 제공하기 위해 스레드를 할당합니다. 그러면 이러한 스레드가 필요한 메모리 양과 디스크 입출력을 요청합니다. 프론트엔드나 클라이언트는 이 통신 계층을 사용하여 엔진과 통신합니다.

디스크 입출력

디스크 관리 구성 요소는 디스크와의 모든 상호 작용을 제어합니다. 디스크 입출력 계층은 메모리 관리 계층과 상호 작용하여 사용자가 요청한 디스크 분할 영역에 버퍼를 지정합니다. 물리 디스크 분할 영역은 DBSPACE라는 논리 디스크 공간 컬렉션에 매핑됩니다. 데

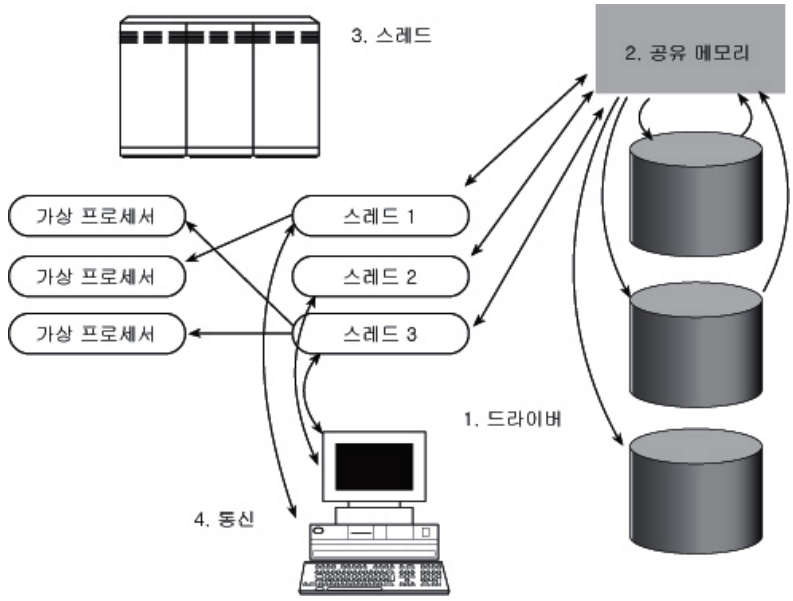


그림 1-2 IDS 엔진 아키텍처

이터베이스 테이블과 인덱스를 선택적으로 여러 DBSPACE에 매핑하여 분할된 테이블이나 인덱스를 작성할 수도 있습니다. 데이터베이스 로깅 역시 이 구성 요소에서 제어합니다.

메모리(버퍼)

Informix 서버는 디스크(캐시)에서 데이터에 대한 버퍼 지정, 데이터 정렬 및 병합, 내부 모니터링 및 제어, 통신 관리 등의 다양한 목적으로 메모리를 사용합니다. 이러한 메모리를 관리하는 것이 서버 운영의 중요한 구성 요소입니다. 서버에 메모리가 충분하지 않으면, 메모리에서 수행되어야 할 정렬이 디스크에서 수행됨으로써 성능이 저하됩니다. 메모리가 적으면 또한 쿼리 실행 중 오류가 발생하고 심각한 경우에는 서버에 오류가 발생할 수 있습니다.

Informix Dynamic Server7(IDS7) 제품 릴리스 이후로 Informix 서버의 버전에서 메모리 관리는 엔진의 필수적 요소가 되었고 따라서 엔진의 성능과 코드의 이식 가능성이 향상되었습니다. 이제 low 레벨의 메모리 할당(`malloc()`)도 운영 체제 호출을 사용하지 않고 서버 코드에서 수행될 수 있습니다.

스레드(프로세스)

Informix 서버는 고유한 내부 스레드 메커니즘을 사용하여 다중 스레드를 수행합니다. 스레드는 비중이 적고(자원 전환이 거의 필요 없음) 우선 순위가 지정되어 있지만, 관리자가 변경할 수 없는 특정 우선 순위도 있습니다. 스레드를 통해 요청한 데이터베이스 작업이 다중 스레드에 분배되어 작업 분할 영역을 병렬로 수행할 수 있습니다.

Informix 서버는 요청 중인 데이터베이스 활동과 처리되고 있는 데이터 위치에 따라 스레드 할당에 대한 의사 결정을 내립니다. 엔진에서는, 병렬 작업을 수행해도 경쟁이 유발되지 않는다고 판단되면 일반적으로 다중 스레드로 작업을 수행합니다. 이러한 방식으로 검색 중인 데이터의 분할화가 병렬 데이터베이스 작업 생성에 큰 역할을 수행하는데, 이는 여러 조각에 상주하는 데이터를 다중 스레드를 사용하여 검색 될 수 있는 반면, 하나의 조각에 위치한 데이터는 다중 스레드를 사용하지 않기 때문입니다.

통신

아키텍처의 통신 구성 요소는 엔진과 데이터베이스 엔진에 연결되는 프로세스 간 통신을 관리합니다. 정보가 엔진 사이를 이동할 때 이 계층을 통과합니다. 엔진에 연결된 각 프로세스에는 프로세스의 활동을 모니터하며 통신 계층의 제어에 영향 받는 세션 제어 블록이 있습니다. Informix 데이터베이스 통신 계층은 소켓과 공유 메모리 등의 UNIX IPC 메커니즘을 사용하여 클라이언트 응용프로그램과 통신합니다.

병렬 데이터베이스 작업

Informix 제품 아키텍처의 중요한 측면은 병렬 데이터베이스 작업 수행 기능입니다. Informix 데이터베이스에는 정적 분할화를 통해 제공되는 물리 데이터 스토리지 기반으로 데이터베이스 작업을 수행하는 기능이 있습니다.

Informix 데이터베이스 서버가 데이터베이스 작업을 수행할 때 병렬로 작업을 수행할 방법을 결정합니다. 가능한 경우, 엔진은 데이터베이스 선택, 삽입 및 업데이트를 병렬로 수행합니다. 내부 다

중 스레드 메커니즘을 사용하여 데이터베이스 작업을 여러 CPU에 걸쳐(사용 가능한 경우) 병렬로 실행할 수 있으므로 성능이 향상됩니다.

이러한 병렬 작업 기능은 Informix 엔진의 주요 특징입니다. Informix 데이터베이스에는 표현식을 사용하여 여러 개의 디스크에 걸쳐 데이터를 분할하거나 분산시키는 표현식 기반 분할화 기능이 있습니다. 이러한 표현식은 데이터를 검색할 때 엔진에서 평가하기 때문에 데이터베이스 액세스와 관련하여 지능적인 결정을 내릴 수 있습니다.

또한 데이터베이스가 다중 스레드를 사용하여 내부 기능을 수행하므로 로깅 및 공유 메모리 관리와 같은 기능에 내부 다중 스레드를 이용한 효과를 얻을 수 있습니다. 또한 IDS 엔진이 내부 다중 스레드를 사용하는 데 따른 이점은 데이터 스캔, 정렬 또는 병합을 수행하는 데이터 액세스 작업에도 적용될 수 있습니다.

다음 장에서 설명하겠지만, 옵티마이저가 쿼리에 다중 실행 스레드를 사용하도록 지정 또는 지시하는 병렬 데이터 쿼리 기능도 데이터 분할화를 고려한 기능입니다.

요약

이 장에서는 Informix 제품을 간략하게 살펴보고 제품의 출발에서 현재의 다양한 제품에 이르는 과정을 대략적으로 살펴보았습니다. Informix Software 제품군이 1980년에 Relational Database Systems라는 이름으로 출시된 이래, Informix의 강력한 제품 세트는 지속적으로 발전하여 현재 두 가지 종류의 데이터베이스 제품과 광범위한 모니터링, 보관, 복구 및 개발 도구를 갖춘 풍부하고 복합적인 제품이 되었습니다.

다음 장 미리보기

데이터베이스 서버 아키텍처에 대한 기초 지식 없이는 데이터베이스 서버의 변천 과정을 이해할 수 없습니다. 다음 장에서는 Informix 서버 아키텍처를 전체적으로 다루고 그 구성 요소에 대해서도

설명합니다. 2장에서 설명한 개념은 이후에 모니터링 및 성능 튜닝에 대한 설명과도 연관이 깊습니다.

2장

Informix의 기본 개념 및 서버 아키텍처

데이터베이스 서버 기술을 논의하기 전에 먼저 서버 아키텍처를 이해해야 합니다. 여기에서는 아키텍처를 IDS 서버의 기본 개념을 설명하는 기본 구조로 사용합니다.

이 장에서는 Informix 엔진 아키텍처 및 기본 엔진 구성 요소 개념에 대해 설명합니다. 이 책의 나머지 장들의 기본이 될 엔진을 이해하는 것이 그 목표입니다. 먼저, 기본 엔진 구성 요소에서부터 알아봅시다.

IDS 기본 개념

여러 가지 기본 개념을 이해하기 위해서는 Informix 엔진을 이해해야 합니다. 이러한 개념은 데이터를 저장하는 메모리 위치, 디스크의 논리적 구성 요소와 물리적 구성 요소의 이름, 작업 수행 시 사용하는 내부 다중 스레드 등 Informix 제품의 핵심과 관련됩니다. 다음 절에서는 이러한 핵심 개념과 기타 핵심 데이터베이스 개념에 대해 자세하게 설명합니다.

공유 메모리 버퍼

Informix 서버는 데이터를 저장하고 버퍼를 지정하며, Informix

서버의 제어 하에서 실행되는 다양한 스레드를 추적하고 제어하는 등의 광범위한 작업에 공유 메모리를 사용합니다. 머신이 고장 났을 때 이러한 공유 메모리 버퍼를 제어하고, 서버가 제어하는 디스크와 버퍼를 동기화하며, 데이터 무결성을 보장하는 일이 결코 쉬운 일은 아닙니다.

버퍼는 크기가 고정된 페이지(일반적으로 2048byte/4096byte이지만 하드웨어 플랫폼에 따라 다양함)로, 논리적으로 나뉩니다. 이러한 버퍼 페이지에는 디스크에서 페이지가 들어 있는 파티션, 페이지에 쓰여진 데이터량, 페이지에 있는 행의 크기와 수에 대한 헤더 정보가 들어 있지만, 페이지에 있는 테이블이나 행에 관한 특정 정보는 들어 있지 않습니다. 데이터 페이지의 주요 식별자는 파티션 번호나 페이지 번호입니다. 인덱스 엔트리는 페이지 번호를 참조하며 페이지 내의 특정 행을 오프셋합니다.

다중 스레드와 가상 프로세서

가장 중요한 IDS 엔진 기능 중 하나가 내부 다중 스레드입니다. 내부 다중 스레드는 동시에 실행할 수 있는 다중 실행 스레드로서 여러 작업을 엔진에서 내부적으로 실행하는 기능입니다.

다중 프로세스 동시 실행은 새로운 개념이 아닙니다. 한동안은 운영 체제가 이 기능을 수행했습니다. 그러나 운영 체제에서 실행한 것은 대개 각 실행 프로그램이 별도의 운영 체제 작업이나 프로세스를 나타내는 다중 작업이었습니다. 운영 체제가 하나의 작업에서 다른 운영 체제로 전환되는 이 컨텍스트 전환 작업에는 운영 체제가 작업 자원을 처리하는 비용이 다소 많이 듭니다. 따라서 내부 응용 프로그램 다중 스레드는 이 컨텍스트 전환을 피하려고 합니다.

프로그램 스레드는 비중이 적고 효율적인 단일 프로세스 방식을 제공합니다. 단일 프로세서 방식에서는 다수의 내부 실행 스레드를 실행하며, 이러한 스레드 간을 이동할 때 오버헤드가 거의 발생하지 않습니다. 다중 내부 스레드는 동일한 프로그램 스토리지 영역에 액세스하므로 프로세스 간 통신을 사용하여 작업(별도의 실행되는 프로그램에서 수행해야 하는 작업) 사이의 정보를 교환할 필요가 없습니다.

Informix는 비중이 적은(오버헤드가 거의 발생하지 않음) 스레드를 위한 독점 패키지를 사용하는 스레드를 구현합니다. CPU 자원 간 스레드를 간편하고 신속하게 전환할 수 있도록 스레드를 구현합니다. 이렇게 구현된 스레드로 운영 체제 오버헤드, 전반적인 메모리 요구사항, 데이터베이스 관리 시스템(DBMS) 내에서의 자원 경

쟁이 줄어듭니다. 이러한 이점은 대칭 멀티프로세서(SMP) 플랫폼과 같은 멀티프로세서를 지원하는 하드웨어 플랫폼에서 특히 실현성이 높습니다.

Informix에는 사용자 세션용으로 시작되는 세션 스레드와 데이터베이스의 하우스키핑 작업을 관리용으로 시작되는 내부 스레드, 두 가지 유형의 스레드가 있습니다. IDS 엔진은 일반적으로 클라이언트 연결을 위한 단일 세션 스레드와 데이터베이스 입출력, 로깅, 디스크에 데이터 페이지 쓰기 및 다양한 관리 작업을 수행하는 내부 스레드를 실행합니다.

Informix의 다중 스레드 구현에서 가상 프로세서(VP)는 머신 하드웨어에서 CPU가 제공하는 논리 기능을 제공합니다. 다중 VP를 구성할 수도 있습니다. VP에서 실행되는 스레드가 현재 사용할 수 없는 자원을 요청하는 경우, 이 VP를 해제하여 다른 스레드가 사용할 수 있도록 합니다. 스레드가 기다리던 자원을 사용할 수 있게 되면, 스레드는 다른 VP에서 실행을 시작할 수 있으므로 다른 VP로 이관한 것같이 됩니다.

Informix는 CPU 가상 프로세서를 표 2-1에서와 같이 여러 클래스로 나눕니다.

표 2-1 가상 프로세서 클래스

가상 프로세서	목적
CPU	세션 스레드와 일부 시스템 스레드 실행
PIO	OS 파일일 경우, 물리 로그 파일에 쓰기
LIO	OS 파일일 경우, 논리 로그 파일에 쓰기
AIO	비로깅 디스크 입출력 수행
SHM	공유 메모리 통신
TLI	TLI를 사용한 네트워크 통신
SOC	소켓을 사용한 네트워크 통신
OPT	광학 디스크에 입출력 수행
ADM	관리 기능
ADT	감사 기능
MSC	기타: 매우 큰 스택이 필요한 시스템 호출 요청 관리

가장 일반적인 가상 프로세서는 CPU VP로, 이 CPU VP는 시스

템 성능에 가장 많은 영향을 줍니다. CPU VP의 수는 구성 매개변수를 사용하여 설정할 수 있고 엔진이 온라인일 때 동적으로 조정할 수 있습니다. CPU VP를 동적으로 추가할 수 있는 기능을 사용하여 CPU 자원을 시스템 로드에 따라 조정할 수 있습니다. 시스템 로드가 증가하면 추가 CPU를 늘리고 시스템 로드가 감소하면 CPU VP 수를 줄일 수 있습니다.

VP와 실제 CPU가 반드시 일대일로 대응할 필요는 없지만, 일반적으로 CPU 하나를 운영 체제에 할당하고 나머지 CPU를 단일 VP에 할당하는 방법을 사용합니다. 이러한 방법을 사용하면, CPU가 네 개인 시스템에서 CPU 하나는 운영 체제에, 나머지 CPU는 세 개의 Informix 가상 프로세서에 할당됩니다.

기본적으로 CPU VP는 하드웨어 CPU와 바인드되지 않습니다. Informix CPU VP는 다른 프로세스와 마찬가지로 하나의 하드웨어 CPU에서 다른 하드웨어 CPU로 이관할 수 있는 프로세스로 구현됩니다. SMP 플랫폼에서 실행되는 대부분의 운영 체제는 프로세스 이관을 효율적으로 관리할 수 있도록 최적화되어 있으므로 시스템의 CPU 자원에 대한 로드 밸런스를 적절하게 조정할 수 있습니다. 그러나 어떤 플랫폼에서는 한 CPU에서 다음 CPU로 프로세스를 이관하는 작업을 프로세서 친화도를 사용하여 무시할 수 있습니다. 프로세서 친화도가 지원될 경우, 프로세스를 특정 CPU에 바인드하는 데 사용할 수 있습니다. 친화도를 사용하여 처리된 프로세스가 CPU에 첨부되고 해당 CPU에는 우선 순위가 지정됩니다. PROCESSOR_AFFINITY 구성 매개변수를 사용하여 이러한 동작을 트리거할 수 있습니다. (플랫폼에 프로세서 친화도가 제공되지 않으면, 엔진에서 PROCESSOR_AFFINITY 구성 매개변수를 무시합니다.)

스레드 작동

Informix 스레드는 협력적 다중 작업 환경에서 작동됩니다. Informix 엔진은, 실행 프로세스를 미리 확보할 수 있는 UNIX 운영 체제와 달리 스레드를 양보해 줄 때까지 기다려야 합니다. 스레드를 엔진에 양보하면, 가상 프로세서가 다른 스레드를 로드하고 이 스레드를 처리하기 시작합니다. 이를 컨텍스트 전환이라고 합니다. 양보된 스레드가 없으면, 가상 프로세서가 이 스레드를 계속해서 처리해야 합니다. 대부분의 스레드는 처리 중 사전에 결정된 몇 개의 지점에서 그리고 공유 자원을 기다려야 하는 모든 시점에서 엔진에게 양보됩니다.

스레드 스케줄링은 공유 메모리의 가상 분할 영역에서 유지보수

되는 제어 구조를 통해 수행됩니다. 스레드 상태를 기록하기 위해 세션 제어 블록과 스레드 제어 블록이 유지보수됩니다. 이러한 제어 구조는 스레드가 휴면 상태가 되거나 다른 VP로 이관하기 위해 처리를 중단할 때 사용합니다.

Informix 스레드는 스택을 사용하여 전용 데이터를 저장하고 검색합니다. Informix 스레드가 가상 프로세서를 공유할 수도 있으므로 사용하는 데이터는 다른 스레드가 내용을 겹쳐 쓰지 않을 위치에서 유지보수되어야 합니다. 스택은 이러한 전용 영역의 역할을 합니다. VP가 처리를 새로운 스레드로 전환하면(즉, 컨텍스트 전환을 수행하면) 스택 포인터(스레드 스택의 주소)가 해당 스택의 스레드 제어 블록에 배치됩니다. 스레드가 다시 처리를 시작하면 스택 포인터를 검색하여 이 스택을 처리하기 시작합니다.

또한 큐 처리를 사용하여 스레드 처리를 제어할 수 있습니다. 컴퓨터 용어에서 큐는 프로그램이 추적해야 하는 항목의 목록입니다. IDS 엔진은 준비 큐, 휴면 큐, 대기 큐의 세 가지 큐를 사용합니다.

준비 큐(ready queue)는 현재 실행 준비가 되어 있는 스레드 목록입니다. VP가 컨텍스트 전환을 수행하면, 준비 큐에서 우선 순위가 적절하게 지정된 스레드 중 첫 번째 스레드를 검색하여 처리를 시작합니다.

휴면 큐(sleep queue)는 휴면 상태에 있는 스레드를 추적할 때 사용합니다. 스레드는 특정 기간 동안 휴면 상태에 있거나 영구적으로 휴면을 취할 수 있으며, 특정 기간 동안 휴면 상태에 있는 경우, 관리 VP가 지정된 시간에 해당 스레드를 웨이크업합니다. 그러나 Informix 스레드의 경우, 영구적이라는 것은 대개 자원의 가용성에 따라 결정되는 고정 기간입니다. 즉, 사용 중인 자원을 사용하려는 스레드는 해당 자원이 사용 가능해질 때까지 휴면 큐에서 휴면 상태로 있습니다.

대기 큐(ready queue)는 특정 이벤트가 발생하기를 기다리는 스레드를 추적하는 데 사용합니다. 이러한 큐는 공유 메모리에 있는 버퍼 페이지와 같은 공유 자원에 대한 액세스를 코디네이트할 때 사용합니다.

뮤텍스는 어셈블리 언어 테스트 및 설정 메커니즘을 사용하는 로크입니다. 뮤텍스는 공유 자원에 대한 스레드 액세스를 코디네이트할 때 사용합니다. 스레드가 공유 자원에 액세스하려고 할 때, 자원을 사용할 수 있으면 스레드에 대한 자원을 로크하라는 뮤텍스 요청을 실행합니다. 자원을 사용할 수 없으면 뮤텍스 호출은 요청 스레드에 현재 자원을 사용할 수 없다는 것을 알립니다.

Informix 데이터베이스 로깅

Informix 서버는 디스크 입출력 작업을 비동기적으로 수행합니다. 데이터가 바로 디스크에 쓰여지지 않으나 대신 공유 메모리 버퍼에 대해 디스크 입출력을 수행합니다. 공유 메모리 버퍼는 디스크로부터 읽혀지거나 디스크에 쓸 페이지(디스크 스토리지의 고정 크기 단위)를 보관하는 데 사용됩니다. 데이터베이스에서 데이터를 읽거나 쓰는 프로세스는 모두 우선적으로 이 공유 메모리 버퍼와 상호작용합니다. 이러한 상호 작용은 단순히 디스크 읽기나 쓰기 작업을 수행하는 프로세스에 투명성을 제공합니다.

Informix 데이터베이스의 데이터 무결성을 유지하기 위해서는 확실하게 트랜잭션의 일관성을 유지할 수 있는 기능이 필요합니다. 이러한 제어 레벨을 제공하기 위해 Informix 데이터베이스는 데이터베이스 로깅을 제공합니다. 이 로깅 작업은 Informix 아키텍처 디스크 관리 계층의 개념적 프레임워크에서 고려되는 사항입니다.

Informix 서버에는 논리 로그와 물리 로그의 두 가지 유형이 있습니다(그림 2-1 참조). 논리 로그는 데이터베이스에서 수행되는 트랜잭션을 저장하며 데이터베이스 업데이트 세트를 단일 트랜잭션으로 함께 그룹화하는 기능이 있습니다. 이 트랜잭션은 데이터베이스로 완료되거나 롤백됩니다. 트랜잭션이 완료되면, 트랜잭션의 데이터에 적용된 변경사항이 데이터베이스의 영구적인 부분이 됩니다. 트랜잭션이 롤백되면, 트랜잭션에서 수행된 모든 작업이 데이터베이스에서 제거됩니다. 완료 조치와 롤백 조치는 모두 데이터베이스를 일관성 있는 상태로 둡니다.

물리 로그

물리 로그는 업데이트되고 있는 프로세스에 있는 페이지의 사전 이미지(데이터베이스의 테이블 행이 들어 있는 공유 메모리 단위)를 저장합니다. 급속 복구로 알려진 프로세스를 통해 고장을 복구할 때, 또는 온라인 백업 프로세스 수행 중에 이러한 물리 로그가 사용됩니다. 물리 로그의 기본 역할은 공유 메모리 버퍼의 무결성을 유지하는 데 도움을 주는 것입니다. 이 물리 로그는 버퍼 페이지의 데이터가 업데이트되기 전에 페이지의 사전 이미지를 저장합니다. (일반적으로 물리 로그의 역할은 시스템에서 롤백 페이지를 필요로 할 때 롤백 페이지를 저장하는 것이므로 물리 로그라는 용어는 약간은 잘못된 명칭이라고도 할 수 있습니다.) 물리 로깅의 단계는 다음과 같습니다.

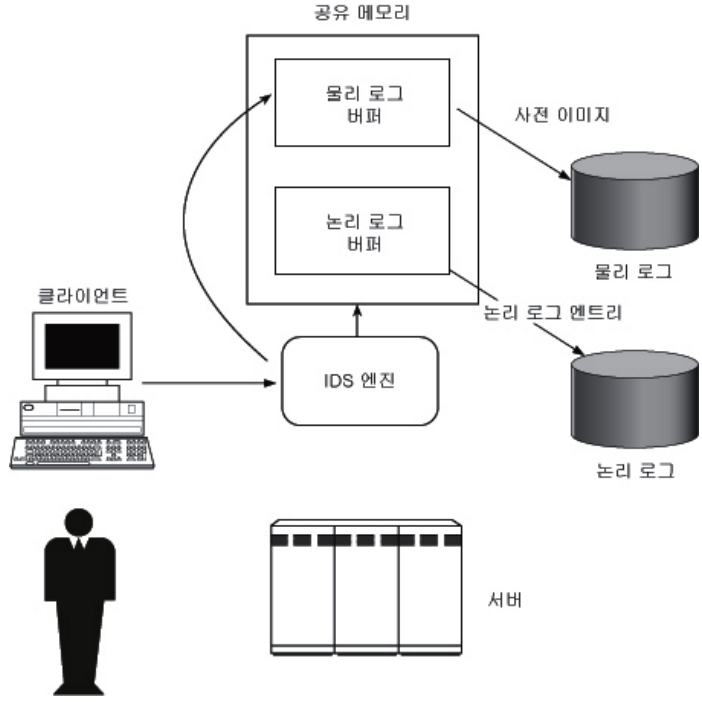


그림 2-1 논리 및 물리 로깅

1. 공유 메모리 버퍼에 페이지가 없으면, 디스크에서 공유 메모리 버퍼로 페이지를 읽습니다.
2. 변경되지 않은 페이지를 물리 로그 버퍼에 복사합니다.
3. 응용프로그램이 공유 메모리 버퍼에 있는 페이지를 수정합니다.
4. 물리 로그 버퍼를 디스크의 물리 로그로 플러시합니다.
5. 수정된 공유 메모리 버퍼 페이지가 디스크에 작성됩니다.
6. 체크포인트가 발생합니다. 공유 메모리 업데이트 페이지를 모두 디스크에 쓰고, 물리 로그 버퍼를 지우고 디스크의 물리 로그를 논리적으로 비웁니다. (포인터는 파일의 시작으로 재설정됩니다.)

데이터베이스를 업데이트하는 과정에서 페이지는 물리 로그에 쓰여지며 체크포인트가 발생할 경우에만 물리 로그에서 지워집니다. 체크포인트는 물리 로그를 데이터베이스와 동기화합니다. 체크포인트는 데이터베이스 무결성이 보장되는 지점입니다. 업데이트된 페이지는 모두 디스크에 쓰여지며 물리 로그는 지워집니다. (이 시점에

서 트랜잭션은 열려 있는 상태이지만 필요할 경우, 논리 로그의 정보를 사용하여 롤백할 수 있습니다.) 단계적으로 종료할 경우, 엔진에서 체크포인트를 수행하고 엔진이 오프라인 상태로 되기 바로 전에 물리 로그를 지웁니다.

단계적으로 종료될 기회가 없이 엔진이 오프라인 상태가 된 경우에는 급속 복구가 필요합니다. 이러한 경우, 엔진이 온라인 상태일 때 물리 로그를 읽고 물리 로그의 모든 페이지를 데이터베이스에 쓰므로, 동기화된 것으로 간주되지 않는 모든 업데이트는 제거됩니다(체크포인트에 의해 로그에서 이러한 업데이트는 모두 제거됨). 급속 복구가 수행되는 동안 논리 복구도 수행됩니다. 논리 복구는 체크포인트 이후에 완료되었고 엔진이 오프라인 상태가 되었을 때 완료되지 않은 트랜잭션을 롤백하는 데 사용됩니다.

논리 로그

논리 로그는 트랜잭션을 롤백할 때 즉, 트랜잭션에서 수행한 모든 업데이트를 제거하는 기능을 제공하는 데 사용합니다. 응용프로그램 프로그래머는 논리 로그를 사용하여 트랜잭션을 시작하고, 다수의 테이블에 일련의 업데이트를 수행한 후, 오류 메시지와 같은 일부 시스템 활동에 따라 트랜잭션을 롤백하여, 트랜잭션이 단일 명령문으로 실행했던 모든 업데이트를 효과적으로 취소할 수 있습니다.

논리 로그 레코드에는 트랜잭션을 롤백하는 데 필요한 모든 정보가 모두 들어 있습니다. 여기에는 로그 레코드 주소, 레코드 길이, 레코드 유형, 트랜잭션 번호, 논리 로그 번호 및 트랜잭션의 이전 레코드에 대한 링크 등의 정보가 있는 헤더가 포함됩니다. 로그 레코드 유형이란 로그 레코드가 기록하는 데이터베이스 조치를 말합니다. 유효한 데이터베이스 작업이란, 사용자가 시작한 활동과 데이터베이스가 하우스키핑 활동의 일부로 시작한 활동을 모두 포함하는, 사실상의 모든 데이터베이스 활동이 될 수 있습니다.

논리 로그는 디스크에 동기적으로 직접 쓰여지는 경우에 버퍼 미지정 로깅 옵션을 선택하지 않는 한, 먼저 메모리에 버퍼가 지정된 후 디스크에 쓰여지므로 트랜잭션이 완료되게 하여 잠재적으로 성능이 저하될 수 있습니다.

디스크에서 데이터를 업데이트하는 모든 데이터베이스 활동에 대해 논리 로그 레코드가 쓰여집니다. 로그 활동을 생성 시 가장 일반적인 사용자의 시작 동작은 SQL update문입니다. update문을 실행하면 영향을 받은 테이블과 행을 표시하는 레코드가 있는 로그 엔트리가 생성됩니다(영향을 받은 디스크의 페이지를 나타내는 내부 식

별자가 레코드에 들어 있는 경우에도). 영향을 받은 각 행에 레코드를 쓸 수 있으므로 update문이 로그 엔트리의 크기에 직접적인 영향을 미칩니다. 여러 행에 영향을 주는 update문이나 insert문에서 사용 가능한 논리 로그 부분을 많이 사용하는 것은 일반적인 일입니다. 로그 엔트리는 디스크에 있는 여러 개의 논리 로그 파일에 쓰여집니다. 정기적으로 논리 로그 파일을 테이프에 쓰고 비우지만, 개방 트랜잭션의 일부인 엔트리가 있는 로그(모든 행을 아직 업데이트하지 않은 광범위한 업데이트)는 테이프에 쓰고 비울 수 없습니다.

이러한 유형의 update문이 아주 많을 때 바람직하지 않은 긴 트랜잭션이 발생할 수 있습니다. 긴 트랜잭션은 거의 모든 사용 가능한 로그를 채운 다음 중단됩니다. 어떤 경우에는 사용 가능한 모든 로그를 채우면 트랜잭션을 시작할 수 있는 로그가 없기 때문에 시스템 활동이 중단되기도 합니다. LTXEHWM과 LTXHWM인 두 개의 시스템 구성 매개변수를 설정하여 하나의 긴 트랜잭션이 전체 인스턴스의 기능에 영향을 줄 가능성을 최소화할 수 있습니다. LTXEHWM과 LTXHWM 매개변수에 대한 자세한 사항은 3장을 참조하십시오.

체크포인트

앞서 언급한 대로, 체크포인트는 사전 정의된 체크포인트 간격(구성 매개변수로 설정)에서 발생하거나 또는 물리 로그 버퍼가 75% 찼을 때 발생합니다. 체크포인트가 발생했을 때, 물리 로그에 있는 모든 업데이트된 모든 페이지는 디스크에 쓰여졌음을 확인되어 물리 로그가 완전히 삭제됩니다.

체크포인트가 발생하면, 중요한 엔진 코드 섹션으로 지정된 것을 데이터베이스 세션에서 입력할 수 없으므로 대부분의 시스템 활동이 정지됩니다. 이는 시스템 무결성을 위해서는 필요하지만 체크포인트 활동의 경우에는 성능의 제한 조건이 됩니다. 과도한 체크포인트 발생이 시스템 성능을 저하시키는 원인이 되는 경우가 많습니다. 공유 메모리 버퍼에 업데이트된 페이지가 많으면, 이러한 페이지를 디스크에 쓰는 데 많은 시간이 걸릴 수 있습니다. 일반적으로 체크포인트는 몇 초가 소요되지만 문제가 있는 시스템의 경우는 1분 이상 소요될 수 있습니다.

청크 및 DBSPACE

청크는 디스크에 대한 IDS의 물리적 표현이고 DBSPACE는 디스크에 대한 논리적 표현으로 간주될 수 있습니다. 청크 세트가 DBSPACE에 지정되면 DBSPACE는 실제적인 청크 세트가 됩니다. 따라서 IDS 엔진은 물리 디스크를 일련의 청크로 인식합니다. 이러한 청크는 차례로 DBSPACE와 관련됩니다. 테이블을 DBSPACE에 작성할 수 있지만 테이블 분할화로 언급되는 정적 분할화를 사용하여 여러 DBSPACE에 걸쳐 쓸 수 있습니다.

청크의 크기는 일반적으로 2GB로 제한되므로 디스크가 2GB 이상 되는 설치에서는 2GB 조각이 IDS 청크에 매핑될 수 있도록 디스크를 파티션해야 합니다. (DBSPACE는 하나 이상의 청크 컬렉션이므로 크기가 2GB로 제한되지 않습니다.)

Informix-SQL은 개별적인 청크를 인식하지 못합니다. 모든 SQL 문은 디스크 스토리지를 지정하는 SQL DDL문의 DBSPACE와 상호 작용합니다.

데이터 스토리지 구조

페이지는 청크 내의 물리 스토리지 단위(공유 메모리 스토리지에 사용되는 페이지 포맷과 유사)입니다. 청크 내에는 DBSPACE 내의 페이지를 추적하는 다양한 구조가 있습니다. 즉, 청크에는 사용 가능 페이지 목록이 포함된 청크 사용 가능 목록 페이지가 있습니다. 이 목록을 사용하면 엔진에서 데이터를 DBSPACE에 추가할 때 사용할 수 있는 페이지를 쉽게 결정할 수 있습니다.

TABLESPACE에 있는 페이지가 인접하는 페이지일 필요는 없으며 일반적으로 페이지가 인접되어 있지 않습니다. 테이블은 익스텐트로 알려진 다중 페이지 세트로 대부분 구성되어 있습니다. IDS 엔진은 이러한 익스텐트를 인접하게 유지하려고 하지만, 익스텐트가 인접하지 않은 경우가 있습니다. 엔진은 테이블을 더 쉽게 스캔할 수 있도록 익스텐트를 인접하게 유지하는 것을 선호합니다.

IDS 엔진에는 TABLESPACE라는 논리 개념이 있습니다. TABLESPACE에는 테이블이나 테이블 조각 내의 모든 페이지가 들어 있습니다. TABLESPACE에는 비트맵 페이지(테이블 익스텐트 내의 페이지 사용을 추적하는 데 사용되는 페이지)와 인덱스에 할당된 모든 페이지도 들어 있습니다.

DBSPACE의 초기 청크에는 중복적으로 이름이 지정된 TABLESPACE TABLESPACE가 있습니다. TABLESPACE TABLESPACE에는 DBSPACE 내에 있는 TABLESPACE에 대한 정보가 들어 있습니다. 이 정보에는 DBSPACE 내에 상주하는 TABLESPACE의 구조 및 위치가 들어 있습니다.

Informix 자료형

Informix 관계형 데이터베이스는 관련된 테이블의 컬렉션에 데이터를 저장합니다. 이러한 테이블에는 작성되었을 때 자료형으로 지정되는 컬럼이 들어 있습니다. Informix 엔진은 기본 ANSI-SQL 자료형과 표 2-2에 표시된 몇 가지 추가 데이터베이스를 지원합니다.

표 2-2 유효한 SQL 자료형

자료형	길이	최소값	최대값
smallint	2	-32,787	+ 32,767
integer, int	4	-2,147,483,647	+ 2,147,483,647
smallfloat, real	4	플랫폼 종속	플랫폼 종속
float, double precision	8	플랫폼 종속	플랫폼 종속
decimal, dec, numeric	정밀도/2 + 1	-32 유효 숫자	+ 32 유효 숫자
money	정밀도/2 + 1	-32 유효 숫자	+ 32 유효 숫자
date	4	1/1/0001	12/31/1999
varchar	1 ~ 최대 255	1byte	255byte
char, character	프로그래머 정의	1byte	32,767byte

Smallint

Informix smallint는 2byte 정수로 C short 정수에 해당됩니다. 부호가 붙으며 +/-32,767의 최소/최대값을 지원합니다. Informix에서 정수 오버플로(최대값보다 더 큰 수 저장 시도)는 오류입니다.

Integer

Informix integers는 C long 정수에 해당되는 4byte 정수입니다. integer에도 부호가 붙으며 +/-2,147,483,647의 최소/최대값을 지원합니다. 정수 오퍼플로 오류는 smallint 자료형의 경우와 마찬가지로 트랩됩니다.

그러나 최대 정수값 테스트를 위해 제공되는 매크로 상수가 없으므로 프로그래머가 이러한 값을 변수에 저장하고 적절하게 확인해야 합니다.

Smallfloat

Informix smallfloat는 대략 C float 또는 real과 같습니다. smallfloat는 공식적으로 머신 종속적인 최소/최대값이 있지만 일반적으로 대략 +/-1016입니다.

Decimal

초기 UNIX 머신의 부동 소수점 수와 연관된 사용 변형 및 어려움으로 인해 Informix는 고유 형식의 부동 소수점 수 표현인 decimal 자료형을 개발했습니다. 숫자 데이터를 모든 머신에서 일관된 포맷으로 저장할 수 있으므로 이러한 설계를 통해 네트워크로 데이터를 이동하는 것이 훨씬 더 수월해진다는 장점이 하나 더 추가되었습니다.

Informix decimal 자료형은 표 2-3에 표시된 대로 내부적으로 저장됩니다.

표 2-3 decimal 자료형 스토리지

구성 요소	스토리지
지수	2byte 정수
소수점 위치	2byte 정수
자리 수	2byte 정수
숫자	길이가 선언된, 정밀도/2인 문자

decimal 자료형은 표현에서 고정 소수점일 수도 있고(선언 프로세스 시 정밀도가 지정된 경우) 부동 소수점이 될 수도 있습니다(정밀도가 지정되지 않은 경우). 정밀도는 대략 +/-1032 또는 32 유효 숫자로 매우 큽니다. decimal 자료형 컬럼은 다음과 같이 선언됩니다.

```
create table sales_rec ( sales_num integer,
sales_amount decimal );
```

이 선언문에서는 정밀도를 지정하지 않기 때문에 변수는 기본적으로 16자리의 정밀도와 소수점 부동수를 갖습니다. 그러나 컬럼이 다음과 같이 선언되면,

```
create table sales_rec (sales_num integer,
sales_amount decimal (10) );
```

최대 유효 숫자가 10인 decimal 변수가 생성되거나 또는 컬럼이 다음과 같이 정의될 수 있습니다.

```
create table sales_rec (sales_num integer,
sales_amount decimal (10,2) );
```

그러면 정밀도가 2(소수점 오른쪽 2자리)인 10자리의 decimal 변수가 작성됩니다.

m이 정밀도이고 n이 스케일이라고 설정할 경우, 컬럼은 다음과 같이 선언됩니다.

```
...

decimal_column decimal (m,n)
```

이 컬럼에 저장될 수 있는 최대값은 $10^{m-n} - 10^{-n}$입니다. 이는 decimal이 다음과 같이 선언된다는 의미입니다.

```
...

decimal_column decimal (2,0)
```

그러면 이 컬럼은 최대 $10^{2-0} - 10^{-0}$ 또는 99(100-1)를 저장할

수 있습니다. 이보다 더 큰 수를 저장하려는 경우 표현식 오류가 발생합니다.

Money

money 자료형은 decimal 자료형의 특수한 경우입니다. money 자료형은 기본적으로 스케일이 2인 고정 십진수입니다. money 자료형이 표시될 때 표시되는 숫자의 왼쪽에는 기본 통화 기호가 표시됩니다. 기본 출력 통화 기호는 DBMONEY 환경 변수를 사용하여 변경할 수 있으며 포맷은 다음과 같습니다.

```
DBMONEY=scb
```

이 포맷에서 s는 통화값 앞에 오는 기호이고, c는 전체 통화 부분과 소수 부분을 구분하는 기호이며, b는 통화값 뒤에 오는 기호입니다. c의 유효값은 "." 또는 ","입니다. s나 b의 유효값은 쉼표나 마침표가 포함되지 않은 일련의 문자입니다.

Float

Informix float 자료형은 머신 플랫폼의 C double과 동일한 정밀도를 가집니다. 정밀도는 머신 종속적이지만 대개 10^{-32}로 제한됩니다. 언어 정의 double precision은 float의 동의어입니다.

Date

Informix date는 1900년 1월 1일을 포함해서 이 때부터의 일 수를 나타내는 4byte 정수로 내부적으로 표시됩니다. Informix 엔진은 기본 포맷 또는 DBDATE 환경 변수가 지정한 포맷에 따라 이 값을 자동으로 포맷합니다. DBDATE 환경 변수는 다음과 같이 설정됩니다.

```
DBDATE=dmync
```

이 명령문에서 d는 설정되고 있는 일, m은 설정되고 있는 월, y는 연도, n은 연도 위치 다음에 나오는 숫자로 인쇄할 연도의 자릿수를

나타냅니다. 연도의 자릿수는 2 또는 4가 될 수 있습니다. 마지막 문자인 c는 날짜의 구성 요소를 구분하는 데 사용되는 문자입니다.

Datetime

datetime 자료형은 시간의 순간을 나타내는 방법으로 날짜와 시간 구성 요소가 모두 포함됩니다. datetime 자료형의 정밀도에는 가변의 정밀도 등급이 있습니다. 사용자는 datetime 변수가 특정한 정밀도가 되도록 정의한 후 필요에 따라 정밀도를 동적으로 변경할 수 있습니다. datetime 자료형은 년, 월, 일, 시간, 분, 초 및 초의 소수 자리로 구성됩니다. 이러한 자료형은 타임스탬프를 레코드와 함께 저장해야 하는 응용프로그램에서 아주 유용합니다. datetime 자료형은 다음과 같이 선언됩니다.

```
...

variable_name datetime first to last
```

이 예에서 first와 last 식별자는 표 2-4의 필드 중 하나이고, 여기서 last 엔트리 레벨은 동일한 datetime 정의에 있는 목록의 first 엔트리보다 앞에 올 수 없습니다.

표 2-4 Datetime 식별자

필드	유효한 엔트리	레벨
년	1-9999	1
월	1-12	2
일	1-31(월에 따라 다름)	3
시간	0-23	4
분	0-59	5
초	0-59	6
초의 소수 자리	소수 다섯 자리 수	7

Interval

interval 자료형은 시간의 간격을 나타내는 데 사용됩니다. interval 자료형은 두 datetime 값 사이의 차이를 나타냅니다. 또한 가변의 밀도 등급을 지원하는데, 이 밀도의 등급은 런타임 시 변수를 정의하거나 변경할 때 정의할 수 있습니다.

특별히 interval 자료형은 날짜와는 무관한 시간 간격을 표시하도록 설계되어 있으므로 날짜와 시간을 혼용할 수 없는 특정 구성 요소로 표시되어야 합니다.

표 2-5 년/월 interval

필드	유효한 엔트리	레벨
년	1-9999(기본값) 1-999,999,999(최대값)	1
월	1-99(기본값) 1-999,9999,9999(최대값)	2

표 2-6 일/시간 interval

필드	유효한 엔트리	레벨
일	1-99(기본값) 1-999,999,999(최대값)	1
시간	1-99(기본값) 1-999,999,999(최대값)	2
분	1-99(기본값) 1-999,999,999(최대값)	3
초	1-99(기본값) 1-999,999,999(최대값)	4
초의 소수 자리	1-999(기본값) 1-99,999(최대값)	5

interval 자료형에는 년/월의 시간 간격을 나타내는 유형과 일/시간/분/초를 나타내는 유형 두 가지가 있습니다. interval은 다음 포맷으로 선언됩니다.

```
...

interval_col interval first to last
```

이 예에서, first 또는 last는 표 2-5와 표 2-6에 나오는 필드 중 하나로, 여기서 last 엔트리 레벨은 동일한 interval 정의에 있는 목록의 first 엔트리보다 앞에 올 수 없습니다.

기본 엔트리는 interval 컬럼을 선언할 때 무시할 수 있는 시스템 기본값입니다. 다음 선언문을 통해 이를 확인할 수 있습니다.

```
...

interval_col interval year(5) to month ...
```

이 예에서, 연도에 네 자리의 기본값이 아닌 다섯 자리의 숫자가 올 수 있도록 interval 컬럼이 정의되었습니다.

Character

character 자료형은 길이가 최대 32,767자인 고정 길이 문자열입니다. character 자료형은 항상 정의된 길이에 공백이 채워집니다. 대개 NULL로 종료되지만, 길이를 결정할 때 NULL 터미네이터에 의존하지 않습니다. 문자 변수를 정의할 때 길이가 고정됩니다.

Varchar

varchar 자료형은 길이가 최대 255자인 가변 길이 문자열입니다. 컬럼의 최소 크기 및 최대 크기는 컬럼을 선언할 때 정의됩니다. varchar의 사용 방법은 character 자료형의 경우와 같습니다. 차이점은 varchar 자료형의 경우 정의된 길이에 공백이 채워지지 않으며, 컬럼에 위치한 문자 데이터를 저장할 때 필요한 스토리지만 사용한다는 점입니다.

특수 자료형

엔진에서 특별히 고려해야 하는 몇 가지의 자료형이 있습니다. 일반적으로 엔진에서 지원하는 표준 자료형은 1byte 또는 2byte에서 32KB 사이의 스토리지가 필요한 작은 데이터 객체입니다. 그러나 모든 스토리지가 표준 자료형 틀에 맞지 않아도 됩니다. 관계형 데이터베이스에 rich 자료형을 저장해야 할 필요성이 점차 높아지고 있

습니다. 오디오와 비디오 클립 및 그래픽 이미지와 같은 데이터는 다른 데이터로 저장해야 합니다. 이러한 필요성으로 인해 BLOB(binary large object)이 나오게 되었습니다.

BLOB은 데이터의 이진 스트림을 저장하는 자료형입니다. Informix 엔진은 BLOB 컬럼에 저장된 데이터의 자세한 정보에 대해서는 전혀 알지 못하며 다만 객체의 크기만 인식합니다. 데이터베이스 엔진에서는 BLOB 컬럼의 검색 및 업데이트에 약간의 제한은 있지만 데이터를 표준 SQL문으로 처리할 수 있습니다.

BLOB 데이터 컬럼의 크기가 몇 메가바이트가 되는 것은 보통입니다. 하지만 이와 같은 대형 객체를 저장하고 처리하려면 문제가 발생합니다. Informix에는 BLOB 데이터를 물리적으로 저장할 때 테이블과 동일한 물리 공간에 저장하거나 고유의 DBSPACE에 저장하는, 두 가지 방법을 사용합니다. 테이블에 저장된 BLOB은 TABLESPACE BLOB이라고 하고, 고유 DBSPACE에 저장된 BLOB은 BLOBSPACE라고 알려져 있는 장소에 저장되며 BLOBSPACE BLOB이라고 합니다.

BLOBSPACE는 특정 크기의 BLOB을 보관할 수 있도록 지정된 DBSPACE의 특수한 경우입니다. BLOBSPACE에 저장된 BLOB은 IDS 공유 메모리 버퍼를 통과하지 않고, 직접 BLOBSPACE(디스크)에 쓰여집니다. BLOBSPACE의 구조는 이러한 프로세스가 좀더 효율적으로 진행될 수 있도록 지정됩니다.

BLOBSPACE가 작성되면 BLOB 페이지라고 알려진 BLOBSPACE의 페이지 크기가 지정됩니다. BLOB 페이지 크기는 IDS 페이지 크기의 배수입니다. BLOB 페이지 크기는 BLOBSPACE에 따라 다양하며 저장할 BLOB의 크기는 평균적으로 BLOB 페이지 크기와 비슷하거나 약간 더 작을 때가 가장 좋습니다.

컬럼에 BLOBSPACE BLOB이 포함된 행에는 BLOB에 대한 실제 데이터가 들어 있지 않습니다. 대신, 56byte BLOB 설명자의 형식으로 BLOB 데이터에 대한 포인터가 들어 있습니다. 이러한 BLOB 설명자는 BLOB의 첫 번째 세그먼트의 위치에 대한 정방향 포인터입니다. 이 세그먼트는 DBSPACE에 있는 BLOB 페이지, 임의의 BLOB 페이지, 또는 광학 플래터(온라인/광학 옵션을 사용하는 경우)일 수 있습니다.

미러링

Informix 서버에는 시스템에 작성된 각 DBSPACE를 미러링하는 기능이 있습니다. 미러링 기능을 사용하면 디스크의 논리 분할 영역

(이 경우 DBSPACE)을 계속적으로 백업합니다. 미러링은 Informix 서버 소프트웨어를 통해 제공되므로 하드웨어 미러링과는 구별됩니다.

Informix 엔진에 미러링이 설정되면 기본 DBSPACE와 보조 또는 미러 DBSPACE가 있습니다. 기본 및 보조(미러) DBSPACE 모두에 대해 항상 디스크 쓰기를 수행하지만, 모든 디스크 입출력은 기본 DBSPACE에 지정됩니다.

일반적으로 기본 청크에 대해 디스크 읽기를 실행하지만, 기본 청크에서 읽는 중에 입출력 대기가 있는 경우 엔진은 2차 청크에서 읽습니다. 기본 DBSPACE에서 읽지 못하면 DBSPACE가 오프라인으로 표시되고 보조 또는 미러 DBSPACE에서 읽기를 시도합니다. 이러한 모든 작업이 사용자에게 명백하게 실행되며, 엔진은 온라인 상태로 남아 있고 쿼리는 성공적으로 완료됩니다.

데이터 복제

Informix 엔진에서의 데이터 복제란 두 개 이상의 사이트에 데이터베이스를 저장하는 프로세스입니다. Informix 복제 기능을 사용하여 기본 서버인 데이터베이스 전체를 한 사이트에서 보조 서버인 다른 사이트로 복제할 수 있습니다. 기본 서버에서는 전체 읽기/쓰기 액세스가 허용되지만 보조 서버에서는 읽기 전용 액세스만 허용됩니다.

Informix 복제는 오버헤드 및 유지보수가 거의 없이 데이터를 복제할 수 있는 고가용성 기능을 제공하도록 설계되었습니다. 복제 기능은 추가 기능이 아니라 엔진에 통합된 부분입니다.

데이터 복제는 저장 및 포워드 접근방식을 사용하여 구현됩니다. 복제할 데이터베이스 레코드는 기본 서버에 저장된 후 보조 서버로 전달됩니다. 즉, 레코드가 복제 데이터베이스에 대해 이 보조 서버에서 롤포워드된 것입니다. 이러한 방법으로 보조 서버는 기본 서버와 동시성을 유지합니다. 로그 엔트리는 기본 서버의 공유 메모리 복제 버퍼에 저장된 후 보조 서버의 수신 버퍼로 전달됩니다. 보조 서버의 수신 버퍼로 전달된 로그 엔트리는 보조 서버의 데이터베이스 사본으로 롤포워드됩니다.

데이터 복제는 동기적으로도 비동기적으로도 수행할 수 있습니다. 동기적 수행에서는, 보조 서버가 트랜잭션 롤포워드를 종료해야 기본 서버의 트랜잭션이 완료됩니다. 이러한 복제 모드에서는 기본 서버와 보조 서버를 모두 동기화해야 합니다(동일한 시점에서 모두 동시성 유지). 이러한 접근 방식의 단점은 기본 서버의 트랜잭션을

완료하기 전에 보조 서버의 각 트랜잭션이 완료되어야 하기 때문에 기본 서버의 성능이 저하된다는 점입니다. 네트워크 연결 상태에서 복제가 실행되므로 모든 트랜잭션 데이터를 기본 서버에서 보조 서버로 전달해야 하는데, 이는 고속 네트워크 연결 시에도 시간이 많이 소요되는 프로세스가 될 수 있습니다.

비동기 복제의 경우, 기본 서버의 로그 엔트리는 논리 로그 버퍼에 쓰여진 다음에 복제 버퍼로 복사됩니다. 그런 후 기본 서버의 복제 버퍼는 기본 서버에서 보조 서버로 정기적으로 전송됩니다. 보조 서버는 수신 버퍼에서 논리 로그 엔트리를 수신한 후에 엔트리를 롤포워드합니다. 기본 서버는 보조 서버가 로그 롤포워드를 완료할 때까지 기다리지 않고, 트랜잭션을 완료한 후 계속해서 처리합니다.

tip *비동기 복제의 경우, 시스템이 고장난 경우 로그 엔트리가 유실될 수 있습니다. 비동기 복제를 사용하고 시스템 고장 이후에 복제를 재시작하면 일치될 수 없는 로그 엔트리는 유실되었으나 알려짐(lost-and-found) 위치에 쓰여지며, 보조 서버가 아닌 기본 서버에서 완료되었을 수 있는 모든 트랜잭션은 기본 서버로 롤백됩니다.*

트랜잭션 상실의 위험을 줄이기 위해서는 기본 서버와 보조 서버 모두에서 버퍼 미지정 로깅 옵션을 사용해야 합니다. 이렇게 하면 기본 서버의 트랜잭션의 완료 작업의 시작과 보조 서버에서 로그 레코드를 수신한 시간 사이에 소요되는 시간을 줄일 수 있습니다.

시스템 예약 페이지

IDS 엔진은 시스템에서 사용할 수 있도록 공유 메모리의 여러 페이지를 예약합니다. 이러한 페이지에는 구성 매개변수, 청크 및 DBSPACE 목록, 백업 및 복구 정보와 같은 시스템에 관한 중요한 정보가 들어 있습니다.

IDS 공유 메모리에는 12개의 예약 페이지가 있습니다. 시스템 예약 페이지에 보관된 정보가 갖는 중요성과 시스템 체크포인트의 요구사항으로 인해 Informix 서버는 다수의 예약 페이지의 사본을 두 개씩 보관합니다. 따라서 사본 하나가 손상된 경우 다른 사본에서 정보를 제공할 수 있습니다. 다음 표 2-7에 IDS 예약 페이지가 공유 메모리에 표시되는 순서대로 나열되어 있습니다.

표 2-7 IDS 예약 페이지

시스템 예약 페이지	목적
1 PAGE_PZERO	시스템 식별
2 PAGE_CONFIG	구성 파일 복사
3 PAGE_ICKPT	체크포인트/논리 로그 추적
4 PAGE_2CKPT	대체 체크포인트 페이지
5 PAGE_1DSP	DBSPACE 설명
6 PAGE_2DSP	대체 DBSPACE 페이지
7 PAGE_1PCHUNK	미러된 청크 설명
8 PAGE_2PCHUNK	대체 기본 청크 페이지
9 PAGE_1MCHUNK	미러된 청크 설명
10 PAGE_2MCHUNK	대체 미러 청크 페이지
11 PAGE_1ARCH	DBSPACE 백업 추적
12 PAGE_2ARCH	대체 보관 페이지

자세한 예약 페이지에 대해서는 다음 절을 참조하십시오.

PAGE_PZERO

이 페이지에는 데이터베이스 시스템 상태, 시스템 플래그, 페이지 크기, 작성된 날짜/시간, 버전 번호 및 내부 정보 등의 IDS 정보가 들어 있습니다.

PAGE_CONFIG

시스템 구성 예약 페이지이며, IDS 인스턴스에 대한 구성 매개변수가 들어 있습니다. 실제로 `onconfig` 파일의 내용입니다.

PAGE_CKPT

이 페이지에는 시스템 논리 로그 파일과 체크포인트를 추적하는 정보가 들어 있습니다. 체크포인트 시간, 물리 로그 크기와 시작 주

소, 논리 로그 식별자, 로그 위치, DBSPACE 설명자 페이지 주소 및 미러된 청크 설명자 페이지를 추적합니다.

PAGE_DSP

이 페이지는 DBSPACE 페이지로 각 DBSPACE와 각 DBSPACE의 현재 상태, DBSPACE의 첫 번째 청크 번호, DBSPACE가 작성된 시간, DBSPACE의 이름 및 DBSPACE의 소유자에 관한 정보를 저장할 때 사용합니다.

PAGE_PCHUNK

이 페이지는 기본 청크 예약 페이지로 인스턴스의 각 청크를 설명할 때 사용합니다. 예약 페이지는 청크, 청크에 대한 경로 이름, 청크가 속한 DBSPACE 또는 BLOBSPACE, 청크의 상태를 설명합니다. 청크 설명 정보에는 청크 크기, 청크 오프셋, 사용 가능한 페이지 수 및 기타 관련 정보가 표시됩니다.

PAGE_MCHUNK

PAGE_MCHUNK 예약 페이지는 인스턴스의 각 미러 청크를 추적할 때 사용합니다. 기본 청크 번호, DBSPACE의 다음 청크 번호, 기타 청크와 청크의 DBSPACE에 관한 정보가 들어 있습니다.

PAGE_ARCH

PAGE_ARCH 시스템 예약 페이지는 데이터베이스 인스턴스에서 수행된 가장 최근 보관 정보 및 그 다음의 최근 보관 정보를 설명할 때 사용합니다. 이 페이지에는 백업을 기록하는 논리 로그 정보와 데이터 복제 정보도 들어 있습니다.

root DBSPACE

root DBSPACE는 데이터베이스 인스턴스의 기본 DBSPACE입니다. root DBSPACE에는 데이터베이스 인스턴스와 관련된 중요한 정보를 추적하는 시스템 예약 페이지가 있습니다. 또한 기본 DBSPACE도 들어 있으므로, create table이나 create database문과 같은 SQL문에 특정한 DBSPACE를 언급하지 않으면 기본 root DBSPACE가 사용됩니다.

시스템 카탈로그

시스템 카탈로그는 데이터베이스로의 입력점을 나타냅니다. 데이터베이스에서 쿼리의 구문을 분석하면 쿼리에서 참조된 데이터베이스 테이블의 위치를 판별하기 위해 시스템 카탈로그를 읽습니다. 또한 옵티마이저에는 쿼리의 테이블에 있는 인덱스와 테이블의 인덱스 및 데이터에 관한 통계 정보가 필요합니다. 이러한 모든 정보가 시스템 카탈로그에 저장됩니다.

어떤 이유로 인해 시스템 카탈로그를 읽을 수 없으면 데이터베이스에도 액세스할 수 없습니다. 이 같은 이유로 root DBSPACE를 미러링해야 합니다. 기본 root DBSPACE가 실패할 경우 미러 root DBSPACE가 명백하게 대신하여 사용될 수 있습니다.

표 2-8에 나열된 테이블은 시스템 카탈로그에 있는 정보입니다.

표 2-8 시스템 카탈로그 테이블

테이블 이름	목적
systables	모든 테이블 목록 포함
syscolumns	모든 컬럼 목록 포함
sysindexes	모든 테이블에 대한 모든 인덱스 목록 포함
systabauth	테이블별 사용 권한
syscolauth	테이블 컬럼별 사용 권한
sysviews	작성된 SQL 뷰

<계속>

표 2-8 시스템 카탈로그 테이블

테이블 이름	목적
sysusers	데이터베이스에 추가된 사용자
sysdepend	각 뷰 또는 테이블이 다른 뷰나 테이블에 어떻게 종속되는지 설명
syssynonyms	각 테이블 또는 뷰에 대한 동의어 목록 포함
syssyntable	각 동의어와 동의어가 나타내는 객체 간 매핑에 대한 요약
sysconstraints	데이터베이스의 적절한 컬럼 제한 조건
sysreferences	데이터베이스의 컬럼에 관한 참조 제한 조건
syschecks	컬럼에 정의된 점검 제한 조건
sysdefaults	컬럼에 대한 사용자 정의 기본값
syscoldepend	점검 제한 조건이 있는 것으로 지정된 컬럼의 목록 포함
sysprocedures	시스템에 작성된 내장 프로시저
sysprocbody	내장 프로시저의 본문(컴파일된 p 코드)
sysprocplan	내장 프로시저의 데이터 처리 명령문에 대한 쿼리 계획
sysprocauth	내장 프로시저에 부여된 권한
sysblobs	BLOB 컬럼의 스토리지 위치
sysopclstr	데이터베이스의 광학 클러스터 설명
systriggers	현재 사용되고 있는 SQL 트리거
systrigbody	현재 사용되고 있는 SQL 트리거의 본문
sysdistrib	옵티마이저가 사용하는 데이터 분산 정보 저장
sysfragments	테이블과 인덱스의 분할화 정보 저장
sysobjstate	특정 데이터베이스 객체(인덱스, 트리거, 제한 조건) 상태에 대한 정보 저장
sysviolations	기본 테이블에 대한 위반 및 진단 테이블
sysfragauth	테이블 조각에 부여된 권한
sysroleauth	시스템에 부여된 사용자 역할

sysmaster 데이터베이스

IDS 서버 인스턴스는 인스턴스 내에 데이터베이스에 관한 정보를 유지보수합니다. 이러한 정보는 각 데이터베이스의 상태, 활성 세션,

성능 관련 정보 및 보관과 복구 통계와 관련됩니다. 이러한 정보는 sysmaster 데이터베이스에서 사용할 수 있습니다. 어떤 정보는 데이터베이스 내의 종래의 데이터베이스 테이블에 보관되며, 어떤 정보는 이 공유 메모리 테이블에 보관된 후 IDS 엔진에 의해 간단하게 SQL 테이블로 매핑됩니다. 표 2-9에는 sysmaster 데이터베이스에 포함된 테이블이 나열되어 있습니다.

표 2-9 sysmaster 테이블

테이블 이름	정보
sysadtinfo	구성 정보 검사
sysaudit	이벤트 마스크 검사
syschkio	청크 입출력 통계
syschunks	청크 정보
sysconfig	구성 정보 페이지
sysdatabases	데이터베이스 정보
sysdbslocale	로케일 정보
sysdbspaces	DBSPACE 정보
sysdri	데이터 복제 정보
sysextents	익스텐트 할당 정보
syslocks	활성 로크 정보
syslogs	논리 로그 파일 정보
sysprofile	시스템 프로파일 정보
sysptprof	테이블 정보
syssesprof	다양한 사용자 활동 계수
syssessions	연결된 각 사용자에 대한 설명
sysseswts	여러 객체 각각의 사용자 대기 시간
systabnames	데이터베이스, 소유자 및 TABLESPACE의 테이블 이름
sysvpprof	사용자 및 각 가상 프로세서에서 사용하는 시스템 CPU

데이터베이스 관리자는 sysmaster 데이터베이스의 테이블을 변경해서는 안 됩니다. IDS 엔진은 이 정보에 따라 달라지며, 변경할 경우 예측하지 못한 결과가 발생할 수 있습니다.

Informix 서버 아키텍처

앞에서 설명한 대로, Informix 엔진에 필요한 처리를 수행하기 위해 계속적으로 상호 작용하는 네 가지 주요 아키텍처 구성 요소가 있습니다. 이러한 구성 요소는 엔진의 자원을 관리합니다. 구성 요소는 다음과 같습니다.

1. 디스크 입출력
2. 메모리(버퍼)
3. 스레드(프로세스)
4. 통신

서버의 모든 활동은 이러한 구성 요소 가운데 하나의 제어를 받습니다. 데이터를 요청하는 쿼리에는 여러 가지 아키텍처 엔티티의 작업이 필요합니다. 통신 구성 요소는 메모리 연결을 관리하고, 메모리 구성 요소는 메모리를 할당하여 데이터 요청을 처리해야 합니다. 디스크 입출력 구성 요소는 디스크의 데이터를 검색해야 하고, 프로세스 관리 구성 요소는 CPU 자원(스레드 형식)을 할당하여 작업을 수행하도록 해야 합니다.

Informix 서버 아키텍처의 중요한 장점은 작업을 병렬로 수행할 수 있는 기능입니다. 엔진의 하우스키핑 기능이 병렬로 수행될 뿐만 아니라 개별 쿼리도 병렬 작업 즉, 개별 실행 스레드로 분류될 수 있습니다. Informix 엔진 아키텍처(그림 2-2 참조)는 이러한 병렬 작업을 분산 제어하여 수행합니다. 위의 아키텍처 구성 요소 중 어떤 구성 요소도 쿼리의 전체 작업을 제어하지는 않습니다. 이러한 구성 요소는 서로 통신하고 요청도 하지만, 어떤 구성 요소도 교통 경찰과 같은 역할을 하지 않으며, 다른 구성 요소의 작업을 모니터하고 제한하지는 않습니다. 하지만 상대적으로 모든 구성 요소는 동일합니다.

처리에 대한 이러한 접근 방법으로 과도한 처리가 진행되는 동안 발생할 수도 있는 경쟁을 줄이고 확장성이 향상시킬 수 있습니다. 또한 XPS 서버 옵션에 사용되는 아무 것도 공유하지 않는 엔진 디자인 접근 방식도 무리 없이 사용할 수 있게 됩니다. 여기에 Informix 아키텍처에 상당한 투자를 한 결과, 대량 병렬 프로세서(MPP) 환경에서 거의 일직선으로 확장이 가능합니다.

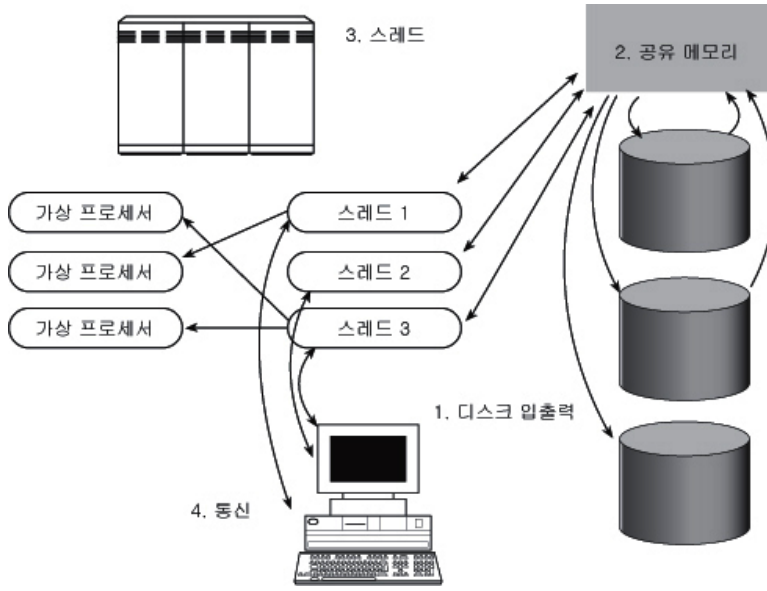


그림 2-2 Informix 엔진 아키텍처

이러한 아키텍처 구성 요소 각각에 대한 자세한 사항은 다음 절을 참조하십시오.

IDS 아키텍처: 디스크 입출력

아키텍처의 디스크 관리 구성 요소가 디스크와의 모든 상호 작용을 제어합니다. 앞에서 언급한 대로 Informix 서버의 경우 디스크는 실제로 두 개의 계층에서 볼 수 있습니다. 이 두 개의 계층이란 물리 디스크(UNIX에서는 장치 파일)에 관련된 청크 컬렉션과 관계있는 물리 계층과, 청크 컬렉션을 DBSPACE에 매핑하는 논리 디스크입니다. 데이터베이스와 데이터베이스 테이블은 차례로 DBSPACE 또는 DBSPACE 세트에 매핑됩니다.

IDS 병렬 처리 및 분할화

다섯 개의 DBSPACE에 걸쳐 테이블이 분할되면 테이블에는 다섯 개의 개별 조각이 포함됩니다. 처리 중인 쿼리에서 전체 테이블이 스캔되어야 할 경우(테이블의 모든 행을 읽어야 함), Informix 서버가 다섯 개의 개별 스레드(다섯 개의 내부 프로세스)를 시작하는 것으로 간주하여 DBSPACE에 있는 테이블의 각 조각을 스캔합니다.
Informix 엔진에는 다중 작업이나 스레드를 병렬로 수행하는 기능이 있습니다. 하나의 작업을 수행할 때 해당 작업의 완료와 관계없는 다른 작업도 수행할 수 있습니다.

데이터를 분할하면, Informix 엔진의 병렬 처리가 트리거되므로 Informix 엔진의 병렬 처리는 확고한 분할화 스키마에 좌우된다는 점에 유의하십시오.

이러한 아키텍처 구성 요소의 일부가 되는 추가 개념이 Informix 서버와 Informix 서버의 멀티스레드 엔진의 중심 개념인 조각 개념입니다. 분할화는 테이블 행을 다중 DBSPACE에 매핑하는 것입니다. 특별히 수평 분할화로 알려진 Informix 엔진에는 테이블의 특정 행을 특정 DBSPACE에 배치시키는 기능이 있습니다. 행을 무작위로 배치할 수도 있고 선택적으로 테이블 컬럼 값과 관련된 표현식에 기초해 배치시킬 수도 있습니다. Informix 서버가 작업에 할당할 CPU 자원량(처리 스레드 수)을 결정할 때, 평가하는 기준의 하나가 검사되고 있는 테이블의 분할화입니다.

IDS 아키텍처: 데이터베이스 메모리

Informix 엔진은 여러 가지 이유로(성능상의 이유는 아님), 엔진에서 필요로 하는 사실상 모든 메모리 관리 작업을 수행합니다. 서버에서 작업에 대한 메모리를 할당하면 서버는 해당 메모리가 더 이상 필요 없을 때 이를 사용할 수 있게 해야 합니다. 메모리 관리 계층은 메모리 부여 관리자(MGM)로 알려져 있습니다.

IDS는 여러 사용자가 공유하는 전역 프로그램 메모리의 공유 부분인 공유 메모리 버퍼를 작성하므로, 엔진에서 처리해야 하는 메모리 관리 작업이 더욱 복잡해집니다. 엔진에서 수행해야 하는 다양한 작업에 대해 메모리를 할당하고 릴리스해야 할 뿐만 아니라 전역 공유 메모리에 액세스하는 다양한 사용자 간의 동시성도 관리해야 합니다. 이를 위해서는 메모리를 효과적으로 제어할 수 있는 복잡한

로크 메커니즘이 필요합니다. 이는 래치(IDS에서 뮤텍스로 구현됨)로 알려진 "테스트 및 설정" 로크 메커니즘을 사용하여 관리합니다.

이전의 Informix 서버 버전에서 LOW 레벨의 메모리 할당은 운영체제 호출에서 전역 힙의 메모리를 할당하고 검색했습니다. Informix Dynamic Server에서부터 엔진에서 LOW 레벨의 메모리 할당을 처리하기 시작했고, 그 결과 엔진에 의한 메모리 할당 제어가 세분화될 수 있었으며, 이를 통해 엔진에서 이 작업을 엄격하게 제어하고 우선 순위를 지정할 수 있게 되었습니다.

Informix 엔진은 공유 메모리를 상주, 가상 및 통신과 같은 서로 다른 분할 영역으로 나눕니다. 상주 분할 영역에는 공유 메모리 버퍼와 기타 다양한 테이블 및 큐가 포함됩니다. 가상 분할 영역에는 엔진이 수행하는 처리와 해당 처리를 지원하는 다양한 버퍼 및 풀에 대한 제어 정보가 포함됩니다. 엔진에서 공유 메모리 프로세스 간 통신을 사용하는 경우, 통신 계층에는 엔진과 엔진의 자원을 사용하는 프로세스 간의 통신을 관리하기 위한 제어 구조와 버퍼가 포함됩니다. 이러한 공유 메모리 구성 요소는 다음 절에 자세하게 설명되어 있습니다.

공유 메모리의 상주 분할 영역

공유 메모리의 상주 분할 영역에는 상대적으로 크기가 고정된 다양한 버퍼와 테이블이 들어 있습니다. 이러한 공유 메모리 부분은 디스크에서의 데이터 버퍼 지정 및 공유 메모리와 디스크 간의 이동을 관리합니다. 디스크를 비동기적으로 업데이트하는 프로세스는 부분적으로 이러한 공유 메모리 부분을 사용하여 관리합니다.

공유 메모리의 상주 분할 영역은 모든 공유 메모리를 추적하는 고정 구조를 저장합니다. 내부 테이블은 사용자, 로크, 트랜잭션, TABLESPACE(TBLSPACE), 디스크 청크, 페이지 클리너 및 버퍼 풀을 추적하기 위해 작성됩니다. 다음과 같은 여러 가지 내부 테이블이 공유 메모리 자원을 추적합니다.

- 버퍼 테이블과 해시 테이블
- 청크 테이블
- DBSPACE 테이블
- 로크 테이블과 해시 테이블
- 페이지 클리너 테이블
- TABLESPACE(TBLSPACE) 테이블과 해시 테이블
- 트랜잭션 테이블

• 사용자 테이블

버퍼 테이블

해시 테이블은 신속한 조회 메커니즘을 제공하는 데 사용됩니다. 이 테이블은 공통 알고리즘을 사용하여 숫자 또는 문자열을 메모리 위치에 매핑합니다. Informix 엔진에서 내부적으로 해시 테이블을 자주 사용하며 특히 버퍼 테이블, 로크 테이블 및 TABLESPACE (TBLSPACE) 테이블에 사용됩니다.

버퍼 테이블은 개별 버퍼의 상태를 추적합니다. 상태는 "비어 있음", "수정되지 않음" 또는 "수정됨"입니다. 수정된 버퍼나 더티 버퍼는 결국 디스크에 작성되어야 합니다. 이를 수행하는 방법은 LRU (least recently used) 큐 메커니즘에 의해 관리되도록 하는 것입니다. 공유 메모리의 버퍼가 공유 자원이므로 버퍼 로크를 통해서 동시성을 관리해야 합니다. 버퍼 로크는 여러 사용자가 다른 사용자의 작업을 방해하지 않고 버퍼 페이지에 액세스할 수 있도록 하는 메커니즘을 제공합니다. Informix 엔진에서는 공유 로크와 배타적 로크 두 가지가 지원됩니다. 버퍼 테이블은 각 버퍼에 대한 현재의 로크 레벨을 추적합니다.

상주 메모리로 지정된 메모리 부분은 실제적으로 메모리에 보존되어야 합니다. UNIX 및 기타 운영 체제에서는 메모리 부분이 디스크(가상 메모리)로 스왑되어 다른 프로그램에서 물리 메모리를 사용할 수 있도록 합니다. 모든 운영 체제는 아니지만 대부분의 운영 체제가 이러한 메모리 세그먼트를 상주시킴으로써 메모리 부분을 RAM에 보존하는 옵션을 지원합니다. IDS 상주 메모리가 상주 상태로 남아 있으려면, 운영 체제가 상주 메모리 블록을 지원해야 하고 운영 체제에 대한 IDS RESIDENT 상주 매개변수가 1로 설정되어야 합니다. 이러한 사항은 메모리 부분에 제한을 많이 받는 IDS 설치를 제외한 모든 경우에 권장됩니다.

Informix 엔진은 동시성을 지원하므로 두 명 이상의 사용자가 버퍼에 액세스하려고 대기하는 경우가 발생할 수 있습니다. 버퍼 테이블에서는 각 스레드의 버퍼와 로크 레벨을 사용하려고 대기하는 사용자(또는 스레드)를 추적하는 액세스 목록을 유지보수합니다.

청크 테이블

청크 테이블은 Informix 서버의 모든 청크를 추적합니다. 미러링이 사용 가능하면, 별도의 미러 청크 테이블도 생성됩니다. 이 청크 테이블에는 제공된 DBSPACE에 대한 청크 번호와 다음 청크의 번호가 들어 있습니다.

DBSPACE 테이블

DBSPACE 테이블은 시스템의 모든 DBSPACE를 추적할 때 사용합니다. 이 DBSPACE 테이블은 DBSPACE 번호, DBSPACE의 이름(DBSPACE가 생성될 때 지정된 이름), DBSPACE가 작성된 날짜와 시간 및 DBSPACE의 상태를 추적합니다.

로크 테이블

로크 테이블은 시스템에 생성된 로크의 공유 메모리 풀입니다. 스레드는 로크를 요청할 때마다 로크 테이블에 대한 엔트리를 작성합니다. 이 로크 테이블은 로크를 소유한 트랜잭션의 주소, 로크 유형, 로크된 페이지 또는 rowid(행 식별자) 및 로크가 배치된 TABLESPACE를 추적합니다. 로크는 배타적, 업데이트, 공유, 바이트 또는 의도적일 수 있습니다.

페이지 클리너 테이블

페이지 클리너 테이블은 버퍼 풀의 업데이트된(더티) 페이지를 디스크에 써야 하는 스레드인 페이지 클리너의 상태를 추적합니다. 이 페이지 클리너 테이블은 페이지 클리너의 상태와 페이지 클리너가 마지막으로 사용된 시간에 관한 정보를 추적합니다.

TABLESPACE(TBLSPACE) 테이블

TABLESPACE 테이블은 IDS 세션에서 현재 사용 중이며, 엔진에서 활성 상태로 간주하는 TABLESPACE를 추적합니다. 활성 TA

BLESPACE로는 테이블, 임시 테이블 및 내부 제어 테이블이 있습니다.

트랜잭션 테이블

트랜잭션 테이블은 엔진에서 현재 활성화된 모든 트랜잭션을 추적합니다.

사용자 테이블

사용자 테이블은 시스템에서 활성 상태에 있는 사용자 스레드에 관한 정보를 추적합니다.

공유 메모리 버퍼 풀

공유 메모리 버퍼 풀은 공유 메모리의 상주 부분에 들어 있습니다. 공유 메모리 버퍼 풀은 가장 큰 상주 공유 메모리 부분으로, 쿼리에서 요청한 데이터베이스 데이터 페이지가 들어 있습니다. 데이터베이스에서 처리해야 하는 모든 데이터는 우선 공유 메모리 버퍼 풀로 읽어드려야 합니다.

버퍼 페이지는 big buffer에서 수행되는 입출력과 BLOBSPACE에서 수행되는 입출력을 제외한 하나의 IDS 페이지 크기입니다. big buffer의 크기는 32 버퍼 페이지입니다. big buffer는 BLOB을 읽고 쓰며, 디스크에서 데이터를 순차적으로 읽는 대형 디스크 읽기와 쓰기의 성능을 향상시킬 때 사용됩니다. IDS 엔진이 인접하는 다중 디스크 페이지를 써야 한다고 결정하면 big buffer를 사용합니다. big buffer를 공유 메모리로 읽어 오면 디스크 페이지가 공유 메모리 버퍼 풀의 일반 버퍼로 할당됩니다.

논리 로그 버퍼

논리 로그는 마지막 백업 이후의 데이터 변경에 대한 레코드를 저장합니다. 논리 로그를 사용하여 트랜잭션을 관리하고 사용자나 응용프로그램에 따라 트랜잭션을 롤백하거나 트랜잭션의 실행을 취소할 수 있습니다. 또한 데이터베이스를 특정 시점으로 복구하는 기능

을 제공할 때도 논리 로그를 사용할 수 있습니다.

IDS 엔진은 성능 향상을 위해 공유 메모리에 로그 엔트리 버퍼를 지정한 후에 엔트리를 디스크에 씁니다. 엔진에는 세 가지 논리 로그 버퍼(그림 2-3 참조)가 있지만 한 번에 하나만 사용합니다. 엔진은 첫 번째 논리 로그를 디스크에 쓰는 동안 로그 엔트리를 계속해서 쓸 수 있도록 두 번째 논리 로그를 현재의 논리 로그로 만든 후에 현재의 로그 버퍼를 디스크에 씁니다. 첫 번째 논리 로그를 디스크에 쓰기 전에 두 번째 논리 로그가 채워지면, 세 번째 논리 로그를 사용하여 첫 번째 논리 로그를 디스크에 쓰는 동안 엔진이 계속해서 로그 엔트리를 공유 메모리 버퍼에 쓸 수 있도록 합니다.

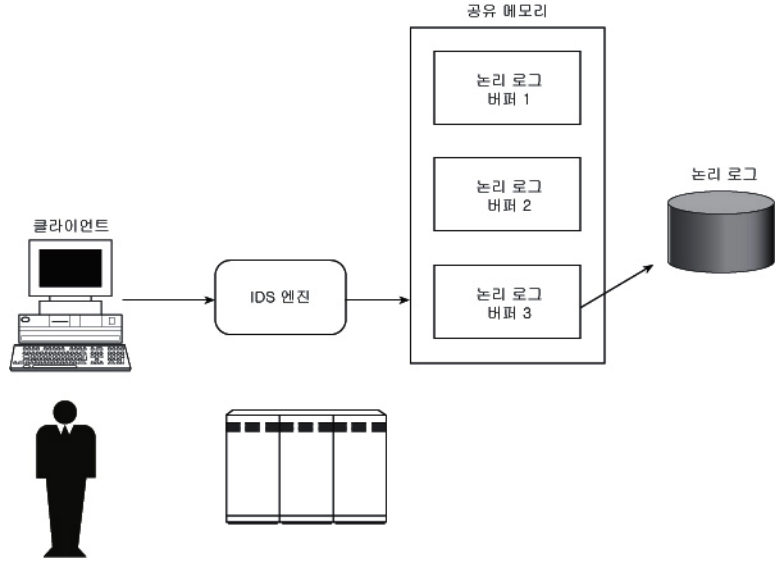


그림 2-3 IDS의 논리 로그 버퍼 풀

물리 로그 버퍼

물리 로그는 시스템 고장 후에 데이터베이스를 일관성 있게 복구할 때 사용합니다. 앞에서 언급한 대로 물리 로그는 업데이트 중인 데이터베이스 페이지의 사전 이미지를 저장합니다.

물리 로그는 공유 메모리에 있는 두 개의 버퍼를 사용합니다(그림 2-4 참조). 논리 로그 버퍼의 경우와 마찬가지로 하나의 물리 로그 버퍼를 디스크에 쓸 때 물리 로그를 디스크에 쓰는 동안 다른 버퍼를

사용하여 데이터베이스가 계속해서 사전 이미지를 쓸 수 있습니다.

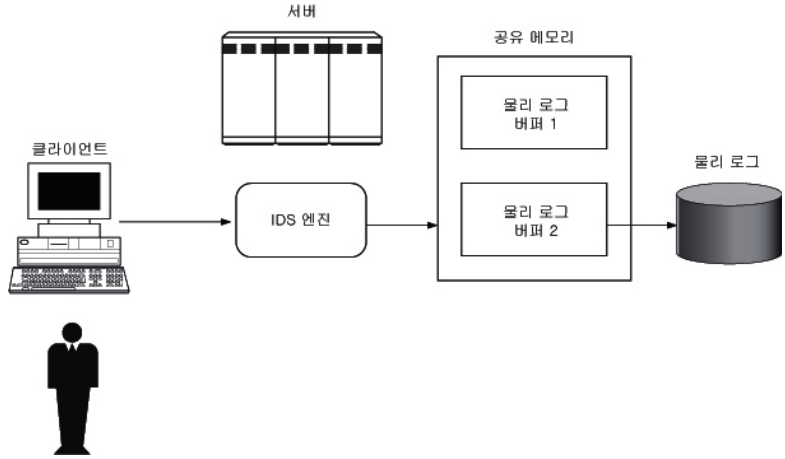


그림 2-4 물리 로그 버퍼

데이터 복제 버퍼

데이터 복제를 사용하여 IDS 엔진(기본 인스턴스)을 다른 머신(보조 인스턴스)으로 복제할 수 있습니다. 인스턴스 하나가 오프라인 상태가 되면 다른 인스턴스가 기본 인스턴스가 되어 데이터베이스의 트랜잭션을 계속해서 실행하고 처리할 수 있습니다.

IDS의 경우, 논리 로그를 사용하여 데이터 복제를 구현합니다. 기본 인스턴스가 공유 메모리의 버퍼에 논리 로그 엔트리를 보유한 후 이를 보조 인스턴스로 전달합니다. 그러면 보조 인스턴스가 로그 엔트리를 사용하여 데이터베이스 인스턴스를 롤포워드함으로써 기본 인스턴스와 동일하게 만듭니다. 데이터 복제 버퍼의 크기는 논리 로그 버퍼와 같습니다.

LRU 관리

공유 메모리 내에서 보통 버퍼는 LRU 버퍼 큐를 통해 구성되고 관리됩니다. 버퍼 풀의 크기가 일반적으로 데이터베이스의 크기보다 훨씬 더 작기 때문에 디스크의 데이터와 버퍼 풀 간에 상당한 상호 작용이 있을 수 있습니다. 큰 테이블을 스캔해야 하지만 테이블의

크기가 버퍼 풀의 크기를 초과한 경우(흔히 발생함), 데이터를 버퍼 풀 안과 밖으로 이동시켜야 합니다. 어떤 페이지를 메모리에 보관하고 어떤 페이지를 메모리에서 제거하거나(업데이트되지 않은 경우) 디스크에 써야 할지(업데이트된 경우)는 LRU 큐 관리에서 결정됩니다.

LRU 큐는 링크된 목록 쌍으로 구현됩니다. 한 목록은 사용 가능하거나 또는 수정되지 않은 큐의 페이지를 추적할 때 FLRU(Free LRU) 큐라고 하는 큐를 사용합니다. 나머지 다른 목록은 큐의 수정된 페이지를 추적할 때 사용되며, 이 페이지는 MLRU(Modified LRU) 큐라는 디스크에 쓰여져야 합니다. 목록을 구성하는 것은 실제 버퍼 페이지가 아니라 버퍼 페이지를 가리키는 포인터입니다.

IDS 엔진이 디스크에서 공유 메모리 버퍼로 페이지를 이동하기 시작할 때 버퍼에 페이지가 가득 차면(흔히 있는 경우임), 공유 메모리로 읽어 와야 하는 페이지의 공간을 마련하기 위해 공유 메모리에서 이동시켜야 할 페이지를 결정해야 합니다. 엔진은 필요한 페이지를 이전에 메모리에 있었던 메모리로 이동시키기 위해서, 추가 오버헤드를 생성하여 디스크에서 페이지를 다시 읽습니다. 따라서 이러한 프로세스를 효과적으로 관리하고 가장 필요하지 않은 공유 메모리 페이지만을 제거해야 합니다. 이것이 바로 LRU 큐의 목적입니다(그림 2-5 참조).

FLRU 버퍼에 대한 링크된 목록은 LRU 버퍼의 순서대로 정렬됩니다. 공유 메모리 버퍼에서 페이지에 마지막으로 액세스했을 때 이는 페이지 사용으로 간주됩니다. 엔진에서 페이지를 공유 메모리 버퍼에 두고 버퍼에서 페이지를 제거해야 할 경우, 엔진은 FLRU 버퍼 목록을 무작위로 선택한 후 큐에서 LRU 버퍼를 찾아냅니다. 버퍼가 래치되면, 해당 버퍼가 메모리에서 제거되고 새 버퍼 페이지가 공유 메모리에 배치됩니다. FLRU 큐가 래치되면, IDS 엔진에서 다른 큐를 무작위로 선택하여 버퍼를 선택하려고 합니다.

공유 메모리의 가상 분할 영역

공유 메모리의 가상 분할 영역에는 세션과 관계된 다양한 구조와 버퍼가 들어 있습니다. 세션이 끊임없이 시작하고 종료되기 때문에 공유 메모리의 가상 분할 영역은 상대적으로 유동적이며 IDS 엔진에서 동적으로 확장이 가능합니다. 세션에 대한 세션 제어 블록과 세션에 대한 쿼리를 처리하는 메모리가 끊임없이 할당되고 할당해제됩니다.

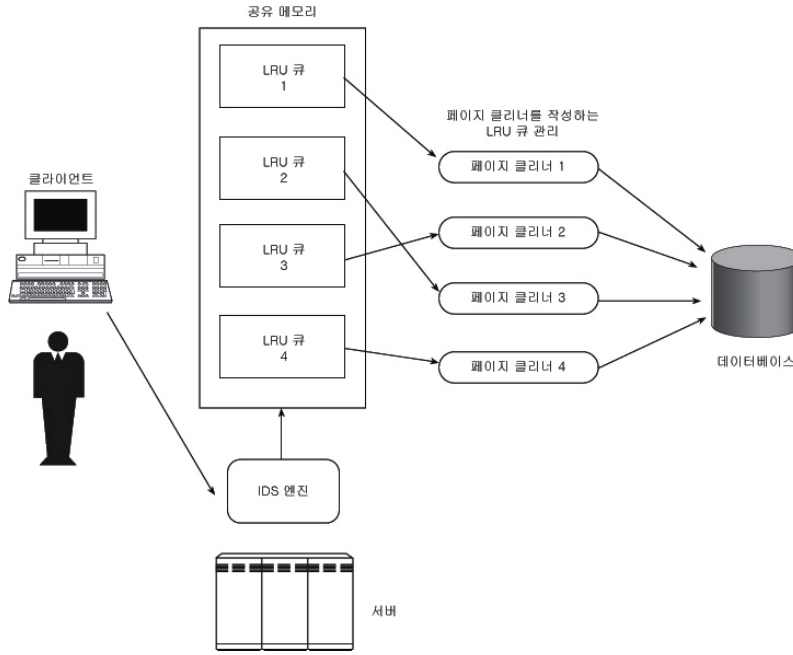


그림 2-5 LRU 큐 관리

메모리 내 정렬을 수행하고 세션에서 일반적으로 사용할 수 있도록 가상 공유 메모리 버퍼가 할당됩니다. 엔진이 메모리의 가상 분할 영역에서 충분히 큰 세그먼트를 찾을 수 없는 경우, 엔진은 IDS 구성 매개변수로 설정된 한도까지 가상 메모리 세그먼트를 추가하려고 합니다. 사용자 세션이 종료되면, 엔진에서 할당된 메모리를 사용할 수 있습니다.

공유 메모리의 가상 분할 영역에는 다음사항이 저장됩니다.

- big buffer
- 세션 데이터
- 스레드 데이터
- 사전 캐시
- 내장 프로시저 캐시
- 정렬 풀
- 전역 풀

다음 절에서는 이 구성 요소에 대해 설명합니다.

Big Buffer

Big buffer는 많은 양의 데이터를 디스크에 씁니다. big buffer를 사용하여 테이블을 순차적으로 읽어 DBSPACE BLOB를 읽고 씁니다. big buffer를 공유 메모리로 읽어 오면, 보통 버퍼에 할당됩니다.

세션 데이터

세션 데이터는 공유 메모리의 가상 분할 영역의 세션 제어 블록에 보관됩니다. 이러한 블록에는 세션 ID, 사용자 ID, 클라이언트 프로세스의 프로세스 ID, 호스트의 이름 및 사용자 세션의 상태를 나타내는 상태 플래그가 들어 있습니다. 세션이 종료되면, 세션 제어 블록이 사용 가능하게 되며 메모리는 가상 공유 메모리 풀로 리턴됩니다.

스레드 데이터

클라이언트 프로세스가 세션을 시작하면, 세션이 최종적으로 스레드를 시작하여 작업을 수행합니다. 스레드는 스레드 제어 블록에서 관리합니다.

엔진이 한 스레드에서 다른 스레드로 전환하면 현재 스레드의 정보는 스레드 제어 블록에 보관됩니다. 프로그램 카운터와 전역 포인터와 같은 정보는 공유 메모리의 스레드 제어 블록에 보관됩니다.

스레드에 사용되는 공유 메모리 영역에는 프로그램 스택 및 힙에 할당된 영역도 포함됩니다. 스택의 크기는 구성 가능하며 기본값은 32KB입니다. 엔진에서 실행되는 모든 스레드의 스택 크기는 STACKSIXE 환경 변수로 설정됩니다. INFORMIXSTACKSIZE 환경 변수를 사용하여 특정 사용자 세션의 스택 크기를 설정할 수 있습니다.

프로그래밍 용어에서 힙은 전역 스토리지의 메모리에 있는 영역입니다. IDS 엔진에서 실행 중인 스레드의 경우, 이 영역은 동일한 목적으로 제공되며, 스레드별로 스토리지에 사용된 메모리에 동적으로 할당된 영역입니다. 스레드 힙 크기는 변경할 수 없습니다.

사전 캐시

IDS 엔진은 시스템 카탈로그라고 하는 일련의 시스템 테이블에 데이터베이스에 관한 데이터를 저장합니다. 시스템 카탈로그 테이블은 데이터베이스 시스템에 있는 실제 테이블이지만 이 정보가 필요할 때마다 데이터베이스 테이블에 액세스하는 오버헤드를 피하기 위해 사용자 스레드가 액세스하는 시스템 카탈로그 부분을 공유 메모리로 읽어드립니다.

내장 프로시저 캐시

내장 프로시저를 먼저 작성하면 최적화된 의사 코드 폼으로 컴파일됩니다. 이 최적화된 의사 코드 폼은 시스템 카탈로그 테이블에 저장된 후 내장 프로시저를 실행할 때 읽혀집니다. 내장 프로시저가 실행될 때마다 시스템 카탈로그 테이블을 읽는 오버헤드를 피하기 위해, 내장 프로시저에 대한 반복 호출이 시스템 카탈로그에서 가상 메모리의 내장 프로시저로 프로시저를 읽습니다.

정렬 풀

IDS 엔진은 정렬을 수행해야 할 경우, 공유 메모리의 가상 분할 영역에서 가능한 많은 정렬을 수행하려고 합니다. 정렬에 할당된 메모리 양은 정렬할 행의 수와 행의 크기에 따라 다릅니다. Informix 메모리 부여 관리자(MGM)는 PDQPRIORITY가 0보다 더 큰 값으로 설정되었을 때 정렬에 사용되는 메모리를 할당해야 합니다. 정렬에 할당할 총 메모리 양의 한도는 DS_TOTAL_MEMORY 구성 매개변수에 의해 설정됩니다.

전역 풀

전역 풀은 통신 폴 스레드에 사용되는 메시지 큐와 같이, IDS 엔진에 대해 전역인 구조를 저장할 때 사용합니다. 모든 IDS 스레드가 메모리의 전역 풀 영역에 액세스할 수 있습니다.

공유 메모리의 통신 부분

공유 메모리의 통신 부분을 끝으로 데이터베이스 메모리에 대한 설명이 마무리됩니다. 공유 메모리의 통신 부분은 메모리에 첨부된 프로세스와 엔진 간의 통신을 관리할 때 사용합니다. 공유 메모리의 통신 부분은, 엔진에 연결되어 데이터베이스 자원을 요청하는 프로세스와 엔진 사이의 메시지 순서를 관리합니다. 모든 프로세스 간 통신은 공유 메모리의 통신 부분을 통해서 관리됩니다.

IDS 아키텍처: 스레드(프로세스)

Informix 서버는 고유의 내부 스레딩 메커니즘을 사용하여 다중 스레드를 제공합니다. 스레드는 비중이 적으며(시작과 중지에 시스템 자원이 거의 필요하지 않음), 전역 메모리 영역을 공유합니다. 스레드는 비중이 작기 때문에 스레드 간에 제어를 전송하는 것이 운영 체제 프로세스 간에 제어를 전송하는 것보다 더 빠르고 효과적입니다.

엔진 작업의 내부 다중 스레드를 통해 개별 SQL 작업을 병렬 처리할 수 있습니다. 제공된 SQL 작업의 경우 스레드를 많이 생성하여 쿼리 부분을 병렬로 실행할 수 있습니다. Informix 엔진은 SQL 문의 어떤 부분을 병렬로 처리해야 할지에 대해 지능적인 의사 결정을 내립니다. 조인된 두 개의 테이블이 동일한 디스크에 있어야 한다고 결정하면 엔진은 테이블을 병렬로 읽는 두 개의 스레드를 생성하지 않습니다. 이는 두 개의 스레드가 모두 동일한 디스크를 액세스하려고 시도함으로써 경쟁이 발생하기 때문입니다. 조인된 두 개의 테이블이 동일한 디스크에 있지 않거나 테이블이 여러 디스크에 걸쳐 분할되어 있으면 다중 스캔 스레드가 시작되어 디스크를 스캔합니다. 이런 방법으로 내려진 분할화 결정이 데이터에 대해 실행되는 쿼리의 병렬 처리에 직접적인 영향을 미칠 수 있습니다.

병렬로 쿼리 실행

Informix 옵티마이저는 작업을 가져와 이를 구성 요소 하위 작업으로 나눕니다. 이러한 작업을 모두 병렬로 실행할 수는 없습니다. SQL 서브쿼리와 트리거를 병렬로 실행할 수 없는 경우도 있습니다. 그러나 대다수의 쿼리에 병렬 처리의 이점이 적용되며, 그 중에서도 가장 두드러진 것이 의사 결정 지원 쿼리입니다.

병렬 처리의 이점은 SMP 머신과 같은 다중 프로세서에서 가장 잘 적용됩니다. 이러한 플랫폼에서는 병렬 스캔과 정렬 작업을 다중 CPU에서 실행할 수 있어, 실행 시간이 오래 걸리는 쿼리의 처리 시간이 대부분 현저히 줄어듭니다.

IDS 아키텍처: 통신

IDS 아키텍처의 통신 계층에서 데이터베이스 엔진과 데이터베이스에 연결되어 서비스를 요청하는 사용자 프로세스 간의 통신을 관리합니다. 이 통신 계층은 다른 서비스 가운데서도 연결을 위해 리스닝하며, 사용자 프로세스를 추적하고, 데이터베이스 사이에서 전송되는 데이터의 버퍼를 지정합니다.

Informix 데이터베이스는 여러 가지 프로세스 간 통신 메커니즘과 네트워크 프로토콜을 지원합니다. 따라서 플랫폼과 클라이언트 연결에서 많은 유연성을 제공합니다. 지원되는 프로세스 간 통신 메커니즘은 다음과 같습니다.

- 공유 메모리
- 스트림 파이프 통신
- TCP/IP 네트워크 프로토콜
- IPX/SPX 네트워크 프로토콜

공유 메모리 통신

공유 메모리 통신은 가장 신속한 통신 유형입니다. 이러한 통신은 클라이언트 프로세스와 엔진 모두에서 볼 수 있는 전역 메모리를 통해 수행됩니다. 클라이언트 프로세스가 메시지와 데이터를 메모리 위치에 씁니다. 그러면 서버는 데이터를 동일한 메모리 위치에 읽고 씁니다. 다중 프로세서가 메모리를 공유하므로 로크는 메모리에 대한 동시 액세스를 제어하는 데 사용됩니다. 이러한 IPC 메커니즘은 클라이언트 프로세스가 서버와 동일한 머신에서 작동되어야 합니다.

스트림 파이프 통신

스트림 파이프 통신 메커니즘은 스트림을 통한 프로세스 간 통신을 사용하며, 운영 체제 호출을 사용하여 클라이언트와 서버 간에 데이터를 이동합니다. 이 경우에도 공유 메모리의 경우와 마찬가지로 클라이언트와 서버가 동일한 머신에 있어야 합니다.

TCP/IP 및 IPX/SPX

TCP/IP와 IPX/SPX 통신 메커니즘 모두, IDS가 지원한 네트워크 통신 프로토콜을 나타냅니다. 이러한 IPC 메커니즘을 동일한 머신에서 사용하거나 원격 머신을 통해 사용할 수 있습니다. 이 메커니즘은 공유 메모리 통신만큼 빠르지는 않지만, 대부분의 PC 기반 클라이언트/서버 통신이 관리되는 메커니즘입니다.

엔진을 통해 쿼리 추적

Informix 엔진에서 발생하는 처리를 잘 이해하기 위해서는 엔진을 통해 쿼리의 실행을 추적하는 것이 유용합니다. 사용자가 Informix 엔진에 연결되면, 세션 제어 블록이 메모리에 생성됩니다. 이 세션 제어 블록은 연결 상태를 추적할 때 사용되는 내부 프로그램 구조 세트입니다. 사용자가 SQL문을 실행하면 쿼리는 세션 제어 블록의 구조를 통해 추적됩니다.

쿼리를 실행하는 첫 번째 단계는 쿼리의 구문 분석입니다. 구문 분석기가 처리 중인 SQL문을 검사합니다. 구문 분석기는 먼저 명령문의 구문이 올바른지 판별합니다. 구문이 올바르면, 데이터베이스 객체(대개 데이터베이스 테이블)를 검사하여 데이터베이스 객체가 존재하는지 여부를 결정하고, 존재하는 경우 사용자에게 데이터베이스 객체에 액세스할 사용 권한이 있는지를 판별합니다.

다음 단계는 데이터 검색에 있어 가장 효율적인 쿼리 경로를 선택하는 프로세스인 쿼리 최적화 단계입니다. 쿼리 최적화는 가장 효율적인 쿼리 처리 방법을 결정하는 매우 중요한 기능입니다. Informix는 데이터베이스 업계에서 Informix 옵티마이저를 최고로 만들기 위해 아주 많은 노력을 기울였습니다. 평가 프로세스의 일부로, 구문

분석기가 쿼리의 데이터베이스 객체의 위치를 살펴봅니다. 개별 테이블이 별도의 DBSPACE에 있거나 테이블이 다중 DBSPACE에 걸쳐 있으면(분할된 상태), 옵티마이저는 다중 스캔 스레드 시작을 선택하여 이 데이터를 검색합니다. 데이터가 디스크의 개별 분할 영역에 상주하므로 다중 스캔 스레드가 잠재적으로 병렬로 작동할 수 있으며, 다중 디스크를 동시에 스캔하고 심지어 검색된 데이터를 병렬로 처리할 수도 있습니다.

데이터 검색 방법을 결정할 때 다른 요소도 살펴봅니다. 어떤 경우에는 옵티마이저가 인덱스를 사용할지 여부를 결정해야 하며, 한 테이블에 다중 인덱스가 있으면, 사용할 인덱스를 결정해야 합니다. 이러한 의사 결정은 부분적으로 인덱스의 선택 빈도에 기초합니다. 인덱스의 선택 빈도는 인덱스에 고유한 것으로, 완전 고유에서 인덱스 키에 대한 어느 정도의 중복에 이르기까지 다양합니다.

대개 SQL문을 실행하려면 조인할 관계형 데이터베이스에 테이블이 필요할 수 있습니다. 대부분의 경우 조인 키는 SQL문의 where절에 지정되어 있습니다. 조인해야 테이블이 여러 개가 있는 더 복잡한 경우도 있을 수 있습니다. 구문 분석기는 다중 테이블을 조인하는 최선의 방법과 사용할 인덱스를 결정한 후에, 모든 데이터를 함께 가져오는 방법을 결정해야 합니다. 어떤 경우에는, 조인된 두 테이블의 행을 검색하고 해시 테이블을 사용하여 테이블의 어떤 행이 조인 기준(해시 조인)에 맞는지를 결정하는 것이 더 나을 수도 있습니다. 다른 방법으로는 두 테이블의 행을 조인 키에 따라 정렬한 후 병합하는 정렬 병합 조인이 있습니다.

옵티마이저는 결과를 정렬해야 할지 여부도 검사해야 합니다. 결과를 정렬하는 경우, 옵티마이저는 정렬을 수행할 수도 있고 또는 정렬을 만족시키는 인덱스를 사용할 수도 있습니다. 인덱스를 사용해야 할지 또는 정렬을 수행해야 할지에 대한 의사 결정은 앞서 말한 다른 많은 요소의 균형에 맞게 내려야 합니다.

Informix 옵티마이저

Informix 옵티마이저는 비용 기반의 옵티마이저로, 사용 가능한 다양한 액세스 경로 대체 방법에 대한 통계 비용에 따라 의사 결정을 내립니다. 예를 들어, 선택 빈도가 낮은 인덱스(중복이 많은 인덱스)에는 선택 빈도가 높은 인덱스(고유 인덱스 또는 중복이 적은 인덱스)보다 높은 비중이 지정됩니다. 경로의 각 선택 사항에 대한 통계 비중이나 비용의 총계를 내서 다양한 데이터 액세스 경로 대체 방법을 평가합니다.

기본적으로 Informix 옵티마이저는 특정 경로를 배제하지 않고 가능한 모든 사항을 다 검색하여 비용이 가장 낮은 옵션을 선택합니다. 이러한 작업에 시간이 많이 소요되는 것 같지만, 필요한 경과 시간은 최소이며 궁극적으로는 향상된 경로 선택의 이점을 통해 쿼리 성능이 향상됩니다.

이러한 기본 옵티마이저 동작은 `set optimization low` SQL문을 사용하여 무시할 수 있습니다. 이러한 최적화 설정을 사용하면 옵티마이저가 액세스 경로 대체 방법을 모두 검색하지는 않지만, 의사 결정 트리의 각 분기에서 비용이 가장 저렴한 대체 방법을 선택해서 이러한 대체 방법에 따라 쿼리 액세스 경로를 평가합니다.

쿼리 결과 처리

옵티마이저가 쿼리 경로를 선택하고 나면 SQL문을 처리할 수 있습니다. 먼저 선택한 액세스 경로에 따라 쿼리를 처리하도록 자원을 할당합니다. 할당된 자원은 사용 가능한 자원과 현재 사용자 세션에 지정된 PDQ 우선 순위에 따라 다릅니다. 쿼리에 지정된 자원은 주로 쿼리를 처리할 때 필요한 스레드와 메모리입니다. PDQ 우선 순위가 높을수록 쿼리 처리에 지정된 메모리와 스레드는 많아지고, 반대로 PDQ 우선 순위가 낮을수록 지정되는 스레드는 적어집니다.

쿼리가 테이블에서 특정 수의 행을 엔진 버퍼로 검색하면, 엔진이 행을 요청하는 사용자 세션으로 행을 리턴하기 시작합니다. 이렇게 되면 엔진에 연결된 프론트엔드 프로세스와 백엔드 또는 서버 프로세스 간의 상호 작용이 발생합니다. 이러한 작업을 수행할 때 동기 프로토콜을 사용하여 프론트엔드로 행을 전송하고 수신 확인을 기다린 후 필요하면 추가 행을 전송합니다.

엔진이 행을 리턴하기 시작하는 시기는 쿼리에 따라 다릅니다. 쿼리가 두 개의 테이블을 조인하고 결과를 정렬할 경우, 모든 행을 검색하고 정렬한 후에 엔진은 결과를 리턴할 수 있습니다. 그러나 엔진이 인덱스를 읽어 리턴할 데이터의 올바른 순서를 얻을 수 있으면, 사용자에게 결과를 바로 리턴할 수 있습니다.

모든 행이 리턴되면 프로세스에 지정된 다수의 자원(스레드, 메모리)의 할당이 해제됩니다. 공유 메모리 버퍼에 있는 특정 세션 제어 블록 부분은 사용자가 엔진과 연결이 끊어질 때까지 남아 있습니다.

병렬 데이터 쿼리 및 PDQ 우선 순위

Informix 데이터베이스 엔진은 다중 스레드로서, 다중 실행 스레드를 사용하여 처리를 내부적으로 수행합니다. 이러한 스레드는 병렬 작업 수행이 가능하며 대개 병렬로 작업을 수행합니다. 엔진에서 처리에 다중 CPU를 사용할 수 있으면 엔진은 내부 병렬 처리의 이점을 활용할 수 있습니다.

Informix 서버는 다중 처리 스레드를 시작하여 필요한 작업을 수행함으로써 다양한 데이터베이스 작업을 병렬로 수행합니다. 다중 처리 스레드를 실행시켜 데이터 테이블 및 조각을 스캔하고, 데이터를 정렬하며, 디스크에 로그 파일을 쓰고, 기타 하우스키핑 또는 엔진에 필요한 기능 처리 등을 수행할 수 있습니다.

PDQPRIORITY 환경 변수는 Informix 서버가 SQL문 처리에 자원을 할당하는 방법에 직접적인 영향을 줍니다. 자원을 올바로 할당함으로써 많은 쿼리 유형의 쿼리 처리 성능을 상당 부분 향상시킬 수 있습니다.

PDQPRIORITY는 세션에 대한 쿼리 처리에 할당될 자원의 비율을 나타내는 정수값입니다. 따라서 PDQ가 20으로 설정되면 엔진 자원의 20% 쿼리 처리에 할당됩니다. 그러나 PDQPRIORITY 환경 변수가 2보다 적은 값으로 설정되어 있는 경우는 예외입니다. 이러한 값에는 표 2-10에 표시된 대로 특별한 의미가 있습니다.

표 2-10 PDQPRIORITY 값

PDQPRIORITY	결과
PDQPRIORITY OFF	PDQ를 사용하지 않음
1 이하	병렬 스캔만 수행
100 이상	사용 가능한 모든 시스템 자원을 사용하여 PDQ 쿼리 처리

PDQPRIORITY는 쿼리가 수신하는 병렬 처리에 직접적으로 영향을 줍니다. 0으로 설정된 경우, 쿼리는 병렬 자원을 전혀 수신하지 않고 연속으로 처리됩니다. 단일 쿼리의 병렬 처리가 거의 또는 전혀 도움이 되지 않는 OLTP 트랜잭션에는 이러한 설정이 적합합니다. (그러나 일반적으로 OLTP 시스템은 내부 다중 스레드를 사용하는 것이 더 유리합니다.)

PDQPRIORITY 값이 2~100사이로 설정되면 쿼리에 대한 병렬

처리의 매개변수 설정에 따라 시스템 자원의 특정 비율을 할당합니다. 이 매개변수는 쿼리의 스캔 스레드 할당에 영향을 줌으로써 병렬화에 영향을 줍니다. 매개변수가 메모리 할당에도 영향을 주기는 하지만, 쿼리에 대한 병렬 처리를 다루는 것은 스캔 스레드 할당입니다. (PDQPRIORITY에 대해서는 IDS 엔진 튜닝에 대한 5장 "IDS 엔진 모니터링 및 튜닝"에서 자세히 설명합니다.)

IDS 옵션 및 확장 기능

앞의 절에서는 코어 Informix 엔진에 대해 살펴보았습니다. Informix는 계속해서 객체-관계형 및 의사 결정 지원이라는 두 가지 전략적 방향으로 코어 엔진을 수정하고 향상시켜왔습니다. Informix에서는 객체-관계형 영역을 Informix Dynamic Server9(IDS9)로, 의사 결정 지원 영역은 Informix Extended Parallel(XPS) 옵션으로 처리했습니다. 두 가지 변형의 Informix 엔진에 대해서는 다음 절에서 설명합니다.

IDS 엔진과 IDS 엔진의 기초 코드 스트림은 그 동안 계속해서 발전해 왔습니다. 소프트웨어 제품의 성장은 일반적으로 기능의 추가를 의미하며, 데이터베이스 서버와 같은 제품의 경우에는 성능 향상을 의미합니다. Informix Dynamic Server의 경우도 여기에 해당합니다.

Informix 서버의 경우에 가장 두드러진 것은 서버 제품을 두 가지 방향으로 차이를 둔 것입니다. 그 중 하나가 MPP 지원 영역입니다. MPP 환경의 강력한 힘을 이용하기 위해 IDS 아키텍처 구현을 약간 변경했습니다. 이러한 변경으로 출현한 것이 Informix Extended Parallel(XPS) 서버입니다. 다른 한 영역은 Informix Dynamic Server에 기초하여 객체-관계형 데이터베이스를 제공하는 것이었습니다. 이러한 노력의 결과가 Informix Dynamic Server9(IDS9)입니다.

XPS의 아키텍처

Informix Extended Parallel 서버는 대량 병렬 머신의 특징을 활용하도록 설계되었습니다. 대량 병렬 처리 머신은 고속 상호 연결을 통해 연결된 일련의 Distinct 서버로 구성되어 있습니다. 이러한 Dis

tinct 서버를 노드라고 합니다. XPS 제품은 각 노드에서 코서버로 알려진 별도의 데이터베이스 인스턴스를 실행합니다. 코서버는 고속 상호 연결을 통해 통신합니다. 이러한 코서버는 Informix 서버 아키텍처 구현을 변경하여 MPP 아키텍처에서 최상의 성능을 얻을 수 있습니다.

XPS 엔진은 MPP 아키텍처 기능을 활용하여, 쿼리 처리의 중앙 코디네이션에 내재한 병목 현상을 막고 코서버에서 실행되는 Distinct IDS 인스턴스에 쿼리 처리를 분산함으로써 최대의 병렬 성능을 실현합니다.

XPS 아키텍처는 코서버 간에 자원을 전혀 공유하지 않는, 즉, 아무 것도 공유하지 않는 아키텍처입니다. Informix 코어 서버 제품에 할당된 자원은 XPS 엔진에서는 약간 다른 식으로 할당됩니다. XPS의 경우, 쿼리를 처리하는 개별 코서버가 쿼리에만 사용되는 많은 양의 자원 할당 작업을 맡습니다. 코서버가 서로 통신하고 코디네이트하는 동안, 쿼리를 처리하는 코서버 각각이 데이터 검색에 얼마나 많은 메모리와 스레드를 할당해야 하는지에 대한 의사 결정을 내릴 수 있습니다.

XPS에서의 쿼리 처리

XPS 서버는 하나의 서버 역할을 하는 코서버의 컬렉션입니다. 사용자는 단일 코서버(선택적으로 XPS 인스턴스에 있는 코서버 중 하나 일 수 있음)에 연결됩니다. 사용자가 연결된 코서버를 연결 코서버라고 합니다. 이 연결 코서버는 쿼리 처리를 코디네이트해야 합니다. 다중 코서버 전역에서 쿼리를 코디네이트하면 초기 코서버에 병목 현상이 발생할 수 있습니다. 그러나 쿼리 계획을 서브계획으로 분할한 후 참여 코서버가 쿼리를 처리하도록 하는 기능을 통해 이러한 코디네이션의 역할이 줄어듭니다. 연결 코서버는 이러한 구현을 통해 쿼리 처리를 시작하고, 고유의 모든 처리를 수행한 후 참여 코서버로부터 나오는 결과를 수집해야 합니다.

각 노드에서 실행되는 IDS 인스턴스가 실현할 수 있는 내부 병렬 처리로 인해, XPS는 MPP 환경에서 SMP 클러스터 노드와 함께 잘 작동됩니다.

XPS 구현이 제공되면, 쿼리 처리는 개별 코서버의 성능에 따라 달라진다는 점에 유의하십시오. 쿼리의 처리 속도는 쿼리의 해당 코서버 중 가장 느린 코서버의 속도입니다. 마찬가지로, 단일 참여

코서버에 쿼리 부분을 처리할 수 있는 적절한 자원이 없거나 또는 단일 노드가 실패하는 경우, 전체 쿼리가 실패합니다.

XPS 서버는 구조적으로 코어 IDS 서버와 동일합니다. 차이가 있는 부분은 자원 할당 구현입니다. 메모리 할당은 참여 코서버의 몫입니다. 각 코서버는 처리해야 할 쿼리 서브계획을 수신합니다. 이 처리는 주로 격리되어 수행되며, 코서버는 MPP 노드에서 사용 가능한 자원만 사용할 수 있습니다. 이 시점에서 메모리 할당은 IDS 엔진에서와 마찬가지로 수행됩니다. 처리 스레드가 메모리를 요청하면, 메모리를 수신하거나 또는 메모리가 거부된 후 이에 따라 조치를 취합니다.

CPU 자원을 할당하는 것도 마찬가지입니다. 참여 코서버가 쿼리 서브계획을 수신한 후 옵티마이저가 데이터 검색 및 처리에 몇 개의 스레드를 할당할지를 결정하게 합니다.

IDS9 아키텍처

IDS9 엔진은 IDS 엔진의 기본 아키텍처입니다. 이러한 두 제품 간의 차이는 이 서버의 IDS9 객체-관계형 기능 구현에서 찾아볼 수 있습니다.

IDS9 서버는 오랜 시간에 걸쳐 그 유효성이 증명된 기능과 관계형 데이터베이스의 성능을 객체 지향 데이터베이스의 프로그래머 생산성 기능과 결합시킨 객체-관계형 데이터베이스 서버입니다.

IDS9 엔진에서는 서버 함수 생성(사용자 정의 함수)을 통한 엔진의 기능 확장 능력, 테이블과 row 자료형 상속 및 사용자 정의 자료형이 허용됩니다. IDS9 서버 함수는 C 언어, Informix 내장 프로시저 언어(SPL), Java 언어(1998년 여름)로 선언할 수 있습니다. 데이터베이스 테이블과 row 자료형 정의를 이러한 자료형 확장을 인식하는 함수와 결합한 것이 DataBlade라는 패키지입니다.

IDS9 엔진은 서버 함수를 엔진의 직접 확장으로 실행합니다. 해당 함수는 엔진 코드의 일부로 실행됩니다. (엔진에 의해 공유 라이브러리 함수로서 로드됩니다.)

기존의 ANSI 자료형(row 자료형)으로부터 자료형을 작성하고 사용자 정의 자료형을 선언하는 함수는 중요합니다. 이러한 자료형 확장의 개념은 객체 지향 프로그래밍의 중심입니다.

객체 지향 프로그래밍에서는 데이터가 개별 엔티티가 아니라 객체라고 알려진 함수(또는 메소드라고 함)에 패키지로 묶이거나 요약된 것으로 간주합니다. 또한 자료형 확장 기능을 통해 오디오, 비디오

또는 디지털 사진과 같은 rich 자료형을 데이터베이스에 저장하고 서버 함수를 통해 표준 SQL 자료형과 유사한 방식으로 처리할 수 있습니다.

IDS9 서버 함수는 데이터를 처리하기 전에 IPC 메커니즘을 통해 데이터를 제공해야 하는 미들웨어 솔루션과는 달리, 엔진의 공유 메모리 버퍼에 직접적으로 액세스합니다. 따라서 미들웨어 솔루션보다 성능이 더 뛰어납니다.

테이블과 행 상속은 이러한 객체 지향 함수를 제공하는 SQL문을 통해 처리됩니다. 이러한 명령문을 사용하여 row 자료형 작성, row 자료형을 사용하는 테이블 작성 및 상속된 테이블의 수퍼 세트인 테이블 작성 등과 같은 다양한 객체-관계형 함수를 수행할 수 있습니다.

IDS9에서의 상속

상속 기능은 객체 지향 코드 재사용이라는 개념의 중심입니다. 상속 기능을 통해 상속된 객체, 데이터 및 메소드의 모든 기능이 상속 객체의 일부가 될 수 있습니다. 객체-관계형 데이터베이스의 세계에서 상속의 개념은 테이블 스키마에 적용됩니다. 테이블 선언을 통해 이름이 지정된 테이블이나 row 자료형의 스키마를 검색하고 해당 스키마를 나름대로 효과적으로 사용할 수 있습니다. 이러한 작업은 테이블 A가 테이블 B에서, 테이블 B는 테이블 C에서 상속되는 방식으로 계층 구조를 통해 진행할 수 있습니다.

요약

이 장에서는 IDS 엔진의 기본 사항에 대한 설명으로 시작하여, 공유 메모리 버퍼, 다중 스레드의 개념, 가상 프로세서 등을 다루었습니다. IDS 엔진을 설명할 때 필요한 구성 요소로서 이러한 항목과 기타 주요 개념을 언급했습니다.

다음으로 IDS9와 XPS라는 두 개의 Informix 엔진 옵션에 대한 설명을 포함하여, IDS 엔진의 아키텍처에 대해 다루었습니다.

다음 장 미리보기

다음 장에서는 Informix 서버의 설치 및 구성에 대해 다룹니다. 이는 IDS 사용에 있어서 가장 중요한 측면 가운데 하나입니다. 서버를 잘못 구성하면 성능이 최적에 미치지 못하게 되기 쉽고 사용자의 불평과 불만족이 늘어나게 됩니다. 서버 구성을 계획하고 실행하는 프로세스는 설치 검증 단계와 함께 설명됩니다.

3장

Informix Dynamic Server 설치 및 구성

IDS 엔진은 유연성이 매우 뛰어납니다. 유연성이 뛰어난 이유는 주로 해당 엔진에 사용할 수 있는 구성 매개변수 때문입니다. IDS 서버에 제공되는 매개변수를 사용하여 메시지 로그 찾기, 로크 수 지정, 메모리 버퍼 크기 지정, 서버에 대한 논리 로그 파일 수 지정 및 기타 엔진 작업 수에 대한 튜닝 및 지정을 실행할 수 있습니다. 4장에서는 IDS 엔진 구성 매개변수에 대해 자세히 설명하며, 가능한 경우 해당 매개변수의 초기 설정 팁도 제공합니다. 매개변수에 대한 추가 내용은 IDS 엔진 모니터링 및 튜닝에 대한 장에서 다룹니다.

IDS 엔진에 대한 각각의 연결은 사용자 세션을 나타냅니다. IDS 엔진을 실행하면 부분적으로 환경 변수 세트를 통해 각 사용자 세션 작업을 제어할 수 있습니다. 3장에서는 이런 환경 변수 및 기능에 대해 설명합니다.

IDS 구성

IDS 엔진을 다양한 플랫폼에서 그리고 다양한 사용자 로드 하에서 설치할 수 있는 기능과 IDS의 유연성은 부분적으로 엔진을 구성할 수 있는 기능으로 인해 구현됩니다. 따라서 다양한 구성 매개변수를 사용할 수 없는 경우, Informix 서버를 유연하게 사용할 수 없습니다.

Informix 서버는 기본적으로 onconfig로 이름이 지정된 ASCII 파일에 구성 매개변수를 저장합니다. 이 파일은 시스템이 시작될 때 읽혀지며, 파일에는 엔진을 제어하기 위한 매개변수와 해당 값이 들어 있습니다. 엔진은 "#" 표시 뒤에 나오는 모든 문자를 무시하므로 파일의 라인에 "#"을 입력하여 주석을 포함시킬 수 있습니다.

구성 파일 이름을 변경하려면 ONCONFIG 환경 변수를 서버가 사용할 onconfig 파일의 이름을 지정하는 값으로 설정하면 됩니다. 예를 들어, ONCONFIG 환경 변수를 onconfig.myfile로 설정하면 Informix 클라이언트 소프트웨어를 실행하는 사용자는 클라이언트 소프트웨어에서 onconfig.myfile 구성 파일을 읽어 데이터베이스 인스턴스에 연결하는 방법을 결정합니다.

Informix 서버는 시작할 때 구성 파일을 읽어 엔트리를 확인합니다. 여러 가지 내부 점검을 실행하며, 일부 항목에 대해 유효성이 검증되지 못하면 엔진이 시작되지 않습니다.

설치 준비

IDS 엔진을 설치하기 전에 하드웨어 플랫폼이 준비되어야 합니다. 데이터베이스 관리자는 설치할 데이터베이스 크기 및 데이터베이스에 연결할 사용자의 수를 고려해야 합니다. 하드웨어 플랫폼에는 데이터베이스에 저장할 데이터를 보관하는 데 사용할 수 있는 적절한 디스크 공간이 있어야 합니다.

2장에 설명한 대로 Informix 서버는, 청크 컬렉션으로 표시되는 원시 디스크 및 하나 이상의 청크를, 디스크의 논리 값에 매핑하는 DBSPACE로 디스크를 인식합니다. 데이터베이스 및 테이블이 Informix 데이터베이스에 작성되면 DBSPACE에 저장됩니다.

Informix 데이터베이스는 다중 Distinct 운영 체제 프로세스에서 볼 수 있는 공유 메모리를 사용하여, 검색된 후 디스크에 쓰여진 데이터에 대해 버퍼를 지정하고 상태 정보 및 기타 시스템 제어 구조를 저장합니다. UNIX의 경우, 공유 메모리는 운영 체제의 일부로 구성해야 합니다. 이 구성에는 장치에서 사용할 수 있는 특정 공유 메모리 크기가 포함됩니다. 성능을 최적화하려면, 엔진에서 로드를 처리할 수 있도록 공유 메모리 버퍼에 사용할 수 있는 메모리가 충분해야 합니다. 시스템에서 사용할 수 있는 로크 및 세마포어의 수도 조정해야 합니다. 제품 릴리스 정보를 검토하여 변경할 구성 매개변수(설치 디렉토리의 /release 서브디렉토리에 있음)의 특정 정보를 참조하십시오.

IDS 설치

일반적으로 Informix Dynamic Server는 테이프나 CD-ROM으로 설치합니다. 장치에서 tar 또는 cpio 포맷 파일이 추출되고 설치 스크립트가 실행됩니다.

Informix 파일의 홈 디렉토리로 지정할 특정 디렉토리를 선택해야 합니다. 설치할 제품 및 제품 옵션에 따라 이 디렉토리에 일련의 서브디렉토리가 작성됩니다. 표 3-1에는 기본 서브디렉토리가 나와 있습니다.

표 3-1 IDS 서브디렉토리

서브디렉토리	내용
etc	소프트웨어 구성 파일: sqlhosts, onconfig 등
msg	오류 메시지 파일 및 도움말 파일
bin	Binaries, dlls 및 공유 객체 파일: oninit, onstat 등
lib	도구의(ESQLC/4GL) 라이브러리 파일

대부분의 경우 소프트웨어 디렉토리 구조는 고정되어 있고 사용자가 변경할 수 없습니다. 상위 디렉토리 아래에 이동할 수 없거나 이름을 재지정할 수 없는 일련의 서브디렉토리가 있습니다. 그러나 상위 디렉토리의 실제 위치는 유연하며, INFORMIXDIR 환경 변수 설정에 따라 달라집니다. 이러한 변수 설정을 보고 소프트웨어의 상위 디렉토리에 대한 전체 경로 이름을 알 수 있습니다. 예를 들어, 이 환경 변수를 /home/informix로 설정하면, Informix /etc 디렉토리에 대한 전체 경로는 /home/Informix/etc입니다

./etc 디렉토리에는 Informix 서버에서 사용하는 구성 파일이 많이 있습니다. 이 디렉토리에 있는 파일 중 하나가 sqlhosts 파일입니다. sqlhosts 파일에는 Informix 서버 이름 값, 이러한 서버에 연결할 때 사용하는 프로토콜 및 표 3-2에 나와 있는 기타 정보가 저장됩니다.

sqlhosts 파일은 다중 엔트리를 포함할 수 있으며, 각 엔트리에는 표 3-2에 나와 있는 5개 필드가 들어 있습니다.

nettype 매개변수는 서버 인스턴스에 사용할 프로토콜을 식별합니다. 모든 플랫폼에서 모든 프로토콜을 지원하는 것이 아니므로, 프로토콜을 선택하는 것은 매우 중요합니다. 잘못된 프로토콜을 사용하면 서버 인스턴스에 제대로 연결할 수 없습니다. 각 데이터베이스

릴리스에는 해당 정보에 고유한 릴리스 정보 세트가 있습니다. 해당 프로토콜 또는 설치 플랫폼에서 지원되는 프로토콜에 대해서는 이러한 릴리스 정보(./etc 디렉토리에 있음)를 참조하십시오.

표 3-2 sqlhosts 파일 엔트리

필드	설명
dbservername	연결에 대한 IDS 서버 이름. 해당 이름은 onconfig 구성 파일의 DBSERVERNAME 매개변수와 일치합니다.
nettype	nettype 매개변수는 이 데이터베이스 서버에 연결할 때 사용할 프로토콜을 표시합니다.
hostname	연결에 대한 데이터베이스 인스턴스를 포함하는 머신 호스트의 네트워크 호스트 이름을 나타냅니다. 엔트리는 머신의 /etc/hosts 파일에 대한 엔트리와 일치해야 합니다.
servicename	이 데이터베이스 서버에 연결할 때 사용할 서비스의 네트워크 서비스의 이름. 이 서비스 이름은 해당 엔트리가 사용되지 않는 공유 메모리 연결에는 적용되지 않습니다.
options	이 필드에는 엔트리에 대한 추가 옵션이 포함됩니다.

설치 프로그램 실행

테이프나 CD-ROM으로 제품을 설치했으면, 설치 프로그램을 실행하여 설치를 완료해야 합니다. 설치 프로그램을 실행하는 동안 유효한 일련 번호와 키(소프트웨어 패키지에 있음)를 입력해야 합니다. 일련 번호와 키를 입력하고 나면 설치 프로그램이 프로그램 실행 파일을 브렌드합니다. 브렌드 프로세스가 없으면 프로그램이 실행되지 않습니다.

IDS 환경 변수

Informix는 다양한 환경 변수를 사용하여 유연성과 구성을 제공합니다. 변수는 사용자 환경에서 설정되므로, 각 사용자가 환경을 사용자 정의할 수 있습니다.

특정적이며 제한된 데이터베이스 인스턴스를 사용자에게 지정함으로써, 사용자가 볼 수 있는 원격 서버를 제한하고(INFORMIXSQLHOSTS, DBPATH), 특정 우선순위(PDQPRIORITY)에 따라 특정 서버 인스턴스(INFORMIXSERVER, ONCONFIG)에 지정할 수 있습니다.

Informix 엔진에서 제공되는 환경 변수는 확장이 가능합니다. 엔진에 직접 적용되는 더 중요한 변수는 표 3-3에 나와 있습니다.

이런 환경 변수는 다양하고 때로 복잡하기도 하지만 이러한 환경 변수를 통해 Informix 엔진의 조정성과 유연성이 향상됩니다. IDS를 사용할 때 이러한 변수의 목적을 이해하는 것은 아주 중요합니다. 다음 절에서는 이 매개변수에 대해 설명합니다.

표 3-3 환경 변수

환경 변수	목적
ONCONFIG	IDS 인스턴스가 실행될 때 사용하는 Informix 구성 파일
INFORMIXSERVER	실행할 Informix 서버 인스턴스 이름(DBSERVER 환경 변수와 DBSERVERALIAS 참조)
PATH	UNIX에서 실행 파일을 검색할 때 사용하는 PATH
TERMINFO	Informix 응용프로그램에 액세스할 때 사용하는 터미널에 대한 terminfo 정의가 들어 있는 terminfo 파일
DBANSIWARN	비ANSI 표준 SQL이 사용될 경우 엔진에서 경고를 생성하게 하려면 true로 설정
DBCENTURY	날짜 또는 날짜 및 시간 값에 두 자리 연도가 지정되었을 때 적절한 확장 자리를 선택하도록 함
DBDELIMITER	문자 로드 및 언로드 기능에 사용되는 ASCII 구분 문자 지정
DBEDIT	Informix 데이터베이스 유틸리티에 사용할 텍스트 편집기 지정
DBLANG	Informix 제품에 사용되는 메시지 파일의 디렉토리 지정
DBFLTMASK	Informix 응용프로그램에 표시되는 소수 자리 수 지정 통화 값의 표시 포맷 지정
DBMONEY	Informix 데이터베이스 인스턴스가 들어 있는 네트워크
DBPATH	표시(SE 제품의 경우, Informix-SE 데이터베이스의 디렉토리 지정)
DBPRINT	Informix 제품에서 사용하는 인쇄 프로그램 지정

<계속>

표 3-3 환경 변수

환경 변수	목적
DBSPACETEMP	임시 테이블을 작성하는 DBSPACE 표시. onconfig 파일에 있는 구성 매개변수 무시
DBTEMP	Informix 게이트웨이 제품과 Informix-SE 엔진에서 생성한 임시 파일을 둘 디렉토리의 전체 경로 이름 지정
DBTIME	시간 표시에 사용할 포맷을 표시
FET_BUF_SIZE	바이트로 표시된 페치 리턴 버퍼(데이터를 서버에서 클라이언트로 리턴)의 크기 지정
INFORMIXSERVER	암시적 클라이언트 연결을 위한 Informix 서버 이름(DBSERVERNAME) 지정
INFORMIXSHMBASE	공유 메모리에서 Informix 서버 인스턴스의 기본 주소 위치 표시
INFORMIXSQLHOSTS	서버 호스트 정보가 있는 파일의 이름을 포함하여 전체 경로 이름 지정
INFORMIXSTACKSIZE	클라이언트 세션에 사용되는 스택 크기 지정. onconfig 매개변수 설정 무시
INFORMIXTERM	Informix 도구에 사용될 문자 터미널 유형. TERM 환경 변수 무시
ONCONFIG	onconfig 파일에 대한 파일 표시. $INFORMIXDIR/etc 디렉토리에 있다고 가정
PATH	실행 파일의 경로 검색. 여기에 $INFORMIXDIR/bin이 포함될 수 있음
PDQPRIORITY	의사 결정 지원 쿼리에 사용되는 우선 순위 설정(자원 할당 비율)
PSORT_DBTEMP	병렬 정렬 작업을 수행하는 동안 임시 파일을 쓰는 디렉토리
PSORT_NPROC	쿼리 정렬에 할당할 스레드 수 지정. 최대값은 10
TERM	모든 응용프로그램에 사용하는 문자 터미널
TERMCAP	문자 기반 응용프로그램에 사용하는 termcap 파일

ONCONFIG

ONCONFIG 환경 변수는 사용할 IDS 인스턴스에 대한 구성 파일의 이름을 식별합니다. 서버가 시작되면 엔진에서 구성 파일을 읽어

디스크 스토리지의 위치, 공유 메모리 주소 및/또는 공유 메모리 버퍼에 대한 오프셋, 이 파일에 포함된 기타 관련 정보 등과 같은 엔진 구성을 판별합니다.

실행 중인 IDS 인스턴스가 둘 이상인 사이트에는, 일반적으로 여러 개의 구성 파일이 있으며, 각 IDS 인스턴스의 구성 파일은 하나입니다. 사용자는 ONCONFIG 환경 변수를 인스턴스의 구성 파일 이름으로 설정하여 엔진의 특정 인스턴스에 연결합니다. INFORMIXSERVER 환경 변수에는, ONCONFIG 환경 변수 설정이 참조하는 구성 파일에 있는 DBSERVERNAME 구성 매개변수나 DBSERVERALIASES 매개변수에 있는 DBSERVER 별칭 문자열에 일치하는 서버 이름이 있어야 합니다.

INFORMIXSERVER

INFORMIXSERVER 환경 변수는 암시적 연결을 위한 Informix 서버의 이름을 지정합니다. (SQL문이나 콜 레벨 인터페이스(CLI) 함수 호출에서 서버 이름이 지정되지 않은 연결을 시도합니다.) 서버의 이름은 DBSERVERNAME 구성 매개변수 또는 ONCONFIG 환경 변수에 지정된 `onconfig` 구성 파일에 있는 DBSERVERALIASES 목록의 엔트리 중 하나와 일치해야 합니다. 연결 프로토콜을 결정할 수 있도록 해당하는 엔트리를 `sqlhosts` 호스트 프로토콜 구성 파일에서도 찾아야 합니다.

TERMINFO

TERMINFO 환경 변수는 터미널 정보 파일(터미널 정보 데이터베이스라고 함)의 이름을 지정합니다. 어떤 응용프로그램에서는 TERMINFO 환경 변수를 사용하여 응용프로그램과 터미널 간의 문자 기반 인터페이스를 정의합니다. (이는 TERMCAP 정보 파일을 대체하는 것이며 사용하는 응용프로그램에 따라 다릅니다.)

PATH

PATH 환경 변수는 시스템의 실행 파일에 대한 검색 경로를 지정

합니다. 이를 사용하기 위해서는 Informix 소프트웨어에 한 가지 요구사항이 필요합니다. Informix 실행 파일 그리고 어떤 경우에는 공유 라이브러리가 이 검색 경로에 있어야 한다는 조건입니다. 일반적으로 검색 경로에서 실행 파일을 위한 Informix 디렉토리를 Informix 홈 디렉토리에 상대적인 ./bin 디렉토리로 지정하면 이 조건이 충족됩니다.

Informix 홈 디렉토리가 INFORMIXDIR 환경 변수에 지정되어 있으므로, $INFORMIXDIR/bin 쉘 구문은 Informix 실행 파일의 전체 경로 이름이 됩니다. 다음 Bourne 쉘 구문을 실행하면 Informix 실행 파일 경로가 PATH 환경 변수에 추가됩니다.

```
PATH=$INFORMIXDIR/BIN:$PATH;export PATH
```

DBANSIWARN

DBANSIWARN 환경 변수는 엔진에서 ANSI SQL에 Informix의 확장자가 있는 구문을 검사해야 한다는 것을 나타냅니다. 엔진이 ANSI SQL에 Informix 확장자가 있는 구문을 찾으면, 경고 메시지를 표시하고 계속해서 프로세스를 처리합니다. 이 변수를 특정 값으로 설정할 필요는 없으며, 이러한 경고를 생성할 수 있도록 설정하기만 하면 됩니다.

이 환경 변수는 프로그래머가 SQL 사용법을 가능하면 ANSI SQL과 유사하게 작성하려는 사이트에 유용합니다. 데이터베이스 프로그래머의 환경에 이러한 환경 변수를 설정하면 ANSI SQL 표준에 위반되는 SQL문에 대한 즉각적이 피드백이 제공됩니다. 다음 Bourne 쉘 명령문을 실행하면 DBANSIWARN 변수가 설정됩니다.

```
DBANSIWARN=1
```

다음 명령문을 실행하면 DBANSIWARN 환경 변수의 설정이 취소됩니다.

```
unsetenv DBANSIWARN
```

DBCENTURY

DBCENTURY 환경 변수를 통해 사용자는 두 자리로 표시된 날짜

에 대한 적절한 확장을 선택할 수 있습니다. 이 DBCENTURY 변수에는 특정 동작을 사용하여 날짜 또는 날짜 및 시간 설정을 확장할 수 있는 다양한 설정이 있습니다. 확장에 대해 현재 세기(기본값), 과거 세기, 미래 세기 또는 가장 가까운 세기를 선택할 수 있습니다. 표 3-4에 열거된 값은 이러한 환경 변수에 유효한 설정입니다.

표 3-4 DBCENTURY 설정

설정	값
P	Past - 과거의 날짜 사용
F	Future - 미래의 날짜 사용
C	Closest -가장 가까운 세기 사용
R	Present - 확장에 대해 현재 세기 사용

DBDELIMITER

DBDELIMITER 환경 변수를 사용하여 데이터를 엔진에서 ASCII 데이터로 로드/언로드할 때 사용하는 구분 문자를 설정할 수 있습니다. Informix SQL 확장 명령문인 `load` 및 `unload`를 실행하면 이러한 기능은 물론 여러 가지 Informix 데이터베이스 유틸리티도 제공됩니다. 이 값은 기본적으로 ASCII 파이프 문자(|)이지만, 다른 공급자가 제공하는 많은 유틸리티에서는 구분 문자로 쉼표나 다른 문자가 사용됩니다. (구분 문자의 값은 로드나 언로드 작업을 시작하는 SQL문에 지정할 수도 있습니다.)

DBEDIT

DBEDIT 환경 변수는 Informix 문자 응용프로그램에 사용할 텍스트 편집기를 지정합니다. Unix 환경의 기본 편집기는 `vi` 텍스트 편집기입니다. Unix 환경에서 널리 사용하는 또 다른 편집기로는 GNU `emacs`가 있습니다.

DBLANG

DBLANG 환경 변수는 Informix 제품의 메시지 파일을 저장하는 디렉토리를 나타냅니다. Informix 제품은 런타임 시 이러한 메시지

파일을 읽고 이를 사용하여 정보 및 오류 메시지를 표시합니다. 메시지 파일을 요구하는 응용프로그램(ESQL, Informix-4GL)에서 이러한 메시지 파일을 찾을 수 없으면, 이 응용프로그램은 실행되지 않습니다.

DBFLTMASK

DBFLTMASK 환경 변수는 Informix 응용프로그램이 표시하는 소수 자리 수를 지정합니다. Informix는 기본적으로 소수점 오른쪽에 최대 16자리가 있는 소수를 표시합니다. DBFLTMASK 환경 변수를 사용하여 기본값을 무시할 수 있습니다.

DBMONEY

DBMONEY 환경 변수는 통화 값의 표시 포맷을 지정합니다. 포맷은 정수를 소수 자리 부분과 구분하는 문자인 선택적 선행 문자 또는 문자 세트, 트레일링 문자 또는 문자 세트를 지정합니다. 특히, 포맷은 다음과 같으며, 표 3-5에 이에 대해 설명되어 있습니다.

```
DBMONEY=<front> <separator> <back>
```

표 3-5 DBMONEY 포맷

문자	기능
front	통화값 앞에 오는 기호. 기본값은 달러 기호인 “$“임
separator	통화값의 정수 부분과 소수 자리 부분을 구분하는 문자. 기본값은 마침표 ’.’ 임
back	통화값 뒤에 오는 기호

DBPATH

DBPATH 환경 변수는 IDS 인스턴스가 들어 있는 네트워크 서버를 지정합니다. `connect`, `database`, `start database` 및 `drop database` 명령문은 DBPATH 환경 변수를 사용하여 데이터베이스의 위치를 결정합니다. 이 DBPATH 환경 변수는 데이터베이스의 위치가

명시적으로 언급되지 않았거나 데이터베이스를 기본 서버에서 찾을 수 없는 경우에만 사용합니다.

DBPATH 환경 변수는 최대 16개의 엔트리를 포함할 수 있습니다. 엔트리의 형식은 다음과 같으며, 콜론(:) 구분 문자로 구분됩니다.

```
DBPATH=//servername1://servername2
```

Informix-SE 엔진을 사용할 경우, 이러한 설정은 다른 의미를 갖게 됩니다. DBPATH 변수는 Informix-SE 데이터베이스 파일 위치의 전체 파일 시스템 경로 이름을 나타냅니다. (Informix-SE 데이터베이스는 데이터베이스 파일에 Unix 파일 시스템을 사용합니다.) DBPATH 환경 변수는 특정 등급의 위치 투명성을 제공함으로써 데이터베이스에 대한 액세스를 요청하는 응용프로그램에 실제 데이터베이스의 위치를 상대적으로 투명하게 합니다.

Informix 클라이언트 데이터베이스 연결 계층은 먼저 엔진을 로컬로 검사합니다. 로컬로 엔진을 찾을 수 없으면 연결 계층에서 DBPATH 환경 변수를 살펴보고, DBPATH 환경 변수가 설정되면 왼쪽에서 오른쪽으로 진행하는 방식으로 환경 변수에 열거된 원격 서버를 검사합니다.

사용자는 이러한 접근 방식을 사용하여 데이터베이스의 백업 버전이 있는 원격 서버로 DBPATH 환경 변수를 설정할 수 있습니다. 데이터베이스의 로컬 버전을 사용할 수 있으면, 연결 계층은 계속해서 사용자를 로컬 데이터베이스에 연결합니다. 로컬 버전을 더 이상 사용할 수 없으면, 연결 계층에서는 DBPATH 환경 변수에 열거된 원격 서버 중의 한 서버에 있는 데이터베이스를 찾으려고 합니다. 데이터베이스가 원격 머신에 연결되면 사용자는 다른 데이터베이스 서버에 연결되며, 사용자에게는 오류 메시지나 경고가 표시되지 않습니다.

DBPRINT

DBPRINT 환경 변수는 Informix 유틸리티에서 기본 인쇄에 사용할 프린터를 지정합니다. Unix 시스템의 경우, 이 DBPRINT 변수는 보통 출력을 프린터 장치로 보내는 실행 가능한 프로그램으로 설정됩니다.

DBSPACETEMP

DBSPACETEMP 환경 변수는 임시 파일을 쓰는 DBSPACE 세트를 나열합니다. 라운드 로빈 방식으로 나열된 DBSPACE를 사용하여 임시 파일을 먼저 하나의 DBSPACE에 쓴 다음, 목록의 다음 DBSPACE에 다음 임시 파일을 작성합니다. 목록의 값은 콜론(:)이나 쉼표(,)로 구분합니다. 이 환경 변수가 이와 동일한 기능을 하는 `onconfig` 매개변수 DBSPACETEMP보다 우선합니다.

DBSPACETEMP 환경 변수를 사용하여 사용자별로 임시 파일을 분배할 수 있습니다. 특정 사용자나 응용프로그램이 많은 수의 임시 파일을 생성할 것임을 알고 있는 경우, 해당 사용자나 응용프로그램은, DBSPACETEMP 환경 변수가 생산 OLTP 시스템 디스크와 다른 디스크에 상주하는 DBSPACE 세트로 설정되도록 환경을 사용할 수 있습니다. 다른 사용자는 `onconfig` 매개변수 DBSPACETEMP에 열거된 DBSPACE에 임시 파일을 계속 작성합니다.

DBTEMP

DBTEMP 환경 변수는 임시 파일에 사용되는 디렉토리의 전체 경로 이름을 지정합니다. Informix-SE 엔진이나 Informix 게이트웨이 제품에서 이러한 임시 파일을 생성할 수 있습니다. 하지만 IDS에서는 임시 파일을 이 디렉토리에 쓰지 않는다는 점에 유의하십시오.

DBTIME

DBTIME 환경 변수는 시간을 표시할 때 사용하는 포맷을 나타내는 문자열입니다. 이러한 설정에 영향을 받는 것은 특정 Informix 도구에 의한 `datetime` 변수의 출력뿐입니다.

FET_BUF_SIZE

FET_BUF_SIZE 환경 변수는 Informix 프론트엔드 도구의 리턴 버퍼 크기를 설정합니다. Informix 도구가 데이터베이스에서 데이터

를 검색하면, 이 데이터는 리턴 버퍼에 저장됩니다(일반적으로 페치 커서 작업에 의해 리턴되므로 FET(CH)_RET(URN)_BUF(FER)임). 이 버퍼의 기본값은 작은 수의 행에서 중간 수의 행(25-70)까지 검색하는 SQL `select`문이 관리할 수 있는 크기입니다.

프론트엔드 도구가 데이터베이스 서버에서 데이터를 검색하면, 서버는 먼저 데이터베이스에서 행을 읽은 후에 읽은 행을 서버의 버퍼에 배치합니다. 서버의 버퍼가 가득 차면, 프론트엔드로 보냅니다. 행을 프론트엔드로 전송하는 이 프로세스에는 행을 네트워크로 보내는 과정이 포함되므로 시간과 비용이 많이 들 수 있습니다. 따라서 네트워크 작업 수를 줄이는 것이 바람직합니다.

성능이 저조할 때, FET_BUT_SIZE를 큰 수로 설정하면 성능을 향상시킬 수 있습니다. FET_BUT_SIZE를 올바르게 설정할 경우, 네트워크 전송 수를 현저히 줄일 수 있으므로 전반적인 트랜잭션 속도가 향상됩니다.

INFORMIXSHMBASE

INFORMIXSHMBASE 환경 변수는 Informix 인스턴스 공유 메모리 버퍼가 있는 공유 메모리의 기본 주소를 제공합니다. 이 INFORMIXSHMBASE 환경 변수는 `onconfig` 구성 파일의 SHMBASE 매개변수 값을 무시합니다.

INFORMIXSQLHOSTS

INFORMIXSQLHOSTS 환경 변수는 `sqlhosts` 파일의 전체 경로 이름(파일 이름 포함)을 제공합니다. 시스템에서는 기본적으로 $INFORMIXDIR/etc 디렉토리에 있는 `sqlhosts` 파일 이름을 사용합니다.

사용자는 INFORMIXSQLHOSTS 환경 변수를 사용하여 Informix 서버 환경을 제한적으로 볼 수 있습니다. 사용자가 액세스할 수 없는 서버가 하나 이상일 경우, INFORMIXSQLHOSTS 환경 변수를 설정하여 해당 사용자에게 지정된 `sqlhosts` 파일에서 해당 서버를 빼면 IDS 엔진을 통해 이러한 서버에 대한 액세스를 금지할 수 있습니다.

INFORMIXSTACKSIZE

INFORMIXSTACKSIZE 환경 변수는 Informix 서버에 있는 클라이언트 세션의 스택 크기를 지정합니다. 이 INFORMIXSTACKSIZE 환경 변수는 `onconfig` 구성 파일에 있는 `onconfig` STACKSIZE 매개변수를 무시합니다.

INFORMIXTERM

INFORMIXTERM 환경 변수는 Informix 응용프로그램에 사용하는 문자 터미널을 지정합니다. 이 INFORMIXTERM 환경 변수는 사실상 TERM 환경 변수의 설정을 무시합니다.

PDQPRIORITY

PDQPRIORITY 환경 변수는 엔진에 의사 결정 지원 쿼리에 대한 우선 순위 설정을 제공합니다. 우선 순위 설정은 사용자 세션이 지속되는 동안 활성 상태이지만 SQL '`set PDQPRIORITY`'문을 실행하여 무시할 수 있습니다.

또한 SQL문을 사용하여 쿼리의 PDQPRIORITY의 값을 설정할 수 있으며, 이렇게 설정된 값은 해당 환경 변수의 설정을 무시하게 됩니다. PDQPRIORITY의 기본값은 0입니다. PDQPRIORITY 변수의 상한은 `onconfig` 파일에 있는 MAX_PDQPRIORITY 매개변수에 의해 결정됩니다.

PDQPRIORITY는 매우 중요한 설정으로, 엔진에서 쿼리 처리에 할당하는 자원의 양을 제어합니다. (사실상, 우선 순위 설정보다는 좀더 개괄적인 자원 할당 비율입니다.) 이러한 환경 변수의 값은 쿼리에 할당할 처리 스레드 수를 결정하는 데 사용하는 기준 가운데 하나이므로, 실제로 PDQPRIORITY 설정은 처리 중에 쿼리가 수신하는 병렬 처리를 제어합니다. 또한 PDQPRIORITY는 쿼리 처리에 할당된 메모리 양에도 영향을 미칩니다.

엔진 설계자가 의도한 바는, 하나의 매개변수가 여러 엔진 프로세스에 영향을 미칠 수 있는 관리 가능한 구성 매개변수 세트를 사용함으로써 구성 프로세스의 복잡성을 제한하려는 것이었습니다. 이는 별도로 조정될 경우 엔진 처리에 미치는 영향을 제한할

수 있는 매개변수를 여러개 작성하는 것보다는 더 간단한 접근 방식입니다. PDQPRIORITY 매개변수를 사용하는 것이 이러한 방법의 한 예입니다.

PSORT_DBTEMP

PSORT_DBTEMP 환경 변수는 병렬 정렬 작업을 실행하는 동안 임시 파일을 쓰는 디렉토리를 지정합니다.

PSORT_NPROCS

PSORT_NPROCS 환경 변수는 병렬 정렬 작업에 할당할 스레드 수를 지정합니다. 최대값은 10입니다.

TERM

TERM 환경 변수는 Informix 문자 응용프로그램을 포함한 모든 응용프로그램에 사용할 문자 터미널을 나타내도록 설정됩니다. INFORMIXTERM 환경 변수가 설정되면, TERM 환경 변수 설정보다 우선합니다.

TERMCAP

TERMCAP 환경 변수를 사용하여 사용할 문자 터미널 정의가 들어 있는 파일을 나타낼 수 있습니다. Informix의 경우, INFORMIXTERM 환경 변수가 설정되어 있지 않으면 TERMCAP 매개변수에서 참조하는 파일을 사용합니다. INFORMIXTERM 환경 변수가 설정되어 있을 경우, Informix 응용프로그램에서는 INFORMIXTERM 환경 변수 설정이 우선시 됩니다.

onconfig 파일

IDS 구성 파일의 엔트리는 Informix 서버의 작동 방법을 나타냅니다. 구현하는 데 특징적인 면은 다양한 설정을 제공한다는 것입니다. 즉, 설정은 데이터베이스 서버의 목적과 서버를 설치하는 플랫폼에서 사용할 수 있는 자원에 따라 다릅니다. 데이터베이스 서버의 목적은 서버에 액세스하여 서버를 사용하는 응용프로그램에 따라 다릅니다. 어떤 응용프로그램은 다수의 사용자가 특정 레코드 또는 적은 수의 레코드를 요청하는 온라인 트랜잭션 처리 응용프로그램의 트랜잭션 프로파일에 적합할 수 있으며, 어떤 응용프로그램은 상대적으로 적은 수의 사용자가 많은 수의 레코드를 요청하는 의사 결정 지원 응용프로그램의 트랜잭션 프로파일에 적합할 수 있습니다. 엔진을 튜닝하는 방법은 이러한 사용 프로파일에 따라 매우 다릅니다. 이 절에서는 매개변수에 적합한 초기 설정에 대해 설명합니다. 5장에서는 이러한 매개변수를 통해 사용 프로파일에 따라 엔진을 튜닝하는 방법에 대해 논의합니다.

엔진은 기본적으로 `$INFORMIXDIR/etc` 디렉토리에 있는 `onconfig`라는 구성 파일을 검색하며, `onconfig` 파일이 있으면, 이 파일에 지정된 구성 매개변수를 사용합니다. 새 설치에 대해 인스턴스 특정 구성 엔트리를 작성해야 하므로, 기본 구성 파일을 그대로 두고 새 인스턴스에 적합한 새 구성 파일을 작성하는 것이 바람직합니다. 보통은 서버 이름이 포함된 확장자로 새 구성 파일의 이름을 지정합니다. 예를 들어, 서버 이름이 `myserver`이면 구성 파일의 이름은 '`onconfig.myserver`'입니다.

IDS 엔진 인스턴스에 대한 사용자의 관점은 ONCONFIG와 INFORMIXSERVER 환경 변수 두 가지 설정을 모두 사용하여 효과적으로 제어하는 것입니다. INFORMIXSERVER 환경 변수는 엔진에 연결되도록 설정해야 하나, ONCONFIG 환경 변수를 설정하는 것은 엔진의 기본값이 `$INFORMIXSERVER/etc/onconfig` 파일이므로 선택적입니다. 구성 파일에는 Informix 인스턴스 연결에 필요한 모든 매개변수가 들어 있으며 INFORMIXSERVER 환경 변수에는 연결에 사용할 Informix 서버 인스턴스의 이름이 들어 있습니다. 다음 절에서는 IDS 엔진에 사용할 수 있는 구성 매개변수에 대해 설명합니다(표 3-6 참조).

표 3-6 onconfig의 구성 매개변수

매개변수 이름	목적
ROOTNAME	root DBSPACE 이름
ROOTPATH	root DBSPACE가 있는 장치 경로
ROOTSIZE	root DBSPACE의 초기 청크 크기
ROOTOFFSET	root DBSPACE의 루트 장치로 오프셋
MIRROR	시스템에서 DBSPACE 미러링을 사용할 경우 TRUE로 미러 플래그 설정
MIRRORPATH	루트 미러가 있는 장치 경로
MIRROROFFSET	root DBSPACE 미러에 대한 미러 장치로 오프셋
PHYDBS	물리 로그의 DBSPACE 위치
PHYSFILE	킬로바이트 단위의 물리 로그 파일 크기
LOGFILES	논리 로그 파일 수
LTAPESIZE	논리 로그 테이프 출력 장치 크기
LTAPEBLK	논리 로그 테이프 장치의 블록 크기
LOGSIZE	논리 로그 파일 크기
MSGPATH	시스템 메시지 로그 파일의 전체 경로 이름
CONSOLE	시스템 콘솔 장치 경로의 전체 경로 이름
ALARMPROGRAM	경보 프로그램 경로
TAPEDEV	테이프 장치 경로
TAPEBLK	테이프 장치 블록 크기
TAPESIZE	테이프 장치의 최대 데이터량
LTAPEDEV	논리 로그 테이프 장치의 테이프 장치
LTAPESIZE	로그 테이프 장치의 최대 데이터량
STAGEBLOB	온라인 광학의 스테이징 장치
SERVERNUM	서버 인스턴스의 고유 ID
DBSERVERNAME	IDS 인스턴스 이름
DBSERVERALIASES	대체 서버 인스턴스 이름 목록
NETTYPE	네트워크 폴 스레드 구성 정보
DEADLOCK_TIMEOUT	네트워크 트랜잭션의 최대 시간
RESIDENT	공유 메모리의 강제 상주 플래그
MULTIPROCESSOR	다중 CPU 머신 플래그
NUMCPUVPS	엔진 시작 시 CPU VP 수
SINGLE_CPU_VP	단일 CPU VP 플래그

<계속>

표 3-6 onconfig의 구성 매개변수

매개변수 이름	목적
NOAGE	프로세서 에이징 플래그
AFF_SPROC	이 CPU 번호에서 프로세서 친화도 시작
AFF_NPROC	이 CPU 번호에서 프로세서 친화도 종료
LOCKS	시스템의 최대 로크 수
BUFFERS	공유 메모리 버퍼 수
NUMAIOVPS	비동기 입출력 가상 프로세서의 수
PHYSBUFF	킬로바이트 단위의 물리 로그 버퍼 크기
LOGBUFF	논리 로그 버퍼 크기
LOGSMAX	허용된 논리 로그의 최대 수
CLEANERS	버퍼 페이지 클리너 프로세스의 수
SHMBASE	공유 메모리 버퍼 주소
SHMVIRTSIZE	가상 공유 메모리의 초기 공유 메모리 크기
SHMADD	런타임 시 추가된 공유 메모리 세그먼트의 크기
SHMTOTAL	허용된 총 초기 공유 메모리
CKPTINTVL	초 단위의 체크포인트 간격
LRUS	LRU 버퍼 큐의 수
LRU_MAX_DIRTY	버퍼 정리가 중지되는 더티 LRU 큐의 비율
LRU_MIN_DIRTY	버퍼 정리가 시작되는 더티 LRU 큐의 비율
LTXHWM	로그 가득 참 비율의 긴 트랜잭션 하이 워터 마크
LTXEHWM	로그 가득 참 비율의 긴 트랜잭션 배타적 하이 워터 마크
TXTIMEOUT	트랜잭션 시간종료
STACKSIZE	킬로바이트 단위의 프로세스 스택 크기
OFF_RECVRY_THREADS	급속 복구 또는 오프라인 복구 동안의 오프라인 작업자 스레드 수
ON_RECVRY_THREADS	온라인 복구 중에 시작된 온라인 작업자 스레드 수
DRAUTO	자동 데이터 복제 전환 플래그
DRINTERVAL	데이터 복제 버퍼 플러시 간의 최대 시간
DRTIMEOUT	데이터 복제의 네트워크 시간종료

<계속>

표 3-6 onconfig의 구성 매개변수

매개변수 이름	목적
DRLOSTFOUND	데이터 복제의 lost-and-found 디렉토리 경로
BAR_ACT_LOG	onbar 활동 로그 경로 이름
BAR_MAX_BACKUP	실행된 각 onbar 섹션에 대한 최대 병렬 프로세스
BAR_RETRY	onbar가 보관을 재시도한 횟수
BAR_NB_XPORT_COU	데이터 교환에 사용된 데이터베이스 버퍼 수
NT	onbar 전송 버퍼의 크기
BAR_XFER_BUF_SIZE RA_PAGES	전체 테이블을 스캔하는 동안 사용하는 미리 읽기 페이지 수
BAR_BSALIB_PATH	onbar 유틸리티에서 사용하는 스토리지 관리자의 공유 라이브러리 경로
RA_THRESHOLD	미리 읽기 임계값
DBSPACETEMP	임시 파일의 DBSPACE 목록
DUMPDIR	진단 데이터를 이 디렉토리 경로로 덤프
DUMPSHMEM	공유 메모리 덤프 플래그
DUMPGCORE	gcore 공유 메모리 덤프 플래그
DUMPCORE	코어 공유 메모리 덤프 플래그
DUMPCNT	공유 메모리 gcore 덤프 수
FILLFACTOR	인덱스 노드 작성 채우기 요인
USEOSTIME	운영 체제 시간 사용 플래그
MAX_PDQPRIORITY	최대 허용 PDQPRIORITY 설정
DS_MAX_QUERIES	동시 처리 PDQ 쿼리의 최대 수
DS_TOTAL_MEMORY	킬로바이트 단위의 사용할 총 의사 결정 메모리
DS_MAX_SCANS	허용할 최대 의사 결정 지원 스캔 수
DATASKIP	데이터 스킵(DBSPACE 스킵) 기능 플래그
OPTCOMPIND	옵티마이저 힌트 플래그
ONDBSPACEDOWN	DBSPACE 다운 옵션 플래그
LBU_PRESERVE	TRUE로 설정하여 관리 작업에 대한 마지막 논리 로그 저장
OPCACHEMAX	최대 광학 캐시 크기

ROOTAME

ROOTAME은 시스템 예약 페이지와 sysmonitor 데이터베이스를

저장할 때 사용하는 DBSPACE의 고유 이름입니다. 시스템 예약 페이지에는 데이터베이스에 있는 DBSPACE의 목록과 해당 청크의 목록 등의 정보가 들어 있습니다.

사용자 Informix가 DBSPACE를 저장하는 장치나 파일에 대해 읽고 쓸 수 있는 정도의 소유권이 있어야 합니다.

ROOTPATH

ROOTPATH는 root DBSPACE가 있는 파일이나 특정 장치의 전체 경로 이름입니다. 이러한 장치 내에 ROOTNAME 매개변수에서 참조하는 DBSPACE가 있습니다.

ROOTOFFSET

킬로바이트 단위의 root DBSPACE에 대한 오프셋입니다. 어떤 하드웨어 플랫폼에서는 장치에 대한 오프셋을 영(0)으로 사용할 경우(기본값), 장치 정보가 해당 오프셋에 저장되어 작동되지 않습니다. 이러한 플랫폼에서는 이와 같은 예외적인 사항을 수용할 수 있도록 이 매개변수를 영(0)이 아닌 값으로 설정해야 합니다.

ROOTSIZE

ROOTSIZE 매개변수를 사용하여 root DBSPACE의 크기를 설정합니다. 이 값은 시스템을 초기화하는 동안에만 읽혀집니다. 데이터베이스 인스턴스가 초기화되었으면, ROOTSIZE 매개변수는 무시됩니다. 그러면 ROOTSIZE 매개변수를 변경하여 DBSPACE 크기를 증가하는 것이 아니라 DBSPACE에 청크를 추가하여 DBSPACE 크기를 증가해야 합니다.

MIRROR

DBSPACE와 함께 Informix 데이터베이스 미러링을 사용하려면 이 매개변수를 TURE로 설정해야 합니다. 이렇게 하면 특정 DBSP

ACE에 대한 미러가 생성되지 않으며, 데이터베이스 인스턴스에서만 미러링을 사용할 수 있습니다.

MIRRORPATH

MIRRORPATH 매개변수를 설정하여 root DBSPACE를 미러링할 때 사용하는 장치에 대한 경로를 나타냅니다. MIRRORPATH 매개변수가 설정되어 있으면 root DBSPACE를 미러링합니다. 이 매개변수가 설정되어 있지 않으면, root DBSPACE는 미러링되지 않습니다.

tip *root DBSPACE를 미러링하는 것은 어느 경우에나 좋습니다. root DBSPACE의 정보가 갖는 중요성으로 인해 root DBSPACE가 실패하면, 엔진이 액세스하는 것이 불가능하게 되어 엔진을 시작할 수 없게 됩니다. 이 경우에 선택할 수 있는 유일한 옵션은 전체 복구(엔진 오프라인)입니다.*

또한 root DBSPACE에서 하드웨어가 고장 날 경우, root DBSPACE 미러링을 통해 투명한 장애 복구를 할 수 있습니다. 엔진이 root DBSPACE에서 입출력 작업 시 오류가 발생하면, 간단하게 미러 root DBSPACE로 전환하여 그곳에서 입출력 작업을 수행하면 됩니다.

MIRROROFFSET

MIRROROFFSET 매개변수는 MIRRORPATH 매개변수에서 참조한 청크의 미러 장치에 대한 오프셋을 나타냅니다. 이 청크는 root DBSPACE의 미러 역할을 합니다.

PHYDBS

PHYDBS 매개변수는 IDS 인스턴스에 대한 물리 로그를 저장하는데 사용되는 DBSPACE의 이름을 지정합니다.

PHYSFILE

PHYSFILE 매개변수는 물리 로그 파일의 크기를 킬로바이트로 단위를 표시합니다.

물리 로그 파일의 크기를 적절하게 지정하는 것이 중요합니다. 물리 로그 파일이 채워지면(물리 로그 오버플로라고 함), 체크포인트가 트리거되며 체크포인트가 과도할 경우 시스템 성능이 저하될 수 있습니다. 5장 "IDS 엔진 모니터링 및 튜닝"에서 이 문제에 대해 자세히 다룹니다.

LOGFILES

LOGFILES 매개변수는 논리 `log files`의 수를 지정합니다. 각 로그 파일의 크기는 LOGSIZE 매개변수로 설정됩니다. 이 매개변수는 시스템을 초기화 동안에만 읽혀집니다. 로그 파일 수를 추가하려면, `onparams` 명령이나 `dbmonitor` 또는 기타 적절한 유틸리티를 사용하십시오.

적절한 수의 논리 로그 파일로 엔진을 구성하는 것이 중요합니다. 로그 파일이 너무 적으면, 데이터베이스에서 사용 가능한 로그 파일 수를 초과하는 트랜잭션을 실행할 수도 있습니다. 이러한 상황이 발생하면, 데이터베이스는 트랜잭션을 롤백하려고 합니다. 트랜잭션을 롤백하기에 충분한 로그 파일이 있으면 전체 시스템에 미치는 영향은 최소화됩니다. 그러나 트랜잭션을 롤백하는 데 로그 파일이 충분하지 않을 경우, IDS 엔진이 잠재적으로 로크될 수 있습니다.

LOGSIZE

LOGSIZE 매개변수는 생성된 각 논리 로그 파일의 크기를 설정할 때 사용합니다. 데이터베이스 인스턴스를 시작할 때 이 LOGSIZE 매개변수를 사용합니다. IDS 인스턴스의 초기화 이후 새 값이 설정된 경우, 엔진이 초기화된 후에 생성된 로그 파일은 새 논리 로그 크기를 사용할 수 있습니다.

논리 로그 파일의 크기는 여러 가지 요인에 따라 달라집니다. 데이터베이스 업데이트로 인해 가장 많은 로깅 활동이 생성되므로, 상대적으로 정적인(즉, 업데이트 수가 극히 적은) 데이터베이스에는 로그된 활동사항이 많지 않습니다. 업데이트 활동이 좀더 많은 시스템의 경우, 초기에 다음 공식을 사용하여 논리 로그를 설정해야 합니다.

```
LOGSIZE = (users * maxrows) * 512
```

여기서 users는 시스템에서 예상되는 최대 동시 처리 사용자 수이고 maxrows는 이러한 사용자가 트랜잭션에서 한 번에 액세스할 때 예상되는 최대 행 수입니다.

각 IDS 인스턴스마다 트랜잭션 프로파일이 아주 다양하기 때문에 시스템 로그 활동이 정확히 어떤 것인지를 예측하기란 쉽지 않습니다. 데이터베이스 관리자는 시스템 활동을 모니터하고, 사용할 수 있는 논리 로그의 많은 부분(예: 각각 1000byte인 100,000 행의 데이터 로드)을 이용하는 업데이트 활동을 감시해야 합니다. 이러한 업데이트 활동을 처리할 수 있도록 논리 로그 번호 및 크기를 튜닝하거나 또는 응용프로그램 자체를 변경하여 논리 로그의 오작동을 줄여야 합니다. 응용프로그램은 더 많은 데이터베이스 완료(데이터베이스에 쓰여진 행을 완료하고 논리 로그를 사용 가능한 상태로 하는 작업)를 수행하도록 다시 작성할 수 있습니다.

MSGPATH

MSGPATH 매개변수를 사용하여 시스템 메시지 파일의 전체 경로 이름을 나타낼 수 있습니다. 시스템 메시지 파일은 데이터베이스 서버가 생성한 메시지를 저장합니다.

CONSOLE

CONSOLE 매개변수를 사용하여 엔진 메시지 출력 장치를 표시합니다. 엔진 메시지 출력 장치는 터미널 장치일 수도 있고 ASCII 파일의 전체 경로 이름일 수도 있습니다. 엔진 상태 메시지(로그 완료,

체크포인트 완료 등)를 이 출력 장치로 전송합니다.

ALARMPROGRAM

ALARMPROGRAM 매개변수는 경보 이벤트가 실행하는 경보 프로그램의 전체 경로 이름을 저장합니다. 경보가 트리거되면 이 프로그램이 실행되어, 경보의 특성을 나타내는 매개변수가 전달됩니다.

Informix에서는 경보 프로그램을 제공하지 않으므로, 관리자가 이를 IDS 엔진에서 사용할 수 있도록 해야 합니다.

TAPEDEV

TAPEDEV 매개변수는 백업에 사용할 테이프 장치의 경로를 나타냅니다. Informix ontape 백업 유틸리티는 이 장치를 백업의 출력 장치로 사용합니다.

onbar 유틸리티는 출력에 TAPEDEV 매개변수 값을 사용하지 않고, 고유의 구성 파일에 있는 값을 사용합니다.
논리 로그는 이 장치에 대한 출력이 아니라 LTAPEDEV 매개변수에 지정된 장치에 대한 출력입니다.

TAPEBLK

TAPEBLK 매개변수에는 TAPEDEV 매개변수가 참조하는 테이프 장치의 블록 크기가 들어 있습니다. 이 매개변수 값은 ontape 백업 유틸리티에서 사용합니다.

TAPESIZE

TAPESIZE 매개변수에는 블록으로 표시된 테이프 장치의 최대 크기가 들어 있습니다. 이 매개변수 값은 ontape 백업 유틸리티에서 사용합니다.

TAPESIZE 매개변수를 너무 낮게 설정하여 테이프 출력 프로세스에서 파일의 끝(EOF) 오류가 발생하면, 테이프 출력 프로세스가 중지됩니다. 값을 올바로 설정하면 ontape 프로그램이 중지되고, TAPESIZE 바이트가 테이프 장치에 쓰여졌을 때 다른 테이프를 입력하라는 프롬프트가 표시됩니다.

LTAPEDEV

LTAPEDEV 매개변수에는 로그 테이프 출력에 사용되는 장치의 전체 경로 이름이 들어 있습니다.

"인접하는 로깅" 로그 옵션을 사용하면, 해당 장치가 백업에 사용되는 TAPEDEV 장치와는 다른 장치여야 합니다.

또한, 디스크에 논리 로그 파일이 가득 차게 되면, 논리 로그 파일을 테이프 장치에 씁니다. 논리 로그 파일을 테이프 장치에 쓰면, 디스크의 논리 로그 파일은 사용할 수 있는 상태가 됩니다. 로그 파일에 있는 모든 로그 레코드가 닫히면, 로그 파일을 테이프에 쓰도록 스케줄링할 수 있습니다.

로그 트랜잭션이 여러 로그 파일에 걸쳐 있기 때문에 로그 파일에 완료되지 않은 로그 레코드가 들어 있는 경우, 로그 파일에 걸쳐 있는 논리 로그 트랜잭션이 완료될 때까지 로그 파일을 테이프에 쓸 수 없다는 점에 유의하십시오.

LTAPEBLK

LTAPEBLK 매개변수에는 LTAPEDEV 매개변수에 의해 지정된 논리 로그 테이프 장치의 블록 크기가 들어 있습니다. 논리 로그 파일을 테이프에 쓰는 내부 스레드에서 이 장치를 사용합니다.

LTAPESIZE

LTAPESIZE 매개변수는 논리 로그 테이프 장치에 쓰는 테이프의 최대 크기(킬로바이트 단위)를 저장합니다. 논리 로그 출력 스레드가 논리 로그를 최대로 쓰고 나면, 콘솔 장치에 새로운 로그 테이프를 입력하라는 프롬프트가 표시됩니다.

STAGEBLOB

STAGEBLOB은 온라인 광학 스테이징 영역으로 사용할 BLOB 장치를 지정합니다. 온라인 광학 스테이징 영역은 광학 데이터를 광학 스토리지 장치에 쓰기 전에 저장하는 영역입니다.

SERVERNUM

SERVERNUM 매개변수를 사용하여 Informix 서버 인스턴스와 관련된 고유 번호를 선택합니다. 동일한 머신에서 실행되는 IDS 인스턴스가 둘 이상인 경우, 각 SERVERNUM은 Informix에 할당된 공유 메모리 내의 상대 위치 또는 오프셋을 나타냅니다. 이 SERVERNUM 값은 로컬 호스트 컴퓨터 내의 각 인스턴스마다 고유해야 하지만, 네트워크에서는 고유할 필요가 없습니다.

DBSERVERNAME

DBSERVERNAME 매개변수는 데이터베이스 인스턴스에 할당된 고유 이름입니다. 이 엔트리는 `sqlhosts` 파일에 해당 프로토콜과 일치하는 엔트리가 있어야 합니다. 데이터베이스 서버에 연결하는 데 사용하는 프로토콜이 둘 이상이면, `sqlhosts` 파일에 추가 프로토콜과 서버 이름을 입력해야 하며, DBSERVERALIASES 구성 매개변수 목록에도 지정된 추가 서버 이름이 들어 있어야 합니다.

DBSERVERALIASES

DBSERVERALIASES를 통해 데이터베이스 인스턴스에 하나 이상의 대체 이름을 제공합니다. 이러한 대체 이름을 사용하여 엔진 연결에 필요한 추가 연결 프로토콜(예: TCP/IP)을 지정합니다. `sqlhosts` 파일에 이러한 서버의 이름에 해당하는 엔트리와 프로토콜이 있어야 합니다.

NETTYPE

NETTYPE 매개변수 엔트리는 Informix 서버에 연결할 때 사용하는 연결 프로토콜에 대한 정보를 제공합니다. Informix 엔진 연결에 사용되는 각 연결 프로토콜에는 해당 연결 유형을 리스닝하는 폴 스레드 세트가 있습니다.

`sqlhosts` 파일에는 NETTYPE 엔트리에 해당하는 엔트리가 있어야 합니다. `sqlhosts` 파일의 엔트리는 폴 스레드가 연결을 리스닝하는 위치 및 방법을 지정합니다. 예를 들어, 폴 스레드가 시작되어 TCP/IP 연결을 리스닝하면, `sqlhosts` 파일에 있는 엔트리는 스레드가 TCP/IP 연결을 리스닝하는 네트워크 서비스(TCP/IP 포트와 관련)를 지정합니다.

NETTYPE 구성 엔트리는 네트워크 연결 유형, 메모리 연결 관리에 지정된 스레드 수, 해당 네트워크 유형의 동시 연결 수 및 해당 연결에 대한 폴 스레드를 실행하는 가상 프로세서 클래스에 관한 정보를 제공합니다.

NETTYPE 매개변수는 서버가 사용하는 각 프로토콜 유형에 대해 이러한 정보를 지정합니다. 다음은 엔트리에서 사용하는 포맷입니다.

```
NETTYPE protocol, poll threads, connections, VP class
```

이러한 구성 파일 엔트리에서, NETTYPE는 지정된 구성 매개변수를 나타내고, protocol은 지정된 연결 프로토콜이며, poll threads는 연결 프로토콜을 관리하기 위해 시작된 폴 스레드 수이고, connections는 연결 프로토콜에 허용된 최대 연결 수이며, VP class는 이러한 폴 스레드가 실행되는 가상 프로세서 클래스입니다.

특정 프로토콜의 연결 수가 최대에 이르면, 데이터베이스 엔진에서는 해당 프로토콜에 대한 추가 연결을 허용하지 않습니다.

DEADLOCK_TIMEOUT

DEADLOCK_TIMEOUT 매개변수는 시스템이 원격 로크를 대기하는 시간을 초로 설정할 때 사용합니다. 이는 분산 트랜잭션에만

적용되며, 시스템에서 사용하는 다른 로크에는 적용되지 않습니다.

RESIDENT

RESIDENT 매개변수는 TRUE 또는 FALSE, 두 가지 설정이 있는 플래그를 나타냅니다. 이 플래그를 FALSE(또는 0)로 설정하면 Informix의 공유 메모리 부분이 디스크로 스왑됩니다. TRUE로 설정하면 강제 상주가 시행되고 운영 체제는 공유 메모리를 디스크로 스왑할 수 없게 됩니다.

일반적으로 RESIDENT를 TRUE로 설정하는 것이 최적의 설정으로 간주됩니다. 디스크 사이에서 메모리를 스왑하는 작업은 시스템 성능에 심각한 영향을 줄 수 있는 시간이 많이 드는 작업일 수 있습니다.

MULTIPROCESSOR

MULTIPROCESSOR 매개변수를 사용하여 머신 플랫폼이 멀티프로세서(SMP-대칭 멀티프로세서) 머신인지 여부를 나타냅니다. 멀티프로세서 머신인 경우, TRUE인 1로 설정하고 멀티프로세서 머신이 아닌 경우, FALSE인 0으로 설정합니다.

엔진이 수행해야 하는 작업에 처리를 지정할 때 내부적으로 이 코드를 사용합니다. IDS 엔진이 사용할 수 있는 CPU가 하나 밖에 없다는 것을 인식하면, 작업을 다르게 관리합니다.

NUMCPUVPS

NUMCPUVPS는 엔진이 시작할 때 생성할 엔진에 대한 CPU 클래스의 가상 프로세서의 수로 설정됩니다. 이 값은 onmode 유틸리티를 사용하여 런타임 시 동적으로 변경할 수 있습니다.

SINGLE_CPU_VP

엔진이 데이터베이스 인스턴스에 SINGLE_CPU_VP를 할당해야 할 경우, SINGLE_CPU_VP 매개변수는 TRUE인 1로 설정됩니다.

NOAGE

NOAGE 매개변수가 1로 설정되어 TRUE가 되면, 엔진에서는 운영 체제가 운영체제의 가상 프로세서를 나타내는 프로세스를 에이징할 수 없습니다. Unix 운영 체제는 기본적으로 오래 실행되는 작업의 프로세스 우선 순위를 에이징하거나 낮춥니다. 이러한 매개변수가 TURE로 설정되면, 엔진에서는 프로세스의 에이징을 허용하지 않으려고 합니다.

AFF_SPROC

AFF_SPROC 매개변수를 통해 멀티프로세서 플랫폼에서 프로세서 친화도를 사용할 수 있습니다. 프로세서 친화도는 멀티프로세서 머신에서 특정 프로세스가 CPU에 바인드될 수 있도록 하는 운영 체제 기능입니다. 이 기능을 사용할 수 없는 운영 체제도 있으며, 이 경우 엔진에서 AFF_SPROC 매개변수 설정을 무시합니다.

AFF_SPROC 매개변수는 프로세서 친화도를 위해 시작 CPU에 설정됩니다.

프로세서 친화도가 있으면 운영 체제의 한 CPU에서 다른 CPU로 실행 중인 프로세스를 이관하지 않아도 됩니다. CPU의 프로세서 친화도 기능을 통해 하나의 프로세스가 CPU에 바인드됩니다. 이론적으로는 이러한 과정을 통해 프로세스가 CPU 사이클에 더 많이 액세스할 수 있지만, 실제적으로는 이러한 매개변수를 변경한다고 해도 성능에 별다른 영향을 주지 못하는 경우도 있습니다. 대부분의 SMP Unix 버전은 CPU 간에 프로세스를 이관하고 효과적으로 CPU 사이클을 할당하는 작업을 잘 수행하므로, 결과적으로 이러한 매개변수를 설정할 필요가 없습니다.

AFF_NPROC

AFF_NPROC 매개변수를 사용하여 프로세서 친화도에 대한 종료 프로세서를 나타냅니다. 예를 들어, CPU가 여섯 개인 시스템에서 프로세서 친화도를 다섯 개의 CPU에 할당하면(나머지 CPU 하나는 운영 체제에 둠), AFF_SPROC가 CPU 1에 대해 0으로 설정되고 AFF_NPROC는 CPU 4에 대해 4로 설정됩니다.

LOCKS

LOCKS 매개변수를 사용하여 시스템에 사용되는 로크 수를 설정합니다. IDS는 로크를 사용하여 특정 작업을 하는 동안 행을 로크하고 페이지에 버퍼를 지정합니다. 쿼리를 수행하는 동안 IDS 엔진에 로크 수가 부족해지면, 쿼리는 실패하고 엔진 작업은 계속됩니다. 로크는 44byte의 공간을 소비하며 따라서 적절한 수의 로크를 구성하면 시스템에서 상당한 양의 메모리 자원이 제거되지 않습니다.

BUFFERS

BUFFERS 매개변수를 사용하여 시스템에 할당된 공유 메모리 버퍼의 수를 설정합니다. 이 매개변수는 공유 메모리 버퍼에 할당된 바이트를 식별합니다.

할당된 BUFFERS 수는 실행되는 응용프로그램과 시스템에서 사용할 수 있는 메모리 자원의 양에 따라 다릅니다. 일반적으로 할당된 버퍼의 수는 사용할 수 있는 시스템 메모리의 20-25%이어야 합니다.

NUMAIOVPS

NUMAIOVPS 매개변수를 사용하여 엔진에 대한 비동기 입출력 수행에 할당되는 AIO VP(가상 프로세서)의 수를 설정합니다.

PHYSBUFF

PHYSBUFF 매개변수는 시스템의 두 개의 물리 로그 버퍼 각각에 할당되는 킬로바이트 수를 나타냅니다. 메모리의 이 위치에서 물리 로그 페이지에 버퍼를 지정한 후 디스크의 물리 로그에 쓰여집니다. 두 개의 버퍼를 사용하여, 하나의 버퍼에 계속 쓰면서 다른 로그 버퍼를 디스크에 씁니다. 디스크의 물리 로그에 쓰여집니다.

LOGBUFF

LOGBUFF 매개변수는 IDS가 사용하는 세 개의 논리 로그 버퍼 각각에 할당할 킬로바이트 수를 나타냅니다. 버퍼 지정 로깅 옵션을 사용하면, 논리 로그 페이지는 메모리의 이 위치에서 버퍼가 지정된 후에 디스크의 논리 로그 페이지에 쓰여지게 됩니다.

LOGSMAX

LOGSMAX 매개변수는 시스템에 추가할 수 있는 논리 로그의 최대 수를 나타냅니다. 논리 로그를 추가하려는 시도로 인해 논리 로그의 총 수가 이 최대 수를 초과하게 되면, 오류가 리턴되고 논리 로그 추가 프로세스가 실패합니다. 데이터베이스 인스턴스를 초기화한 후에는 논리 로그의 최대 수를 변경할 수 있지만, 엔진을 다시 시작해야 값을 인식할 수 있습니다.

CLEANERS

CLEANERS 매개변수는 LRU 큐를 관리하기 위해 할당된 페이지 클리너 스레드의 수를 나타냅니다. 페이지 클리너 스레드를 사용하여 업데이트된(더티) 버퍼 페이지를 디스크에 씁니다.

SHMBASE

SHMBASE 매개변수는 Informix 엔진에서 사용하는 공유 메모리 버퍼의 기본 또는 시작 주소를 나타내는 16진값입니다. 어떤 시스템에서는 기본 SHMBASE 주소가 다른 응용프로그램에 사용되므로 Informix 엔진은 다른 기본 주소를 사용하도록 지정되어야 합니다. SHMBASE 매개변수는 INFORMIXSHMBASE 환경 변수에 의해 무시됩니다.

SHMVIRTSIZE

SHMVIRTSIZE 매개변수를 사용하여 가상 공유 메모리 세그먼트의

초기 크기를 설정합니다. 가상 공유 메모리는 고정 오버헤드, 공유 구조 및 전용 구조에 사용되는 공유 메모리 부분으로 정의됩니다. 이러한 매개변수를 추정하는 공식은 다음과 같습니다.

```
fixed overhead = shared structures + (max concurrent sessions
                                      * private structures)
```

이러한 값의 정의는 다음과 같습니다.

고정 오버헤드	전역 풀 + 스레드 풀
공유 구조	AIO 벡터 + 정렬 메모리 + DBSPACE 백업 버퍼 + 디렉토리 크기 + 내장 프로시저 캐시 크기 + 히스토그램 풀 + 기타 풀
전용 구조	세션 스택 + 세션 힙 + 세션 제어 블록 구조

SHMADD

SHMADD 매개변수를 사용하여 DSS 쿼리를 처리하는 동안 필요한 경우, 시스템에 추가되는 공유 메모리 세그먼트의 값을 설정합니다. 이 매개변수 값이 0이 아니면, 메모리는 SHMTOTAL 매개변수 값까지 또는 운영 체제에서 더 이상 서버에 추가 메모리를 제공하지 않는 시점까지 추가됩니다.

tip *메모리를 큰 분할 영역에 추가하면 좀 더 효과적으로 추가할 수 있으므로, SHMADD를 높은 수이지만 관리 가능한 수로 설정하는 것이 최적입니다. 이러한 방식의 단점은 메모리가 쿼리에 할당되기 때문에 더 이상 필요하지 않은 경우에도 IDS 엔진에서 사용 가능해야 한다는 점입니다.*

SHMTOTAL

SHMTOTAL 매개변수는 엔진에서 할당하는 총 DSS 메모리 양으로 설정됩니다. 이 값이 0이면, 엔진에서는 운영 체제가 추가 메모리의 할당을 금지할 때까지 메모리를 할당하려고 합니다.

CKPTINTVL

CKPTINTVL 매개변수를 사용하여 Informix 서버 체크포인트의 간격을 초 단위로 설정합니다. 버퍼 캐시가 디스크와 동기화되는 동안 체크포인트에서 대부분의 시스템 활동을 중지시킵니다. 즉, 과도한 체크포인트가 시스템 성능에 영향을 줄 수 있으므로 CKPTINTVL은 매우 중요한 매개변수입니다. 그러나 체크포인트 간격을 너무 높게 설정하면 체크포인트를 실행하는 데 오랜 시간이 소요되므로 시스템의 처리량에 영향을 줄 수 있습니다.

LRUS

LRUS 매개변수를 사용하여 엔진에 대한 LRU 큐 수를 설정합니다. LRU를 사용하여 시스템 버퍼의 업데이트된(더티) 페이지를 추적합니다. 업데이트된 페이지는 어떤 시점에서 디스크에 쓰여져야 하는 페이지입니다. 업데이트 집약적 시스템의 성능에 상당한 영향을 줄 수 있으므로 이러한 페이지를 언제 디스크에 써야 하는지에 대한 의사 결정은 중요합니다. Informix 서버는 LRUS 매개변수를 LRU_MAX_DIRTY 및 LRU_MIN_DIRTY 매개변수와 함께 사용하여 페이지를 디스크에 플러시하거나 쓰는 시기 및 방법(LRU 버퍼 정리라고도 함)을 결정합니다. LRU 큐 튜닝에 대해서는 5장에서 자세히 설명합니다.

LRU_MAX_DIRTY

LRU_MAX_DIRTY 매개변수는 클리너 스레드가 웨이크업되어 LRU 큐를 정리하기 전에 LRU 큐가 소유할 수 있는 최대 페이지 수를 나타냅니다. LRU 큐는 LRU_MIN_DIRTY에 지정된 비율까지 정리됩니다.

LRU_MIN_DIRTY

LRU_MIN_DIRTY 매개변수는 큐에 있는 총 버퍼 수의 비율로, LRU 큐에 남겨진 최소 더티 페이지를 지정합니다. 페이지 클리너가 웨이크업되어 큐를 정리할 때, LRU_MAX_DIRTY 비율에서 정리를

시작하며, 큐가 LRU_MIN_DIRTY 비율 더티 페이지에 도달할 때까지 정리(업데이트된 페이지를 디스크에 쓰기)를 계속합니다.

LTXHWM

LTXHWM 매개변수는 긴 트랜잭션 하이 워터 마크를 나타내는데 사용합니다. 긴 트랜잭션 하이 워터 마트(LTXHWM)는 Informix 서버에 할당된 총 로그 수의 비율입니다. 채워진 로그의 수가 이 비율에 도달하면, 논리 로그 롤백을 수행하도록 스레드가 지정됩니다.

긴 트랜잭션은, 다중 트랜잭션 로그에 걸쳐 있는 하나의 트랜잭션이 트랜잭션을 완료하는 데 로그가 충분하지 않아 롤백을 해야 할 경우에 발생합니다. 롤백 완료에 필요한 로그가 충분하지 않아 데이터베이스 인스턴스를 사실상 동결하게 되는 상황을 피하려면, LTXHWM 매개변수를 적절한 비율로 설정하여 롤백을 완료할 수 있도록 로그 파일에 사용할 수 있는 논리 로그를 충분히 남겨 놓아야 합니다.

LTXEHWM

LTXEHWM 매개변수를 사용하여 긴 트랜잭션 배타적 하이 워터 마크를 나타냅니다. 이 값은 Informix 서버에 할당된 총 로그 수의 비율입니다. 채워진 논리 로그 수가 이 비율에 도달하면, 논리 로그 롤백이 시작되고 롤백을 수행하는 스레드에 논리 로그에 대한 배타적 액세스가 제공됩니다.

TXTIMEOUT

TXTIMEOUT 매개변수는 분산된 트랜잭션에 대한 트랜잭션 시간종료(초 단위)를 나타냅니다. 분산 트랜잭션에 관련된 서버가 원격 서버에 조치를 수행하라고 요청할 때, TXTIMEOUT에 지정된 시간(초)만큼 응답을 기다립니다. 지정된 시간(초)이 지난 후에도 응답을 받지 못하면 트랜잭션이 중단됩니다.

TXTIMEOUT은 IDS 다중 인스턴스 간의 트랜잭션(인스턴스가 다른 머신에 상주하는지 여부는 관계없음)인 분산 트랜잭션에만 적용됩니다. TXTIMEOUT은 Informix 인스턴스에 로컬인 트랜잭션에는 적용되지 않습니다.

STACKSIZE

STACKSIZE 매개변수는 IDS 스레드의 스택 크기(킬로바이트 단위)로 설정됩니다. 이 값은 상한이 없습니다. 일반적으로 스택은 32 KB로 설정되지만 내장 프로시저에서 수행하는 작업과 같은 반복적 작업에는 더 많은 스택 공간이 필요합니다. 사용할 수 있는 스택 공간이 없으면, Informix 엔진을 사용하여 스택을 증가시킬 수 있습니다. 많은 양의 스택 공간이 필요한 반복적 또는 기타 작업이 많을 경우, STACKSIZE 매개변수를 32KB가 넘는 숫자로 설정하여 런타임 시 추가 스택 공간을 할당하는 오버헤드를 피하도록 하십시오.

OFF_RECVRY_THREADS

OFF_RECVRY_THREADS 매개변수를 사용하여 데이터베이스의 논리 복구를 수행하기 위해 할당하는 오프라인 복구 스레드 수를 나타냅니다. 서버가 단계적으로 종료되지 않았다고(이는 물리 로그의 페이지를 보고 알 수 있음) Informix 서버가 판별하면, OFF_RECVRY_THREADS 매개변수에 지정된 오프라인 복구 스레드 수를 시작하여 서버를 다시 일관성 있는 상태로 가져와 온라인 상태가 되게 합니다.

ON_RECVRY_THREADS

ON_RECVRY_THREADS 매개변수는 엔진이 온라인 복구(선택 복구) 중에 시작되는 온라인 복구 스레드의 수로 설정됩니다. 온라인에서 필요한 로그 롤포워드를 복구할 경우, 온라인 복구 스레드가 필요합니다.

DRAUTO

DRAUTO는 0,1 또는 2의 정수값을 지정하여 데이터베이스의 복제 인스턴스가 실패할 경우 사용하는 데이터 복제 전환 유형을 나타냅니다. 표 3-7에는 DRAUTO의 값에 대한 설명이 나와 있습니다.

표 3-7 DRAUTO 값

값	데이터 복제 유형
0	수동
1	보유
2	무효

수동 전환은 데이터 복제 실패 시 엔진에서 자동으로 전환되지 않고 수동으로 전환을 해야 한다는 것을 나타냅니다.

보유 매개변수는 데이터 복제 실패 시 자동으로 IDS 인스턴스가 보조 인스턴스를 표준 모드로 전환한 후 데이터 복제 재시작 시 자동으로 기본 인스턴스를 보조 인스턴스로 전환한다는 것을 나타냅니다.

무효 매개변수는 데이터 복제 실패 시 자동으로 엔진이 보조 인스턴스를 표준으로 전환한 후 복제 재시작 시 자동으로 원래의 기본 인스턴스를 보조 인스턴스로 전환한다는 것을 나타냅니다.

DRINTERVAL

DRINTERVAL 매개변수를 설정하여 Informix 서버의 복제된 두 인스턴스 간 버퍼 플러시 사이의 간격을 초 단위로 나타냅니다. 엔진은 설정된 이 시간(초)만큼 기다렸다가 하나의 데이터베이스 인스턴스에서 다른 인스턴스로 버퍼 플러시를 수행합니다.

DRTIMEOUT

DRTIMEOUT 매개변수는 복제된 인스턴스간의 네트워크 시간종료를 초 단위로 나타냅니다. 이 수는 엔진이 복제된 인스턴스를 더 이상 사용할 수 없다고 판별하기 전에 복제된 인스턴스로부터 응답을 기다리는 시간(초)입니다.

DRLOSTANDFOUND

DRLOSTANDFOUND 매개변수는 특정 유형의 데이터 복제 상황에서 상실될 수도 있는 긴 트랜잭션의 스토리지 디렉토리 경로로 설

정됩니다. 상실된 트랜잭션에는 기본 데이터베이스에서 완료되었으나 보조 서버에서 완료되지 않은 트랜잭션이 들어 있습니다.

CDR_LOGBUFFERS

CDR_LOGBUFFERS 매개변수를 사용하여 엔터프라이즈 복제의 로그 버퍼에 사용할 메모리를 식별합니다.

CDR_EVALTHREADS

CDR_EVALTHREADS 매개변수를 사용하여 엔터프라이즈 복제에 할당할 평가 스레드 수를 설정합니다.

CDR_DSLOCKWAIT

CDR_DSLOCKWAIT 매개변수는 엔터프라이즈 복제가 로크를 기다리는 시간(초)을 나타냅니다.

CDR_QUEMEM

CDR_QUEMEM은 엔터프라이즈 복제에서 송신/수신 큐에 사용하는 최대 메모리 양을 나타냅니다.

BAR_ACT_LOG

BAR_ACT_LOG 매개변수는 onbar 활동 로그의 위치를 지정합니다. onbar 백업 및 복구 유틸리티가 메시지를 이 로그 파일에 씁니다.

BAR_MAX_BACKUP

BAR_MAX_BACKUP 매개변수를 사용하여 실행되고 있는 각 onbar 명령에 허용되는 최대 병렬 프로세스 수를 설정합니다. 전체

시스템 백업이나 복구의 경우 이 매개변수는 무시됩니다.

BAR_RETRY

BAR_RETRY 매개변수는 onbar가 백업 또는 복구 작업을 재시도하는 횟수를 나타냅니다. 가능한 설정은 표 3-8에 나와 있습니다.

표 3-8 BAR_RETRY 설정

설정	설명
BAR_ABORT	오류가 발생하면 백업이 중지되고 오류 메시지가 리턴됩니다.
BAR_CONT	데이터베이스 객체에 오류가 발생하면, 오류 메시지가 리턴되고 해당 백업이 중지되지만, 남은 다른 데이터베이스 객체는 보관 또는 복구됩니다.
(n)	특정 수로 설정되면, 이 수만큼 재시도를 한 후에 해당 데이터베이스 객체의 백업을 중지하고 다음 객체로 이동합니다.

논리 로그 파일 보관에 실패할 경우, onbar 유틸리티가 BAR_RETRY 설정과 관계없이 이러한 시도를 중지시킵니다.

BAR_NB_XPORT_COUNT

BAR_NB_XPORT_COUNT 매개변수는 각 onbar 프로세스에서 데이터를 엔진과 교환할 때 사용하는 데이터 버퍼의 수를 지정합니다. 이 값은 더 많은 버퍼가 있는 onbar의 성능에 영향을 주어, 보관/복구 프로세스의 처리 속도를 향상시킬 수 있습니다.

BAR_XFER_BUF_SIZE

BAR_XFER_BUF_SIZE 매개변수는 페이지로 표시되는 전송 버퍼의 크기를 지정합니다. 전송 버퍼를 사용하여 엔진과 onbar 유틸리티 사이에서 데이터를 이동합니다.

BAR_BSALIB_PATH

BAR_BSALIB_PATH 매개변수는 onbar 유틸리티가 사용하는 스토리지 관리자의 라이브러리 경로 이름을 나타냅니다. 기본 경로 이름은 /user/lib/libsad001.xx이고, 여기서 xx는 공유 라이브러리 파일 이름입니다.

RA_PAGES

RA_PAGES 매개변수는 엔진에서 테이블을 스캔할 때 사용하는 미리 읽기 페이지의 수로 설정됩니다. 이 RA_PAGES 기능은 엔진에서 테이블이나 인덱스를 스캔하거나 많은 수의 인접하는 페이지를 읽을 경우에만 구현됩니다.

RA_THRESHOLD

RA_THRESHOLD는 IDS 미리 읽기의 미리 읽기 페이지 수로 설정됩니다. 이는 엔진이 다음 미리 읽기 작업을 수행하기 위해 신호를 보내는, 버퍼 페이지에 있는 처리되지 않은 페이지의 수를 지정합니다.

DBSPACETEMP

DBSPACETEMP 매개변수는 임시 스토리지에 사용되는 DBSPACE 목록을 지정합니다. DBSPACE 목록은 세미콜론이나 쉼표로 구분합니다. 이 목록의 DBSPACE는 라운드 로빈 방식으로 목록에 표시된 순서대로 사용됩니다.

DUMPDIR

DUMPDIR 매개변수는 시스템이 예기치 않게 종료되어야 할 경우에 엔진 진단을 덤프하는 디렉토리로 설정됩니다.

DUMPSHMEM

DUMPSHMEM 매개변수를 설정하여 시스템 고장 시 공유 메모리를 파일에 써야 하는지(덤프) 여부를 나타냅니다. 공유 메모리를 덤프하지 말아야 하는 경우, DUMPSHMEM 매개변수는 0으로 설정되며 공유 메모리를 덤프해야 하는 경우에는 1로 설정됩니다.

공유 메모리 덤핑은 특정 시점에서 엔진에 문제가 발생한 경우 디버깅을 할 때 유용하기도 합니다. 이러한 상황이 발생했을 때 시스템 고장 시 공유 메모리를 덤핑한 후 나중에 onstat 명령을 사용하여 분석할 수 있습니다.

DUMPGCORE

DUMPGCORE 매개변수는 가정 오류 발생 시 엔진에서 gcore 유틸리티를 사용하여 코어 이미지를 쓸(덤프할) 것인지 여부를 나타냅니다. 매개변수가 0으로 설정되면, 엔진에서 gcore를 사용하여 공유 메모리를 쓰지 않습니다. 매개변수가 1로 설정되면, 엔진은 gcore 유틸리티를 사용하여 공유 메모리를 씁니다.

DUMPCORE

DUMPCORE 매개변수는 가정 오류 발생 시 엔진에서 코어 이미지를 써야(덤프) 하는지 여부를 나타냅니다. 매개변수가 0으로 설정되면, 코어 이미지를 쓰지 않습니다. 매개변수가 1로 설정되면, 가정 오류 시 코어 이미지를 씁니다.

DUMPCNT

DUMPCNT 매개변수는 가정 오류로 인해 스레드가 덤프를 수행하는 횟수를 지정합니다. gcore 유틸리티를 사용하여 공유 메모리가 덤프되거나 코어 파일이 생성됩니다. DUMPCNT 매개변수는 양의 정수값이어야 합니다.

FILLFACTOR

FILLFACTOR 매개변수를 사용하여 추가 인덱스 엔트리가 사용할 수 있는 인덱스 노드의 비율을 나타냅니다. 엔트리를 추가할 때 인덱스 노드에 추가 공간을 남겨놓으면 인덱스 노드에 새 엔트리를 작성해야 할 때 노드를 분할할 필요가 없습니다. 값을 높게 설정하면 대부분의 인덱스 노드가 압축되므로, 엔트리가 100%인 경우 새 엔트리를 추가하려면 노드를 분할해야 합니다.

인덱스를 작성할 때 특정 구문이 인스턴스의 FILLFACTOR를 무시할 수 있으므로, SQL `create index`문을 사용하여 인덱스에 대한 채우기 요인을 정의할 수 있습니다.

USEOSTIME

USEOSTIME 플래그를 사용하여 엔진 프로세스 타이밍에 운영 체제 기능을 사용하는지 여부를 나타냅니다. USEOSTIME 플래그가 1로 설정되어 TRUE가 되면, 엔진이 운영 체제에서 시간을 요청하여, 날짜 및 시간과 간격 값에 서브초 단위를 사용합니다. 이 플래그가 0으로 설정되어 FALSE가 되면, 엔진은 운영 체제에서 1초당 한 번씩 시간을 검색합니다.

MAX_PDQPRIORITY

MAX_PDQPRIORITY는 엔진에서 허용하는 최대 PDQPRIORITY 값을 나타냅니다. 프로세스의 PDQPRIORITY 설정에 따라 엔진이 쿼리를 처리할 때 사용할 수 있는 엔진 자원의 양이 정해지므로, 데이터베이스 관리자는 MAX_PDQPRIORITY 엔진 구성 매개변수를 사용하여 특정 사용자가 액세스할 수 있는 자원의 양을 제한함으로써 엔진에 대한 DSS 로드를 제한할 수 있습니다.

DS_MAX_QUERIES

DS_MAX_QUERIES 매개변수는 실행할 수 있는 동시 DSS 쿼리 수를 제한합니다. `onmode` 유틸리티를 사용하여 런타임 시 DS_MAX

_QUERIES 설정을 무시할 수 있습니다.

DS_TOTAL_MEMORY

DS_TOTAL_MEMORY 매개변수는 의사 결정 지원 쿼리가 사용할 수 있는 총 메모리 양을 지정합니다. 이 숫자는 Informix 버퍼 풀과 운영 체제에서 필요로 하는 메모리를 제외한 물리 메모리보다 적어야 합니다. DS_TOTAL_MEMORY 매개변수는 다음 공식의 경우에서와 같이 SHMTOTAL 매개변수에 배타적입니다.

```
DS_TOTAL_MEMORY = SHMTOTAL - non decision support memory
```

DS_MAX_SCANS

DS_MAX_SCANS 매개변수를 사용하여 동시에 실행할 수 있는 총 스캔 스레드 수를 제한합니다. 엔진은 다음 기준에 따라 하나의 쿼리에 대해 시작되는 스레드 수를 결정합니다.

- 세션의 PDQPRIORITY 값
- DS_MAX_SCANS로 설정된 스캔 스레드에 대해 제한
- MAX_PDQPRIORITY 구성 매개변수로 설정된 PDQ 우선 순위에 대해 제한
- 스캔 중인 테이블의 조각 수

다음 공식에 따라 스레드를 프로세스에 예약할 수 있습니다.

```
threads = min (nfrags, (DS_MAX_SCANS * PDQPRIORITY /100 *
MAXPDQPRIORITY / 100 )
```

onmode 유틸리티를 사용하면 런타임 시 해당 구성 매개변수를 무시할 수 있습니다.

DATASKIP

DATASKIP 매개변수는 IDS 인스턴스에서 IDS 데이터 스킵 기능을 활성화할지 여부를 나타내는 플래그를 표시합니다. 데이터 스킵

기능이 활성화되면, 서버에서는 DBSPACE가 다운된 경우에도 쿼리를 계속 실행할 수 있습니다. IDS 엔진은 오프라인 상태인 DBSPACE에서 데이터가 필요할 경우, 일반적으로 쿼리를 중단합니다.

OPTCOMPIND

OPTCOMPIND 구성 매개변수는 서버가 쿼리에 사용할 최적의 조인 메소드를 결정할 수 있도록 지원하는 플래그입니다. 표 3-9에 허용 가능한 값이 나와 있습니다.

표 3-9 OPTCOMPIND 값

값	설명
0	조인된 두 개의 테이블에서 인덱스를 사용할 수 있으면, 적절한 인덱스를 사용하여 두 개의 테이블을 조인하십시오.
1	트랜잭션 모드가 RR(Repeatable Read)이 아닐 경우, 옵티마이저가 OPTCOMPIND=2의 경우와 마찬가지로 작동합니다.
2	옵티마이저가 액세스 경로를 판별하는 데 비용을 소비합니다. 인덱스 스캔에는 환경 설정이 없으며, 이것이 기본값입니다.

ONDBSPACEDOWN

ONDBSPACEDOWN 매개변수는 쿼리에서 액세스한 DBSPACE가 오프라인일 때 쿼리를 처리하는 동안 취한 조치를 나타냅니다. 표 3-10에는 조치 및 적절한 플래그 설정이 나와 있습니다.

표 3-10 ONDBSPACEDOWN 조치 및 플래그 설정

설정	조치	설명
0	계속	DBSPACE 오프라인을 표시한 후 계속해서 처리합니다.
1	중단	쿼리를 중단하고 엔진을 오프라인 상태로 만듭니다.
2	대기	DBSPACE를 비활성 상태로 만든 후, 다음 체크포인트가 발생하는 즉시 Informix 서버에서 모든 스레드 업데이트를 정지시킵니다.

ONDBSPACEDOWN 구성 매개변수를 설정할 때 DBSPACE를 오프라인으로 할 경우의 영향을 고려해야 합니다. 시스템에서 단일 DBSPACE(데이터베이스의 단일 부분)를 오프라인으로 만들고 계속해서 작업을 하고 의미 있는 결과를 리턴할 수 있으면, 구성 매개변수를 0으로 설정하는 것이 적절할 것입니다. 그러나 하나의 DBSPACE가 오프라인 상태일 때 시스템에서 의미 있는 결과를 리턴할 수 없으면, 이 수를 1이나 2로 설정하는 것이 더 적절할 것입니다.

LBU_PRESERVE

LBU_PRESERVE 매개변수는 마지막 논리 로그를 관리 작업에 예약해야 한다는 것을 서버에 표시합니다. IDS 엔진은 마지막 로그가 아니라 마지막 로그 앞의 로그가 채워졌을 때 추가 로그 업데이트 활동을 중단합니다. (이렇게 하면 실제로 논리 로그 하이 워터 마크에 도달했을 때 로그 백업을 사용하여 논리 로그를 사용 가능한 상태로 만들 수 있습니다.)

OPCACHEMAX

OPCACHEMAX 매개변수는 IDS 광학 서브시스템의 메모리 캐시 크기를 지정합니다. IDS BLOB을 메모리 캐시에 저장한 후에 광학 서브시스템에 씁니다. (그런 후에는 최종으로 광학 장치에 씁니다.)

요약

이 장에서는 IDS 엔진의 설치와 구성에 대해 설명했습니다. IDS 엔진을 설치하는 작업은 상당히 간단한 프로세스로, 이 프로세스는 단순히 소프트웨어 실행 파일과 구성 파일을 디스크에 씁니다. 올바로 설정되어야 소프트웨어가 실행되는 일련의 매개변수가 구성 파일 내에 들어 있습니다.

Informix 엔진의 구성 매개변수를 통해 제품의 유연성의 폭이 넓어질 수 있습니다. 이러한 매개변수를 사용하여 구매 목적과 환경에 적합하도록 Informix 엔진을 튜닝하고 조절할 수 있습니다. 그러나 이러한 유연성에는 책임이 따르며, 따라서 데이터베이스 관리자는

엔진을 구성할 때 사용하는 매개변수를 잘 알고 이해해야 합니다.

구성 매개변수를 설정하는 작업은 반복적인 프로세스인 경우가 많습니다. 초기 설정이 테스트 환경에서는 잘 작동하겠지만, 사용자 로드가 늘어나고 시스템에 더 많은 데이터가 추가됨에 따라 엔진의 성능이 저하되어 매개변수를 조정해야 할 필요가 있을 수 있습니다. 이러한 매개변수를 조정하여 엔진을 튜닝하는 프로세스에 대해서는 5장에서 설명합니다.

다음 장 미리보기

다음 장에서는 Informix 엔진을 사용하여 디스크를 관리하는 프로세스에 대해 다룹니다. 1장에서 설명한 대로, Informix 엔진은 초기 릴리스에서 디스크 관리를 운영 체제 관리에서 제거하고 많은 부분을 엔진으로 흡수했습니다. 이렇게 함으로써 성능과 데이터 무결성이 향상되고 데이터베이스 작업에 대한 유연성이 추가되었습니다. 다음 장에서는 IDS 엔진을 사용하여 이러한 기능을 최적으로 사용할 수 있는 방법에 대해 설명합니다.

IDS 디스크 관리

4장

IDS 엔진은 논리 뷰와 물리 뷰를 통해 디스크를 관리합니다. 이 장에서는 IDS 엔진으로 디스크를 매핑하고 구성하는 프로세스에 대해 설명합니다. 먼저 기본적인 개념과 용어를 소개한 후 엔진 내에 디스크 공간을 구성하는 작업을 제시합니다. 마지막으로 디스크 공간과 데이터베이스를 추가하고 삭제하는 프로세스와 디스크 공간 사용 상태를 모니터링하는 프로세스에 대해 설명합니다.

IDS의 디스크 보기

Informix 엔진은 물리 레벨과 논리 레벨에서 디스크를 봅니다(그림 4-1 참조). 물리 레벨에서 엔진은 청크로 디스크를 참조합니다. 이러한 청크는 장치 특수 파일이나 운영 체제 파일과 관련됩니다. 청크는 참조하는 장치의 전체 또는 일부로 구성됩니다. 청크가 장치의 일부로만 구성된 경우, 물리 장치 내에서 청크의 위치를 결정할 때 오프셋이 필요합니다. (청크 선언은 오프셋이 0인 경우에도 항상 오프셋을 필요로 합니다.) 청크는 차례로 DBSPACE라는 디스크의 논리 표현과 관련됩니다. Informix SQL문이 디스크를 참조하면 디스크가 항상 DBSPACE가 됩니다.

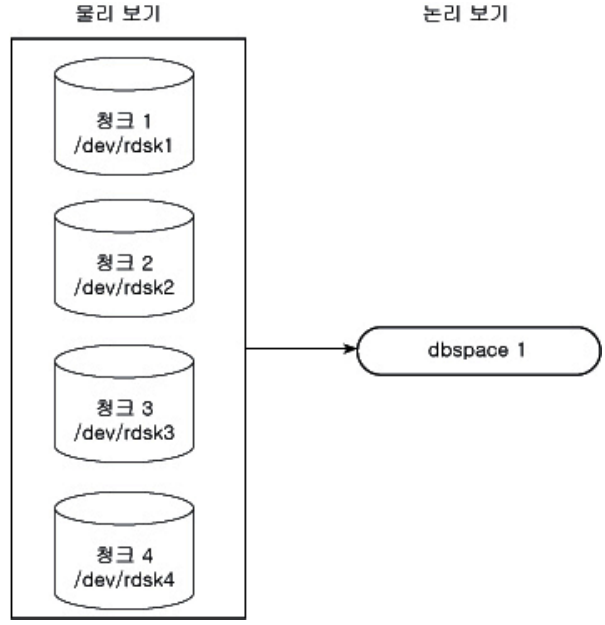


그림 4-1 물리 및 논리 디스크 보기

물리 디스크 레이아웃

Informix 데이터베이스의 청크는 Unix 시스템의 /dev/rdsk 디렉토리에 위치한 파일인 문자 특수 장치를 참조할 수 있습니다. IDS 엔진은 이러한 방법으로 디스크와 상호 작용을 할 때 원시 디스크(Raw Disk)를 사용합니다. 이러한 방법을 통해 최고의 엔진 성능과 가장 신뢰성이 높은 디스크 입출력 방법이 제공됩니다. Informix 엔진은 원시 디스크를 사용할 때 디스크 입출력을 완전하게 제어하므로, 디스크 읽기와 쓰기가 일어나는 시기와 위치를 결정할 수 있습니다.

디스크 입출력의 다른 대체 방법은 디스크 대신 가공된 파일(cooked file)이라고 알려진, 파일 시스템의 파일을 사용하는 것입니다. 가공된 파일은 단순한 운영 체제 파일이며, 운영 체제는 이러한 파일에서 입출력을 수행해야 합니다.

IDS 엔진은 원시 디스크를 사용할 때, 디스크 작업에 비동기 입출력을 수행할 수 있으므로, 전반적인 처리량이 증가합니다. 그러나 가공된 파일은 그렇지 않습니다.

일부 Unix 시스템에서는 블록 특수 장치를 사용하여 디스크에 액세스합니다. 블록 특수 장치는 Unix 입출력 서브시스템을 통해 버퍼에 지정되므로, IDS 엔진에 사용할 때 좋은 몇 가지 기능을 사용할 수 없습니다. IDS 엔진에는 블록 특수 장치를 사용하지 않는 것이 좋습니다.

논리 디스크 레이아웃

IDS 엔진의 경우, 다중 청크는 DBSPACE로 알려진 디스크의 논리 표현에 매핑됩니다. DBSPACE 내의 각 디스크 단위를 페이지라고 합니다. DBSPACE의 물리 레이아웃에는 DBSPACE 내의 청크와 테이블을 설명하는 정보 구조 세트가 들어 있습니다. 청크 사용 가능 목록을 사용하여 청크 내의 사용할 수 있는 페이지를 추적하며, TABLESPACE TABLESPACE를 사용하여 DBSPACE 내의 테이블을 추적합니다.

청크 사용 가능 목록에는 사용 가능한 공간의 각 섹션에 대해 청크의 페이지 오프셋으로서의 시작 페이지와, 페이지로 표시되는 사용 가능한 공간의 길이가 들어 있습니다. 청크 사용 가능 목록은 기본적으로 청크 내의 사용 가능한 디스크 섹션을 가리키는 포인터가 있는 링크된 목록입니다. 청크 내의 사용 가능한 공간을 추적하기 위해 추가 청크 사용 가능 목록이 필요하면, 새 페이지가 생성되어 기존의 청크 사용 가능 목록 페이지에 링크됩니다. 청크 내의 페이지를 테이블에 할당할 때, 청크 사용 가능 목록의 오프셋과 사용 가능한 공간의 크기가 변경되어 변경사항을 인식합니다. 페이지가 사용 가능한 상태가 되면, 청크 사용 가능 목록에 엔트리가 추가되어 사용 가능한 공간의 오프셋 및 크기를 식별합니다. 사용 가능한 상태가 된 공간이 기존의 사용 가능 공간과 인접해 있으면, 사용 가능 공간의 크기가 업데이트되어 새롭게 사용 가능한 상태가 된 공간을 인식합니다.

TABLESPACE는 데이터베이스의 테이블로 구성된 페이지의 컬렉션입니다. 중복하여 이름이 지정된 TABLESPACE TABLESPACE를 사용하여 DBSPACE 내의 모든 TABLESPACE의 위치와 구조를 추적합니다(TABLESPACE를 추적하는 TABLESPACE 또는 구조). TABLESPACE TABLESPACE에는 TABLESPACE에 있는 페이지의 비트맵과 DBSPACE 내의 TABLESPACE를 설명하는 일련의 페이지가 들어 있습니다. 이 테이블의 각 TABLESPACE에는 컬럼, 인덱스 및 자체 추적에 사용되는 TABLESPACE TABLESPACE의 첫

번째 페이지를 포함한 테이블의 익스텐트에 관한 정보가 들어 있습니다.

DBSPACE 유지보수

앞에서 설명한 대로, chunk는 물리 디스크 공간을 나타내며, DBSPACE는 chunk 세트를 나타냅니다. DBSPACE는 사실상 디스크 공간의 논리적 표현이며 chunk가 디스크 공간의 물리적 표현입니다. IDS 엔진을 사용하여 초기 chunk로 DBSPACE를 생성한 후에 DBSPACE에 chunk를 추가합니다. chunk를 추가하는 프로세스는 원시 디스크와 가공된 파일의 경우 모두 동일합니다.

onspaces 명령을 사용하여 명령행에서 DBSPACE와 DBSPACE를 구성하는 chunk를 유지보수할 수 있습니다. onspaces 명령 유형은 다음과 같습니다.

```
onspaces -d <dbspacename> |
            -c <dbspacename> |
            -a <dbspacename>
      -p <pathname>  -o <offset>  -s <size>  [-m path offset]
```

'-a' 매개변수를 사용하여 DBSPACE, BLOBSPACE 또는 청크를 추가할 수 있고, '-c' 매개변수를 사용하여 새 DBSPACE나 BLOBSPACE를 작성할 수 있으며, '-d' 매개변수를 사용하여 DBSPACE, BLOBSPACE 또는 청크를 삭제할 수 있습니다. 표 4-1에 onspaces 유틸리티로 전달되는 매개변수 값이 설명되어 있습니다.

표 4-1 onspaces 유틸리티의 매개변수 값

매개변수	설명
dbspacename	-d 또는 -a 매개변수의 인수일 때 작성할 DBSPACE 또는 BLOBSPACE. DBSPACE가 -a 매개변수의 인수일 때 청크를 추가할 DBSPACE
pathname	사용할 장치 특수 파일 또는 운영 체제 파일의 경로 이름
offset	참조한 장치 또는 파일에 대한 오프셋
size	킬로바이트로 표시된 파일 또는 장치의 크기

DBSPACE 작성 및 삭제

DBSPACE를 IDS 인스턴스에 추가하는 방법 중 하나는 onspaces 명령을 사용하는 것입니다. onspaces 명령을 사용하여 다음과 같이 DBSPACE를 작성할 수 있습니다.

```
onspaces -c dbspace1 -p /dev/rdsk0 -o 0 -s 12000
```

이 명령을 실행하면 오프셋이 영(0)이고 크기가 12,000KB인 /dev/rdsk0의 첫 번째 청크를 사용하며 dbspace1이라는 새 DBSPACE가 작성됩니다.

또는 onmonitor 유틸리티를 사용하여 메뉴 구동의 문자 기반 인터페이스를 사용하는 DBSPACE를 작성할 수 있습니다. onmonitor를 사용할 경우, 사용자는 "DBSPACE" 메뉴와 "Create a dbspace" 옵션을 선택하게 됩니다. 그러면 DBSPACE 이름, DBSPACE의 기본 청크(첫 번째 청크)의 경로 이름, DBSPACE가 미러링을 사용할 것인지 여부 또는 DBSPACE가 임시 DBSPACE인지 여부에 관한 정보가 들어갈 필드가 있는 화면이 표시됩니다. 적절한 정보를 입력한 후 Esc 키를 눌러 해당 정보를 저장하고 DBSPACE를 작성하십시오. 입력한 정보는 필드에서 종료할 때 검사되므로 위반 값에 대한 피드백이 즉시 제공됩니다. (onspaces 명령은 이와 대조적으로 값이나 매개변수가 잘못되었는지를 항상 식별하지 않습니다.)

때때로 DBSPACE를 삭제하고 디스크 공간을 다른 목적에 사용해야 할 필요가 있습니다. Informix 엔진에서는 DBSPACE에 현재 데이터가 있으면 DBSPACE를 삭제할 수 없습니다. (DBSPACE의 데이터가 테이블의 데이터 레코드일 필요는 없으나, 단순히 데이터베이스에 관한 정보가 포함된 레코드가 있는 다수의 시스템 카탈로그 테이블이 들어 있는 빈 데이터베이스가 될 수도 있다는 점에 유의하십시오.) DBSPACE를 삭제하는 사용자는 Informix 사용자이거나 데이터베이스 관리자(DBA) 권한이 있어야 합니다. DBSPACE를 삭제하는 onspaces 명령의 구문은 다음과 같습니다.

```
onspaces -d <dbspacename> [-p <pathname> -o <offset>] [-y]
```

선택적 경로 이름 구문은 청크를 삭제할 경우에만 필요합니다. mydbspace라는 빈 DBSPACE를 삭제하려면 다음 명령을 사용하십시오.

```
onspaces -d mydbspace
```

이 명령을 실행하면 DBSPACE를 실제로 삭제하기 전에 해당 명령을 실행한다는 메시지가 사용자에게 표시됩니다. 그러나 '-y' 플래그를 명령에 붙이면, 사용자에게 메시지가 표시되지 않고 아무런 주석 없이 DBSPACE가 삭제됩니다.

onmonitor 명령을 사용하여 DBSPACE나 BLOBSPACE를 삭제할 수도 있습니다. 먼저 "DBSPACE" 메뉴를 선택한 후에 DBSPACE "Drop" 옵션을 선택하십시오. 그러면 시스템의 현재 DBSPACE 목록이 표시됩니다. 이제 화살표 키를 사용하여 삭제할 DBSPACE를 강조 표시를 하고 Ctrl-B 또는 F3 키(기능 키 3)를 눌러 DBSPACE를 삭제하십시오.

DBSPACE에 청크 추가

계속적으로 데이터가 DBSPACE에 추가되면, 추가 디스크 공간을 할당해야 합니다. DBSPACE에 공간을 추가하려면 청크를 추가해야 합니다. onspaces 명령을 실행하면 기존 DBSPACE에 청크를 추가하는 옵션 세트가 제공됩니다. 다음 명령은 오프셋이 0인 상태에서 /dev/dsk1 원시 장치에서 dbspace1에 청크를 추가합니다. 추가된 청크의 크기는 12,000KB입니다.

```
onspaces -a dbspace1 -p /dev/rdsk/dsk1 -o 0 -s 12000
```

다음 명령은 -p 매개변수에 붙는 인수를 제외하고 이전의 명령과 동일합니다. 이 경우에, Unix 운영 체제 플랫 파일을 사용하여 DBSPACE에 추가하는 청크에 대한 장치를 참조합니다. 이 파일이 Unix 시스템 파일이면, IDS 엔진에서 파일을 열고 파일에 2진수 영(0)을 사용하여 지정된 크기로 확장합니다.

```
onspaces -a dbspace1 -p /mydata/myfile -o 0 -s 12000
```

이외에도 onmonitor 프로그램을 사용하여 DBSPACE에 청크를 추가할 수 있습니다. onmonitor 프로그램을 사용하여 청크를 추가하려면 "Dbspace" 메뉴 옵션을 선택한 후 "Add chunk" 옵션을 선택하면 됩니다. 시스템에 있는 현재 DBSPACE의 목록은 그림 4-2와 같이 표시됩니다.

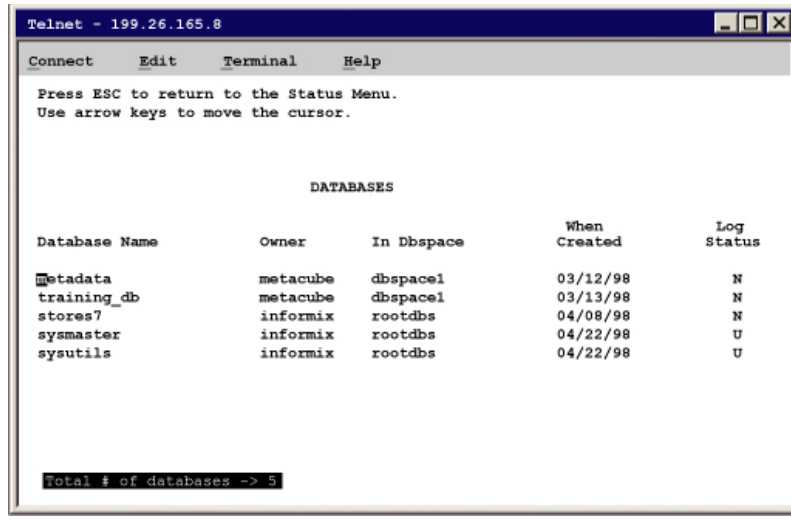


그림 4-2 onmonitor를 사용하여 청크 추가

사용자는 화살표 키를 사용하여 청크를 추가할 DBSPACE를 강조 표시를 한 후 Ctrl-B 또는 F3 키(기능 키 3)를 누르십시오. 그러면 앞서 설명한 onspaces 명령에 대한 매개변수 입력 필드를 포함하는 "add chunk" 화면이 표시됩니다. 여기에는 청크를 추가할 DBSPACE의 이름, 추가 중인 청크의 전체 경로 이름, 청크에 대한 오프셋, 청크의 크기 및 해당하는 경우 미러 청크 정보가 표시됩니다. 데이터를 입력하면 onmonitor 프로그램에서 각 필드에 대한 입력을 검증합니다.

DBSPACE 미러링 사용

DBSPACE 미러링을 사용하려면, '-m' 매개변수를 사용하여 DBSPACE에 추가되는 각 청크에 미러 정보를 입력해야 합니다. DBSPACE 미러링을 사용할 경우, DBSPACE에 작성한 모든 데이터는 IDS 엔진을 사용하여 다른 DBSPACE에 작성하는 방법으로 미러됩니다. 기본 DBSPACE가 사용 중이면, 데이터 읽기 작업은 선택적으로 미러 DBSPACE에서 읽을 수 있습니다. 미러링이 시작되면 onspaces 명령이 사용되어 미러링을 끝 때까지 사용자 개입없이 실행됩니다. 다음 예에서는 dbspace1에 기본 청크 및 미러 청크를 추가합니다.

```
onspaces -a dbspace1 -p /dev/rdsk1 -o 0 -s 12000 -m /dev/rdsk2 0
```

이러한 명령문을 실행하면 dbspace1에 청크와 해당 미러 청크가 함께 추가됩니다. 명령이 제대로 실행되면 미러링을 통해 DBSPACE가 작성되며, 그렇지 않은 경우 이 명령이 실패합니다.

청크 삭제

청크에서 현재 사용되고 있는 공간이 없을 경우, 청크가 삭제될 수도 있습니다. 데이터베이스 관리자 권한이 있는 사용자만이 다음 onspaces 구문을 사용하여 청크를 삭제할 수 있습니다.

```
onspaces -d <dbspacename> [-p <path name> -o <offset>] [-y]
```

dbspace1에서 청크를 삭제하려면, 다음 명령문을 사용하십시오.

```
onspaces -d dbspace1 -p /dev/rdska01 -y
```

이 명령을 실행하면 dbspace1이라는 DBSPACE에서 /dev/rdska01 청크가 삭제됩니다. onspaces 명령에 '-y' 플래그가 지정되었으므로, 사용자에게 프롬프트가 표시되지 않고 청크가 삭제됩니다.

DBSPACE에서는 비어 있는 청크만 삭제할 수 있습니다. IDS가 데이터를 청크에 쓰는 방식의 경우, 한동안 시스템의 일부였던 청크에 청크 페이지의 여러 다양한 테이블의 데이터가 있을 수도 있습니다. 청크를 사용 가능하게 하려면 제공된 청크에서 페이지를 제거하기 위해(데이터를 언로드하고, 테이블을 삭제한 후 데이터를 재로드하는 프로세스를 통해) 테이블을 삭제하거나 이동시켜야 합니다. 이러한 프로세스는 아주 까다로울 수 있습니다.

실제로, DBSPACE에서 단일 청크를 삭제하는 것이 일반적인 것은 아닙니다. 좀더 일반적으로 사용하는 방법은 DBSPACE에서 데이터를 완전히 제거한 후 DBSPACE를 삭제하여 DBSPACE 내의 청크를 사용 가능하게 하는 것입니다.

소프트 링크를 사용하여 청크 지정

청크 이름과 장치 파일 간에 간접 레벨을 작성하는 것이 유용할 때가 많습니다. 청크가 일단 IDS 엔진에 추가되면 이름과 크기를 수

정할 수 없기 때문입니다. 장치 파일을 이동하거나 디스크의 이름을 재지정해야 할 경우에, 효과적으로 청크 참조의 이름을 재지정하는 것이 유용합니다. 이러한 간접 레벨은 Unix 파일 링크를 사용하여 작성할 수 있습니다. 링크는 파일 시스템 전반에 걸친 소프트 링크일 수도 있고 링크가 동일한 파일 시스템에 파일 자체로 있는 하드 링크일 수도 있습니다.

Unix ln 명령을 사용하여 소프트 링크를 작성하십시오. 소프트 링크를 작성하는 구문은 다음과 같습니다.

```
ln -s <existing file> <link file>
```

장치 파일에서 Unix 파일까지 소프트 링크를 작성하려면 다음 명령을 실행해야 합니다.

```
ln -s/dev/rdsk0 /data/myrawfile
```

이 명령을 실행하면 Unix 파일과 원시 장치 파일(raw device file) 간의 소프트 링크가 작성됩니다. 원시 장치 파일의 이름을 변경한 경우(데이터가 새 디스크, 새 제어기로 이동되었거나 새 원시 드라이버를 획득한 경우), 원시 장치 파일 이름이 다음과 같이 변경될 수 있습니다.

```
rm /data/myrawfile
in -s /dev/rsk0a1 /dev/rsk0a1
```

데이터베이스 테이블 작성

데이터베이스, 테이블 및 인덱스를 생성할 때 Informix-SQL문은 DBSPACE를 참조합니다. 데이터베이스가 생성되면, 데이터베이스와 데이터베이스의 시스템 테이블은 기본적으로 root DBSPACE에 배치됩니다. 다음과 같이 <dbspace> 명령문을 사용하여 기본값을 무시할 수 있습니다.

```
create database db1 in daspace1;
```

주의: 링크 관련 문제

링크를 사용하여 원시 장치를 IDS에 매핑할 때, IDS는 사용하는 실제 물리 디스크를 인식하지 못합니다. 링크가 상호 참조되어 두 개의 청크가 동일한 디스크 부분에 지정될 수도 있습니다. 다음 예가 이를 설명해 줍니다.
다음과 같이 chunk1이라는 링크를 통해 단일 청크를 /dev/rdsk1a 원시 장치로 매핑하여 DBSPACE가 작성됩니다.

```
ln -s /ids_chunk/chunk1 /dev/rsk0a1
onspaces -c -d dbspace1 -p /ids_disk/chunk1
```

다음과 같이 부주의로 단일 청크를 첫 번째 DBSPACE에 사용한 동일한 원시 장치에 매핑하면 두 번째 DBSPACE가 작성됩니다.

```
ln -s /ids_chunk/chunk2 /dev/rsk0a1
onspaces -c -d dbspace2 -p /ids_disk/chunk2
```

IDS 엔진에서는 이러한 오류를 인식하지 못합니다. 엔진은 내부적으로 경로 이름(onspaces 명령에 -p 매개변수로 지정)의 유효성을 확인하지만, 이 경우에는 유효성 확인을 통해 /ids_disk/chunk1과 /ids_disk/chunk2가 별개의 디스크 부분으로 인식됩니다.
이러한 상황이 발생할 경우, DBSPACE가 작성되기는 하지만, 곧 이어 디스크 쓰기 실패가 발생하고 데이터가 손상됩니다. 이 때 사용할 수 있는 유일한 복구 옵션은 가능하다면 두 개의 DBSPACE를 모두 삭제함으로써 문제를 수정하고 테이프에서 복구하는 것입니다.
"유비무환"이라는 속담이 있듯이, 청크와 원시 디스크의 링크 상태를 면밀히 추적하여 앞서 언급한 문제가 발생하지 않도록 하는 것이 최선입니다. 단일 쉘 스크립트 파일 내에 모든 링크 명령문을 배치시키면 링크 유효성 확인에 필요한 일관성 있는 위치가 제공됩니다. 차후 버전에서는 이러한 쉘 스크립트를 참조하여 링크가 매핑되는 물리 디스크를 결정함으로써 링크를 더 추가 할 수 있습니다.

이 명령문을 실행하면 dbspace1에 db1이라는 데이터베이스가 생성됩니다. 그러면 데이터베이스의 모든 시스템 테이블이 db1의 DBSPACE에 배치되고, 기본적으로 데이터베이스에 작성된 모든 테이

블이 해당 DBSPACE에 배치됩니다. DBSPACE에 데이터베이스가 작성되었으면, 이동이 불가능합니다. 다음과 같이 DBSPACE절을 삭제하면, root DBSPACE에 데이터베이스가 작성됩니다.

```
create database db1;
```

Informix 데이터베이스에 테이블이 생성되면, 작성된 테이블은 기본적으로 데이터베이스가 위치한 DBSPACE에 저장됩니다. 다음 create table문은 이러한 기본값을 보여줍니다.

```
create database db1 in dbspace1;
create tabl t1 (col1 int, col2 char (20));
```

이 테이블은 데이터베이스가 생성된 DBSPACE인 dbspace1에 생성됩니다. 이 기본값은 다음과 같이 무시할 수 있습니다.

```
create database db1 in dbspace1;
create tabl t1 (col1 int, col2 char(20)) in dbspace5;
```

create table문을 실행하면 DBSPACE dbspace5에 t1 테이블이 생성됩니다. 다음 절에서와 같이 분할화 명령문을 통해 테이블을 디스크에 추가로 분산시킬 수 있습니다.

IDS 엔진이 DBSPACE의 청크를 사용할 때 사용하는 청크의 순서는 추가된 순서입니다. 엔진은 테이블의 페이지를 인접한 상태로 유지하려고 합니다. 작성된 테이블은 테이블 익스텐트를 통해 사전 확장됩니다. 익스텐트는 테이블에 대한 인접한 페이지 컬렉션이고, 테이블은 하나 이상의 익스텐트 세트입니다(그림 4-3 참조). 익스텐트의 페이지는 인접해 있지만, 테이블로 구성된 익스텐트는 인접해 있지 않습니다(두 개의 인접한 익스텐트가 IDS에 의해 단일 익스텐트로 병합되므로). 익스텐트는 단일 청크에 상주하며, 여러 청크에 걸쳐 있을 수 없습니다.

기본값이 8개의 시스템 페이지인 초기 익스텐트로 하나의 테이블이 작성되므로, 2KB 페이지 크기인 시스템에서 기본 익스텐트 크기는 16K가 됩니다. 새 익스텐트 크기 값을 사용하여 테이블을 작성해도 익스텐트의 기본값은 역시 16K입니다. 다음과 같이 create table문으로 초기 익스텐트와 다음 익스텐트를 지정할 수 있습니다.

그림 4-3 익스텐트가 있는 테이블

```
create table testit ( col1 int,
                      col2 int,
                      col3 char (10) )
   extent size 50
   next size 10 ;
```

이 테이블은 초기 익스텐트 크기를 extent size절에서 지정한 대로 50K로 작성하고, 다음 익스텐트 크기를 next size절에서 지정한 대로 10K로 작성합니다.

테이블의 페이지가 행으로 채워질 때, 테이블의 첫 번째 익스텐트가 채워집니다. 그러면 IDS 엔진은 필요한 인접 공간량("next extent" 크기에 따라 테이블에 정의된 양)에 따라 공간을 찾기 시작합니다. 엔진은 청크가 DBSPACE에 추가된 순서로 DBSPACE의 청크 사용 가능 목록을 검색하여 이를 수행합니다. 청크에서 적절한 인접 공간을 찾으면(찾은 공간 >=다음 익스텐트 크기), 해당 청크의 테이블에 다음 익스텐트를 할당합니다. 할당된 다음 익스텐트가 테이블의 마지막 익스텐트와 인접할 경우, 두 익스텐트는 단일 익스텐트로 병합됩니다.

익스텐스가 적은 테이블이 스캔하기 더 수월합니다. 테이블에 익스텐스가 많으면(oncheck -pe 명령의 출력을 통해 확인), 테이블의 데이터를 인접하게 하여 테이블 스캔 성능을 향상시킬 수 있습니다. 테이블의 데이터를 인접하도록 하려면, 현재 테이블에 있는 데이터를 보관할 수 있을 만큼 충분히 큰 초기 익스텐트가 있는 새 테이블을 작성해야 합니다. 이렇게 하기 위해서는 데이터를 이전 테이블에서 새 테이블로 이동해야 합니다. 다음 SQL을 실행하면 데이터를 디스크에 언로드하지 않고도 이러한 작업을 완료할 수 있습니다.

```
create table newtable (col1 int, col2 int, col3 char(10) )
      extent size 100
      next size 50;
    insert into newtable select * from testit;
    rename table testit to x;
    rename table newtable to testit;
  -- validate that operation was successful
  drop table x; - drop the old table
```

이 SQL 스크립트는 익스텐트 크기가 더 큰 새 테이블을 작성한 다음, 이전 테이블의 데이터를 새 테이블로 복사합니다. 그러면 이전 테이블의 이름이 임시 이름으로 재지정되고, 새 테이블은 올바른 테이블의 이름으로 재지정됩니다. 복사 조작이 완료되면, 임시 이름을 갖는 이전 테이블을 삭제할 수 있습니다. 이러한 조작의 경우, 데이터를 디스크로 언로드할 필요는 없지만, 테이블에서 조작을 완료할 때 필요한 공간이 두 배이어야 합니다. 다른 옵션으로는 테이블의 데이터를 디스크로 언로드하고, 이전 테이블을 삭제한 다음, 초기 익스텐트 크기가 더 큰 테이블을 다시 작성한 후 데이터를 다시 테이블로 로드하게 됩니다.

테이블에 익스텐트가 너무 많기는 하지만 관리자가 테이블을 다시 로드하지 않으려면, 다음 익스텐트 크기를 변경하여 IDS에서 테이블의 새 익스텐트를 작성해야 하는 필요성을 줄일 수 있습니다. 다음과 같이 alter table문을 사용하여 익스텐트의 크기를 변경할 수 있습니다.

```
alter table testit
       modify next size 20;
```

이 예를 통해, testit 테이블의 다음 익스텐트 크기가 10K에서 20K로 변경됩니다.

테이블의 로크 단위

테이블을 작성하면 페이지 레벨 로크나 행 레벨 로크 두 가지 방법으로 로크 단위를 표현할 수 있습니다. 기본 로크 단위는 페이지 레벨 로크입니다.

페이지 레벨 로크의 경우, 테이블을 읽는 사용자가 단일 행의 로크를 필요로 하면 전체 페이지가 로크됩니다. 사용자가 해당 페이지의 다음 행을 읽어야 할 때, 사용자는 이미 해당 페이지에 대한 로크를 보유하고 있으므로 엔진에서 사용자에 대한 다른 로크를 획득하지 않아도 됩니다.

페이지 레벨 로크는 프로세스에서 전체 테이블 스캔과 같이 테이블에 있는 많은 수의 인접하는 행을 읽어야 할 경우 효과적인 로크 방식입니다. 전체 페이지에 대해 로크를 하나만 획득해야 하므로 로크 오버헤드가 더 적게 듭니다. 테이블을 읽을 때 필요한 로크가 적을수록 더 적은 시스템 자원이 사용됩니다.

페이지 레벨 로크의 단점은 동시성이 상실된다는 점입니다. 페이지에서 적은 행 수를 읽는 프로세스라도 페이지의 모든 행을 로크하게 되므로, 사용하지 않는 행에 대해서는 액세스가 제한됩니다.

행 레벨 로크의 경우, 액세스하는 행만 로크하고 전체 페이지를 로크하지는 않습니다. 따라서 동시성이 훨씬 더 높아지기는 하지만, 시스템 오버헤드와 사용되는 시스템 자원은 늘어납니다.

동시성이 중요하며 일반적으로 장시간 로크를 보유하지 않는 OLTP 시스템과 같은 시스템에서는 행 레벨 로크가 최적의 선택입니다. 일반적으로 테이블에서 여러 개의 인접한 행을 읽으며 동시성이 제한될 수도 있는 배치 시스템에는 페이지 레벨 로크가 적합합니다. 페이지 레벨 로크는 또한 전체 테이블 스캔이 일반적인 데이터 웨어하우스나 의사 결정 지원 액세스에도 적합합니다.

다음 create table문을 실행하면 시스템의 기본값인 페이지 레벨 로크가 무시됩니다.

```
create table regions (region int,
                      region_name char (10),
                      region_description char (25)
                lock mode row;
```

이 SQL문에서는 행 로크 모드를 사용하여 테이블을 작성합니다. 이렇게 하면 해당 테이블에 행 레벨 로크가 설정됩니다.

alter table문을 사용하여 테이블의 로크 모드를 변경할 수 있습니다. 다음 명령문을 실행하면 테이블의 로크 모드가 기본값이 페이지 레벨 로크에서 행 레벨 로크로 변경됩니다.

```
alter table regions lock mode (row);
```

(트랜잭션의 격리 모드 및 데이터베이스의 트랜잭션 모드 등 많은 요인이 데이터 액세스의 로크 및 동시성에 영향을 준다는 점에 유의하십시오. 이러한 문제는 5장에서 자세하게 설명합니다.)

분할화

Informix Dynamic Server의 가장 중요한 기능 중 하나는 디스크의 다중 분할 영역에서 데이터를 정적으로 분할하는 기능입니다. Informix 구문 분석기는 이 테이블 분할화를 이해하고 이 기능을 사용하여 데이터를 병렬로 처리하는 쿼리 계획을 개발할 수 있습니다(그림 4-4 참조).

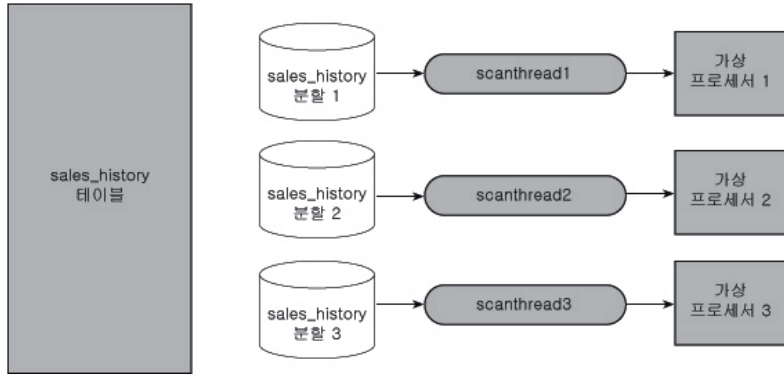


그림 4-4 분할화 및 병렬 액세스

Informix 엔진에서는 Informix-SQL 'fragment by'절을 사용하여 DBSPACE 전체에 데이터를 분산시키는 방법으로 이러한 기능을 사용할 수 있습니다. 즉, 'fragment by'절을 'create table'문에

사용하여 테이블의 분할화 방법(분산 방법이라고도 함)을 지정합니다. 분할화 방법이란 라운드 로빈 분산을 지정하거나 또한 표현식 기반 분산 방법을 사용하여 데이터를 분산시키는 것입니다.

다음 명령문을 실행하면 라운드 로빈 분산 방법으로 네 개의 DBSPACE에 데이터를 분산시킵니다.

```
create table t1 (col1 int,
            col2 char (20),
            col3 int )
fragment by round robin in dbs1, dbs2, dbs3, dbs4;
```

여기서 데이터는 DBSPACE dbs1, dbs2, dbs3 및 dbs4에 연속적인 방식으로 분산됩니다. 즉, 데이터를 테이블에 쓰면 하나의 행을 dbs1에 쓰고, 다음 행을 dbs2에 쓰는 방식입니다.

표현식 기반 분할화 방법이 병렬 처리를 수행하는 IDS 엔진의 기능과 결합되면 강력한 쿼리 기능을 사용할 수 있습니다. 또한 IDS 엔진에서 쿼리를 처리할 때, 쿼리를 만족시키려면 테이블이나 인덱스를 스캔해야 할 수도 있습니다. 더불어 스캔 스레드를 시작하여 이러한 처리를 수행하며, 가능하다면 다중 스캔 스레드를 시작하여 쿼리를 실행하는 동안 각 개별 조각을 읽어야 할 수도 있습니다. 또한 테이블이 다중 조각 전역에 분산되면, 특정 조건에 따라 이러한 조각을 읽을 다중 스캔 스레드를 시작할 수도 있습니다. 어떤 표현식에 의해 테이블이 분할되고 쿼리의 where절이 해당 표현식의 컬럼 값을 참조하면, 테이블 스캔 작업에서 조각 제거로 알려진 프로세스를 통해 하나 이상의 조각을 제거할 수도 있습니다. 조각 제거는 쿼리 처리에 필요한 작업량을 줄일 수 있으므로 쿼리 처리에 유용합니다.

테이블 내의 컬럼과 상수를 포함하는 모든 표현식을 사용하여 실제적으로 테이블을 분할할 수 있습니다. 다음은 Sales 데이터가 들어 있는 테이블에 대한 분할화 방법을 보여줍니다.

```
create table sales_facts (
            salesman int,
            region char(5),
            sales_date date,
            sales_amount decimal (12,2) )
fragment by expression
            region in ("NW", "NE", "MA" ) in dbspace1,
```

```
region in ("SE", "SN" ) in dbspace2,
region in ("SW", "MW" ) in dbspace3;
```

이러한 분할화 방법을 수행하면 세 개의 DBSPACE에 테이블의 행이 분산됩니다. 여기서 분할은 영역 키(다차원 모델링 용어로는 차원)에 대한 등식을 기반으로 합니다. 디스크에 쓰는 레코드의 영역 컬럼에 따라 특정 영역의 데이터는 특정 DBSPACE에 배치됩니다.

IDS 엔진은 쿼리를 해석하여 데이터가 들어 있는 테이블을 필터 기준으로 분할해야 할지 여부를 식별하는 옵티마이저를 사용합니다. 필터 기준으로 데이터를 분할하면 필터 기준에 따라 쿼리 계획에서 일부 조각을 제거할 수도 있습니다. 다음 쿼리는 이 기능을 보여줍니다.

```
select *
from sales_facts
where region in ("NW", "SE" );
```

옵티마이저는 이러한 쿼리를 검토하여 테이블의 데이터가 영역 컬럼에서 분할되었는지를 결정합니다. 쿼리에서 "NW" 영역(dbspace1에 있음)과 "SE"(dbspace2에 있음)만 요청하므로, dbspace3에 대한 스캔 스레드는 필요하지 않습니다. dbspace1과 dbspace2에 있는 두 개의 테이블 조각에 대해 두 개의 스캔 스레드가 시작되며 dbspace3에 있는 조각은 검색 경로에서 제거되어 검색되지 않습니다.

분할화 방법 개발

분할화를 통해 테이블 데이터의 물리 레이아웃을 제어할 수 있습니다. 이전 장에서 설명한 대로, IDS 엔진은 테이블의 데이터 분할화를 통해 테이블을 스캔할 때 사용할 스캔 스레드 수를 결정합니다. 별도의 CPU VP에서 각 스캔 스레드를 실행할 수 있습니다. IDS 엔진은 병렬 처리를 사용하여 데이터베이스에서 데이터를 검색하므로, 데이터의 분할화가 IDS 엔진의 병렬 처리에 직접적인 영향을 줍니다.

병렬 처리를 할 수 있도록 테이블에서 데이터 조각을 찾음으로써, 특정 쿼리(주로 많은 양의 데이터를 스캔하는 의사 결정 지원 쿼리)의 성능을 향상시킬 수 있습니다.

보관을 위한 분할화

분할화는 일반적으로 병렬 처리를 통해 엔진의 성능을 향상시키고자 할 때 가장 많이 사용합니다. 그러나 백업과 복구 목적으로 데이터를 세그먼트할 때도 분할화를 사용할 수 있습니다. 테이블의 데이터 부분이 드물게 업데이트되는 기존 데이터일 때가 이러한 예에 해당됩니다. 기존 데이터에 대한 백업과 복구 스케줄을 통해 데이터가 한 달에 한 번만 보관되므로 테이블의 남은 현재 데이터는 매일 보관해야 합니다. 기존 데이터를 현재 데이터와는 별도의 DBSPACE에 두는 방법으로 분할화하면, 기존 데이터와 현재 데이터를 별도의 백업 스케줄에 둘 수 있습니다. 이는 특정 IDS 보관/복구 유틸리티가 선택적으로 DBSPACE를 참조하여 DBSPACE를 단독으로 또는 그룹으로 보관할 수 있기 때문에 가능합니다.

성능을 위한 분할화

쿼리를 병렬로 처리하려면, 테이블의 데이터에 액세스하는 방법을 반영하는 분할화 방법을 사용하여 데이터를 디스크에 정렬해야 합니다. 가장 중요한 것은, 두 가지 주요 데이터베이스 시스템 유형인 의사 결정 지원 시스템과 온라인 트랜잭션 처리에서 데이터 액세스 방법이 다르다는 점입니다. 이러한 두 가지 유형의 데이터베이스 시스템은 데이터를 매우 상이한 방식으로 검색하며, 따라서 분할화 방법의 경우에서와 마찬가지로 적합한 방법을 결정할 때 필요한 예상 응답 시간과 데이터에 지정할 쿼리에 고려해야 할 몇 가지 사항이 있습니다.

OLTP 분할화 방법

OLTP 시스템은 일반적으로 많은 동시 사용자가 적은 수의 레코드에 신속하게 액세스해야 합니다. OLTP 시스템에 사용할 수 있는 분할화 방법에는 몇 가지가 있으며 각 접근 방식마다 이점이 있고 비용도 듭니다.

OLTP 응용프로그램의 데이터에 액세스하는 경우, 테이블의 많은 부분을 스캔할 시간이 없으므로 데이터를 신속하게 검색해야 합니다. 이러한 이유로 인해 일반적으로 인덱스를 사용하여 데이터에 신속하게 액세스합니다. 트랜잭션 프로파일에는 일반적으로 b-트리

인덱스를 통한 빠른 읽기가 포함되어 있으므로, 적은 수의 레코드를 읽어 올바른 인덱스 엔트리에 도달합니다. 인덱스 엔트리에는 데이터가 저장된 테이블의 페이지 및 페이지의 해당 행에 대한 참조가 들어 있습니다. 이 페이지/행 참조가 사용되여 테이블에서 행이 검색됩니다. 전체 작업을 수행하는 데는 일반적으로 1초가 소요되지 않습니다. 사용량이 많고 튜닝이 잘 된 시스템일 경우, IDS 엔진은 1초에 수 백 개의 OLTP 트랜잭션을 처리할 수 있습니다.

OLTP 시스템에서 분할화와 분할화의 병렬 처리가 시스템 성능에 거의 아무런 영향도 주지 않을 수 있습니다. OLTP 트랜잭션 처리 로드가 많은 응용프로그램의 경우, 여러 디스크 및/또는 제어기에 실제 데이터 레코드(인덱스가 아닌)를 분산시키면 디스크 입출력 로드가 균형 있게 분산되고 디스크 경쟁이 줄어듭니다. 인덱스 페이지의 액세스 레벨에 비추어 볼 때, 인덱스 페이지가 계속해서 버퍼에 남아 있을 것이라는 가정 하에, 이러한 방법을 사용하여 인덱스를 단일 DBSPACE(분리된 인덱스)에 배치할 수 있습니다. 테이블의 데이터는 라운드 로빈 방식으로 분할되어 사용할 수 있는 디스크 및/또는 제어기 세트 전체에 균등하게 분산될 것입니다. (이는 여러 개의 디스크에 데이터를 균등하게 배치하는 디스크 스트라이핑 방법과 유사합니다.)

많은 양의 데이터를 버퍼 풀에 적용해야 하는 시스템에서 인수가 작성될 수도 있으며, 데이터 테이블 페이지의 많은 부분이 메모리에 있으므로, OLTP 디스크 입출력 옵션은 사실상 무시될 수도 있습니다. `-g iof` 옵션과 함께 `onstat` 명령을 사용하면 디스크 부분의 입출력에 경쟁 문제가 있는지를 알 수 있습니다. 이 명령을 실행하면 각 청크/파일에 입출력 통계가 인쇄되며, 여기에는 임시 파일과 정렬 작업 파일에 관한 정보가 포함됩니다. (`onstat` 명령을 사용하여 디스크를 모니터링하는 방법에 대해서는 5장을 참조하십시오.)

중요한 디스크 입출력에 경쟁 문제가 없다고 결정되면, 분할화 방법이 OLTP 트랜잭션 처리 성능을 저해하지 않는 한, 테이블의 데이터에 대한 백업과 복구의 필요성에 따라 분할화 방법을 사용해야 합니다. 이 경우, 인덱스는 분리된 상태에서 여러 개의 조각에 테이블 데이터를 분산시켜야 한다는 의미입니다.

분할화는 인덱스에도 도움이 될 수 있습니다. 인덱스의 크기가 커짐에 따라, b-트리 인덱스가 더욱 깊어지며, b-트리 인덱스가 깊어질수록 인덱스에 대한 액세스 속도가 느려집니다. 인덱스가 작을수록 내부 b-트리의 깊이는 더 얕아지고 액세스는 더 빨라집니다. 이러한 이유로 큰 테이블(해당 인덱스도 큼)에서는 첨부된 인덱스가 있는 표현식 분할화 스키마를 사용하는 것이 좋습니다. 표현식 분할화

스키마를 사용하여 큰 테이블을 몇 개의 더 작은 부분으로 나눌 수 있으며, 이 때 각 부분에 첨부된 해당 인덱스도 더 적어지므로 더 얕아집니다. 인덱스 작성, 분할화 및 기타 튜닝에 관한 부분은 5장에 자세히 설명되어 있습니다.

DSS 분할화 방법

DSS 쿼리의 트랜잭션 프로파일은 테이블에서 다수의 레코드를 읽고 데이터에 일종의 요약 작업을 수행하는 쿼리입니다. DSS 쿼리는 일반적으로 OLTP 쿼리에 비해 실행 시간이 길고, 데이터 테이블에서 다수의 레코드를 스캔합니다. 다수의 레코드를 읽기 때문에 보통 데이터 테이블의 데이터에 액세스할 때 인덱스를 사용할 필요가 없습니다. 대신, 쿼리는 전체 테이블을 스캔하려고 합니다. 이러한 쿼리에는 연속 방식으로 테이블을 스캔하지 않아도 되는 병렬 처리가 유용합니다.

병렬 쿼리를 생성하려면 데이터를 분할해야 합니다. 선택한 분할 스키마는 표현식과 관련된 스키마로, 해당 쿼리에서 일부 조각 세트를 제거할 수 있으므로, 쿼리 수행에 필요한 작업량을 줄여야 합니다. 분할화 표현식에는 테이블에 지정된 쿼리에서 계속적으로 사용되는 컬럼이 포함되어야 합니다. 앞의 예에서 slaes_facts 테이블을 사용할 때, 분할화 방법을 다음과 같이 개발해야 합니다.

```
create table sales_facts (
            salesman int,
            region char (5),
            sales_date date,
            sales_amount decimal (12,2) )

fragment by expression
            region in ("NW", "NE", "MA" ) in dbspace1,
            region in ("SE", "SN" ) in dbspace2,
            region in ("SW", "MW" ) in dbspace3;

- query to run against the sales fact table

select *
from sales_facts
where region in ("NW", "SW") and
```

```
            (sales_date > "01/01/1998" and
                 sales_date < "03/31/1998");
```

이 예에서, region 컬럼에 의해 데이터 테이블이 분할되었습니다. 테이블에서 데이터를 선택하는 쿼리는 필터 기준 내에서 region을 사용하지만, 필터 기준 내에서 sales_date를 사용하기도 합니다. sales_date가 부울과 작업의 일부이어서 영역 필터 기준과 sales_date 필터 기준을 모두 충족시켜야 하므로 엔진은 조각 제거를 계속 사용할 수 있습니다. 쿼리가 쿼리문에서 or 조건을 사용했으면, 다음과 같이 조각이 제거되지 않습니다.

```
select *
from sales_facts
where region in ("NW", "SW") or
            (sales_date > "01/01/1998" and
                 sales_date < "03/31/1998");
```

이 예에서, 테이블에서 검색한 각 행에 대해 쿼리 필터 조건을 모두 다 충족시켜야 합니다. region 기준으로 데이터를 분할했고 sales_dates 기준으로 조각을 제거할 수 없으므로, 모든 조각을 읽어야 합니다.

표현식에 의한 분할화의 단점은 여러 개의 조각에 균등하게 데이터를 분산시키는 방법을 찾기가 어렵다는 점입니다. 데이터가 균등하게 분산되지 않아 어떤 조각에는 다른 조각보다 데이터가 현저하게 더 많은 경우가 자주 발생합니다. 쿼리를 병렬로 실행하고 다중 조각을 스캔할 경우, 스캔 작업은 가장 오래 실행되는 스캔 스레드만큼 시간이 소요되므로 가장 큰 조각을 읽어야 하는 스캔 스레드가 작업의 병목 현상이 됩니다.

분할화 스키마에서 또 고려해야 할 사항은 IDS 엔진에 사용할 수 있는 CPU의 수입니다. 엔진에 사용할 수 있는 CPU가 네 개밖에 없으면, 병렬로 실행할 수 있는 스킨 스레드는 많아야 네 개입니다. 제거될 하나 또는 두 개의 조각이 있는 아홉 개의 조각에 대해 데이터를 분할하는 분할화 스키마를 사용하려면, 일곱 개의 스캔 스레드가 필요합니다. 병렬로 실행할 수 있는 스캔 스레드가 네 개밖에 되지 않으므로, 스캔 스레드가 일곱 개인 분할화 조각이 스레드가 네 개(엔진에서 사용할 수 있는 CPU 수와 동일)인 스키마보다 반드시 더 낫다고도 할 수 없습니다. 이러한 경우, 엔진에 사용할 수 있는 CPU의 수뿐만 아니라 사용할 수 있는 물리적 디스크와 해당 디스크의

입출력 처리량도 문제가 됩니다. 스캔 스레드가 입출력을 기다리는 경우가 많으므로, 물리 CPU의 수가 초과되면 사실상 처리량이 향상될 수 있습니다. 다양한 벤치마크 반복 실험을 통해 데이터베이스 인스턴스에 할당할 최대 CPU의 수와 균형이 맞는 최적의 분할화 스키마를 결정해야 합니다. (이러한 문제와 기타 성능 튜닝과 관련한 문제는 5장을 참조하십시오.)

보관/복구를 위한 분할화 방법

특정 테이블의 데이터를 디스크의 특정 DBSPACE에 분할함으로써, 테이블 데이터를 백업과 복구 목적에 맞도록 정렬할 수 있습니다. 그러나 이러한 방법이 성능 최적화에는 맞지 않을 수도 있습니다. 데이터베이스 관리자는 이러한 두 가지 접근 방식의 균형을 맞추어 최적화를 달성하고 유용한 백업과 복구 메커니즘도 제공할 수 있어야 합니다.

앞의 예에서 sales_fact 테이블을 사용할 경우, 현재 월의 데이터에만 일시적인 데이터가 들어 있고 이전 월은 상대적으로 정적인 정보를 보유하고 있으면, 관리자는 이전 월의 데이터를 주별 또는 월별로 백업하거나 일별로 현재 월의 데이터를 백업하려고 할 것입니다. Informix 백업 유틸리티와 복구 유틸리티에서 DBSPACE를 가장 작은 복구 객체로 사용하므로, 현재 월 데이터를 하나의 DBSPACE에, 이전 월 데이터나 하나 이상의 다른 DBSPACE에 배치하는 방법을 사용하여 데이터를 분할해야 합니다. 다음은 이러한 방법의 한 예입니다.

```
create table sales_facts (
    salesman int,
    region char (5),
    sales_date date,
    sales_amount decimal (12,2) )

fragment by expression
    sales_date > "03/31/1998" in dbspace1, - current month
    sales_date < "02/01/2998" and sales_date > "01/31/1998" in dbspace2,
       - prior months . . .
    sales_date < "01/01/1998" and sales_date < "02/01/1998" in dbspace3,
    sales_date < "01/01/1998" in dbspace4; - prior year
```

이러한 방법을 사용하게 되면 현재 월과 모든 이전 월이 개별 DBSPACE에 배치되고 예외적으로 이전 연도의 데이터만 단일 DBSPACE에 배치됩니다. 이렇게 되면 현재 월의 데이터는 밤마다, 이전 월의 데이터는 주별로 그리고 이전 연도의 데이터는 월별로 백업되는 세 개의 서로 다른 백업 계획을 구현할 수 있습니다.

이러한 방법이 반드시 성능 최적화 목표와 반대되는 것은 아닙니다. 쿼리가 sales_date에 따라 데이터를 검색할 경우, 분할 제거가 가능합니다.

기타 분할화 요인

어떤 경우에는, 성능 이외의 다른 요인에 따라서 데이터를 분산시킵니다. 앞서 언급한 대로, 보관은 이러한 요인 중 하나입니다. 또 다른 고려사항으로는 효과적인 데이터 공간 할당을 들 수 있을 것입니다. 데이터에 대해 실행 중인 쿼리에 따라 유용한 분할화 스키마를 개발할 수 없으면, 분할화 스키마를 중요한 고려사항으로 볼 수 없습니다. 예를 들어, 15개의 조각에서 고작 하나의 조각을 제거하는 표현식 기반 분할화 스키마는, 같은 테이블에 있는 라운드 로빈 분할화 스키마와 비교해 그다지 향상된 스키마라고는 볼 수 없습니다. 이러한 경우에는, 여러 조각 전반에 걸쳐 데이터를 균등하게 분산시키고 균등하지 못한 조각을 스캔하는 데 필요한 추가 쿼리 처리 시간을 없애주는 라운드 로빈 분할화 스키마가 더 나은 스키마일 수 있습니다.

데이터 분산 사용

Informix 엔진에는 데이터베이스 테이블에 저장된 데이터의 특성을 결정하는 유용한 방법이 있습니다. 이러한 정보는 데이터 분산이라는 형식으로 사용할 수 있으며 이는 데이터를 샘플링하는 통계 방법입니다. 데이터 분산에는 여러 행에 걸친 테이블의 컬럼에 저장된 값에 관한 정보가 들어 있습니다. 따라서 컬럼에 정수가 있으면, 컬럼에 있는 정수값과 특정값의 발생 수에 대한 샘플링이 해당 테이블의 데이터 분산에 제공됩니다. 이는 문자열 데이터의 경우에도 마찬가지입니다. 문자열 값과 해당 발생 수에 대한 샘플링이 해당 테이블의 출력에 표시됩니다.

데이터 분산은 매체 또는 높은 매개변수를 사용한 update statistics 명령으로 컴파일하고 저장한 정보를 사용합니다. 따라서 이러한 정보를 표시하기 전에 "update statistics" 명령을 실행해야 합니다.

dbschema 명령은 "update statistics" 명령으로 생성된 정보를 인쇄합니다. 통계 매체 업데이트 명령을 사용하여 통계를 생성하면, 테이블의 모든 데이터에 대해 실행되는 update statistics high 명령과는 달리 데이터의 샘플링만 사용됩니다. 적절한 쿼리 액세스 경로 결정이라는 이러한 정보의 목적에 비추어 볼 때, 통계 샘플링은 전체 테이블에서 데이터를 읽고 정렬하는 것만큼 유용하며, 정확도가 줄어든다고 해도 통계를 생성하는 이러한 방법을 사용하면 시간이 절약됩니다. dbschema 명령은 다음 형식을 사용하는 컬럼이나 테이블에 관한 데이터 분산 정보를 표시합니다.

```
dbschema -d <database name> -hd <table name>/ 'all' |
            <table name:column name>
```

데이터베이스 이름은 해당 테이블이 들어 있는 데이터베이스의 이름입니다. 테이블 이름을 사용하거나 테이블 이름과 컬럼 이름을 사용하여 사용할 테이블과 컬럼의 이름을 나타낼 수도 있고 또는 "all"을 인수로 사용하여 데이터베이스의 모든 테이블에 대한 데이터 분산 정보를 생성해야 한다고 표시할 수도 있습니다.

다음 예는 앞에 나온 sales_facts 테이블 예에 제품 ID와 판매 코드에 대한 차원 속성이 더 추가되었습니다. 확장된 사실 테이블의 스키마는 다음과 같습니다.

```
create table sales_facts (
       salesman_id   integer,
       Prod_id       char(5),
       sales_code    char(5),
       region        char(5)
sales_amt    decimal(12,2) ;
```

salesman_id는 판매 담당자의 고유 식별자입니다. pro_id는 판매된 제품을 식별하며, 제품 차원 테이블의 외부 키입니다. sales_code는 판매에 관한 정보 코드를 나타내며, 판매와 관련한 정보를 추적하는 추가 정보로서, 판매나 다른 목적을 생성할 때 사용한 판촉에 관한 것일 수 있습니다. sales_amt는 달러로 나타낸 판매금액

입니다. 다음은 이러한 사실 테이블의 샘플 출력입니다.

```
salesman_id   prod_id   sales_code   region   sales_amt
0             AA-331    A-OK         NW       1023.331
1             AB-931    B-1          SW       1605.28
```

다음 예는 이러한 사실 테이블에 대한 데이터 분산의 샘플 출력입니다. 분산은 테이블의 각 컬럼에 대해 준비됩니다. 다음 예의 경우, 첫 번째 분산은 sales_facts 테이블 컬럼 salesman_id에 대한 것입니다.

분산이 작성된 날짜가 열거되고, 다음 라인에 update statistics 레벨이 표시됩니다. 이 경우의 매체 모드는 해상도 2.5에 신뢰도가 95%입니다. 해상도는 출력에 열거된 각 빈이나 그룹화에 배치할 총 샘플의 비율을 나타냅니다. 따라서 샘플이 100행이고, 해상도가 0.1 또는 10%이면 bin이 10개인 출력이 생성됩니다. 다음은 분산의 출력 상태입니다.

```
Distribution for Informix.sales_facts. salesman_id
constructed on 03/14/1998
Medium Mode, 2.500000 Resolution, 0.950000 Confidence

- DISTRIBUTION -
     (                    0)
- OVERFLOW -
1:   (  30,               0)
2:   (  30,               1)
3:   (  30,               2)
4:   (  30,               3)
5:   (  30,               4)
6:   (  30,               5)
7:   (  30,               6)
8:   (  30,               7)
9:   (  30,               8)
10:  (  30,               9)
```

다음 절에는 사실 테이블의 판매 영역(차원)에 대한 분산이 열거되어 있습니다. 업데이트 통계 루틴에서 이 컬럼의 정보를 스캔할 때 Distinct 값을 찾지 못했으나 많은 수의 중복을 찾아냈습니다.

따라서 해당 정보는 분산 출력의 오버플로 섹션에 배치됩니다.

사실 테이블에 다섯 개의 Distinct 판매 영역이 있습니다. 이러한 값은 뒤에 오는 출력에 열거됩니다. 각 행에는 테이블 컬럼에 표시되는 값에 대한 통계가 들어 있습니다. 왼쪽의 출력 컬럼에는 bin 번호(이 경우에는 1에서 5까지)가 표시됩니다. 괄호의 첫 번째 컬럼에는 해당 행의 값이 샘플링에 표시되는 횟수가 표시됩니다. 맨 오른쪽에 있는 수는 행의 정보가 참조하는 값입니다. 예를 들어, 출력의 첫 번째 행에는 다음 정보가 들어 있습니다.

```
1: (   28,      AL   )
```

이것은 update statistics 루틴이 bin 번호 1에서 “AL” 값을 28개 찾아냈다는 것을 나타냅니다. 출력에 표시된 추가 행은 “MW” 값이 48번 발생, “NE” 값이 48번 발생, “SE“ 값이 128번 발생, “SW” 값이 48번 발생했다는 것을 나타냅니다.

```
Distribution for Informix.sales_facts.region
Constructed on 03/14/1998
Medium Mode, 2.500000 Resolution, 0.950000 Confidence
- DISTRIBUTION -

   (            AL   )

- OVERFLOW -

1: (   28,      AL   )
2: (   48,      MW   )
3: (   48,      NE   )
4: (  128,      SE   )
5: (   48,      SW   )

Distribution for informix.sales_facts.prod_id

Constructed on 03/14/1998

Medium Mode, 2.500000 Resolution, 0.950000 Confidence

- DISTRIBUTION -
```

```
   (             AA-123  )

- OVERFLOW -

1: (  156,       AA-123  )
2: (   24,       AA-331  )
3: (   24,       AB-931  )
4: (   48,       BB-321  )
5: (   48,       XX-323  )

Distribution for informix.sales_facts.sales_code

Constructed on 03/14/1998

Medium Mode, 2.500000 Resolution, 0.950000 Confidence

- DISTRIBUTION -

   (              A-OK   )

- OVERFLOW -
1: (    28,       A-OK   )
2: (   100,       A1     )
3: (    48,       B-1    )
4: (    28,       B-OH   )
5: (    48,       H-1    )
6: (    24,       Z-5    )
7: (    24,       Z-6    )
```

다음 절은 sales_amt 컬럼에 대한 값이 넓게 분산되어 있음을 나타냅니다. 판매 총액은 판매별로 아주 다양합니다. 총 300개의 판매 레코드에서, 업데이트 통계 루틴이 sales_amt 컬럼에 대해 70개의 Distinct 값을 찾아냈습니다. (이 경우에, 반복이 나타나는 것은 n*100.33*mod(n,5) 컬럼 값(여기서 n은 반복 계수임)을 생성할 때 사용하는 공식 때문일 수 있습니다.)

```
Distribution for informix.sales_facts.sales_amt

Constructed on 03/14/1998
```

```
Medium Mode, 2.500000 Resolution, 0.950000 Confidence

-- DISTRIBUTION -

    (                  0.00   )

- OVERFLOW -

1:  (    60,           0.00   )
2:  (     3,         100.33   )
3:  (     3,         401.32   )
4:  (     3,         601.98   )
5:  (     3,         902.97   )
6:  (     3,        1103.63   )
7:  (     3,        1404.62   )
8:  (     6,        1605.28   )
9:  (     3,        2106.93   )
10: (     6,        2407.92   )
. . . .
59: (     3,       24982.17   )
60: (     3,       25684.48   )
61: (     3,       26487.12   )
62: (     3,       27691.08   )
63: (     3,       27992.07   )
64: (     3,       29497.02   )
65: (     3,       29697.68   )
66: (     3,       31704.28   )
67: (     3,       33710.88   )
68: (     3,       35717.48   )
69: (     3,       37724.08   )
70: (     3,       39730.68   )
}
```

분산 정보를 사용하여 분할화 방법 결정

'dbschema -hd' 옵션을 사용하여 생성되는 분산 정보에는 표현식 기반 분할화 방법을 개발하는 데 있어 대단히 가치 있는 정보가 있습

니다. 디스크 및/또는 디스크 제어기 전체에 정보를 균등하게 분산시키는 것이 목적이면 데이터 값을 분산시키는 작업을 이해하는 것이 아주 중요합니다.

앞의 예에서, 영역 키를 분할화 표현식에 사용할 컬럼으로 간주했으면 분산 출력에 표시된 정보를 보고 이러한 방법의 적합성 여부를 확인할 수 있습니다.

분산 출력에 표시된 대로 영역 컬럼에 다섯 개의 Distinct 값이 있으므로 이 컬럼에만 데이터를 분할하는 데 최대 다섯 개의 조각을 사용할 수 있습니다. 각 조각에 배치한 값의 수는 다음과 같습니다.

계수	영역 값
28	AL
48	MW
48	NE
128	SE
48	SW

그러나 영역 컬럼의 값만을 기준으로 한 방법을 사용하면 데이터를 완전히 균등하게 분산시킬 수 없습니다. “SE”에 일치하는 영역 값을 보유하는 조각에는 행이 128개이며 이는 다른 조각이 가진 행 수의 두 배가 넘습니다. 해당 데이터에 대한 액세스가 다섯 개의 영역 전반에 균등하게 분산되었다고 결정한 경우에는 문제가 되지 않을 수도 있습니다. 이 경우에, 균등하지 않은 것은 데이터 스토리지뿐이며 이는 “SE” 영역을 보유한 조각에 디스크를 더 추가함으로써 관리할 수 있습니다. 이러한 방법의 분할화 스키마는 다음과 같습니다.

```
. . . fragment by expression

            region = “AL” in dbspace1,
            region = “MW” in dbspace2,
            region = “NE” in dbspace3,
            region = “SE” in dbspace4,
            region = “SW” in dbspace5;
```

디스크 공간에 제한이 있는 경우, 둘 이상의 영역을 단일 DBSPA

CE에 조합하여 더 적은 수의 디스크에 데이터를 분산시키는 것이 가장 좋을 수도 있습니다. 다음과 같이 분산 정보를 사용하여 더 적은 값의 영역을 조합할 수도 있습니다.

```
. . . fragment by expression

        region in ("AL", "MW" ) in dbspace1,
        region in ("NE", "SW" ) in dbspace2,
        region = "SE" in dbspace3;
```

이러한 방법을 사용하면 76(28+48)개의 행이 dbspace1에, 96(48+48)개의 행이 dbspace2에, 128개의 행이 dbspace3에 배치됩니다. 이러한 방법을 사용해도 데이터를 완전 균등하게 분산시킬 수는 없지만, 앞의 첫 번째 방법보다는 더 균등하게 분산시킬 수 있고 필요한 DBSPACE(조각)의 수를 셋으로 줄일 수 있습니다.

시간에 따라 다양한 데이터 분산

'update statistics' 명령을 실행하여 생성한 통계는 스냅샷이며, 시간의 한 지점 즉, 분산 출력에 확인된 시점을 나타냅니다. 계속해서 데이터를 데이터베이스에 추가하면, 분산은 변경됩니다. 어떤 경우에는 데이터 분산의 변화가 상당히 없어서, 3월에 생성된 분산이 6월에도 유효할 수도 있습니다. 그러나 데이터베이스 관리자는 계속해서 분산을 생성하고 검토하여 새 분산 방법이 필요한지를 결정해야 합니다.

닭이 먼저? 알이 먼저?

"닭이 먼저인가, 알이 먼저인가?"하는 논쟁과 유사한 경우로도 볼 수 있겠지만, 분산을 실행하여 테이블의 데이터를 분할하는 방법을 결정하기 전에 테이블을 작성하고 로드해야 합니다. 그러면 관리자는 두 부분으로 된 작업을 수행합니다. 먼저, 관리자는 테이블을 로드하고 테이블 데이터에 분산을 실행합니다. 분할화 스키마를 결정했으면 데이터를 언로드하고 분할화 스키마로 테이블을 재작성한 후 데이터를 재로드합니다. 데이터를 테이블에 두고 "alter fragment"문을 사용하는 방법도 있지만, 이러한 방법을 사용하여 완료하려면 상당히 많은 양의 디스크 공간이나 로그 공간이 필요합니다.

데이터 분산을 검토한 내용에 따라 새로운 분할화 방법이 필요하면, alter fragment문을 사용하여 데이터의 분할화 방법을 변경할 수 있습니다. alter fragment문을 사용하면 기존 조각은 제 자리에 남아 있고 새 조각(DBSPACE)에 데이터를 써야 합니다. 큰 테이블의 경우, 이 작업에는 수정을 완료하는 데 상당한 양의 디스크 공간이 필요할 수 있습니다.

디스크 사용도 모니터링

IDS 데이터베이스 엔진은 익스텐트 추가라는 프로세스를 통해 계속해서 필요한 공간을 테이블에 추가합니다. 프로세스는 DBSPACE에 더 이상 익스텐트를 추가할 공간이 없을 때까지 계속됩니다. 이러한 시점이 되면 테이블에 대한 어떤 쓰기도 실패하고, 해당 SQL 문이 중단되지만, 시스템은 온라인 상태이므로 데이터베이스 관리자는 청크를 추가하여 공간을 DBSPACE에 추가하거나 또는 현재 DBSPACE에 있는 데이터를 제거하고 공간을 사용 가능하게 하여 문제를 처리해야 합니다. (현재 DBSPACE에 있는 테이블을 삭제해야만 데이터를 제거할 수 있으며, 기존 테이블에서 데이터를 삭제한다고 해도 DBSPACE에 사용 가능 공간이 추가되지는 않습니다.)

-d 옵션으로 onstat 명령을 실행하면 데이터베이스의 DBSPACE의 목록과 그 안에 남은 공간이 나열됩니다. 이 명령을 실행하여 얻은 출력은 다음과 같습니다.

```
*INFORMIX-OnLine Version 7.22.WC1  - On-Line - Up 00:01:07 - 9024 Kbytes

Dbspaces
address   number   flags   fchunk   nchunks   flags     owner     name
831980ec    1        1       1         1       N       Informix   rootdbs
1 active, 2047 maximum

Chunks
address   chk/dbs  offset  size  free  bpages  flags    path name
83198158   1   1      0    2120   33            -PO     rootdbs_dat.000
1 active, 2047 maximum
```

이러한 명령의 출력은 두 부분으로 되어 있습니다. 명령의 첫 번째

섹션에는 시스템에 제시된 DBSPACE에 관한 정보가 표시됩니다. 이 섹션의 출력에 나온 첫 번째 컬럼에는 공유 메모리의 DBSPACE 정보 위치를 나타내는 내부 번호인 DBSPACE 주소가 표시됩니다. 다음 컬럼에는 DBSPACE 번호가 표시됩니다. 이 DBSPACE 번호는 출력의 다음 섹션에 있는 DBSPACE를 참조할 때 사용됩니다. 다음 컬럼에는 이 DBSPACE의 플래그가 표시됩니다. 이러한 플래그는 16진수이며 그 의미는 다음과 같습니다.

값	설명
0x0001	미러 없음
0x0002	미러
0x0004	다운
0x0008	새로 미러됨
0x0010	BLOBSPACE

`fchunk`라는 제목의 다음 컬럼은 DBSPACE에 있는 첫 번째 청크의 번호입니다. 앞 예에서, 이 컬럼은 DBSPACE의 초기 청크로서 DBSPACE에서 삭제할 수 없습니다. 다음 컬럼에는 DBSPACE와 관련된 플래그 세트가 나오며 플래그 각각의 의미는 다음과 같습니다.

위치	설명
위치1	M-미러됨 N-미러되지 않음
위치2	X-새로 미러됨 P-물리적으로 복구되어 논리 복구 대기 중 L-논리 복구 중 R-복구 중
위치3	B-BLOBSPACE

마지막 두 개의 컬럼은 DBSPACE의 소유자(DBSPACE를 작성한 사용자)와 DBSPACE에 지정된 이름을 나타냅니다.

```
...
Chunks
address   chk/dbs  offset  size  free  bpages  flags     path name
83198158   1   1      0     2120   33            -PO      rootdbs_dat.000
1 active, 2047 maximum
```

위에 표시된 대로 출력의 마지막 섹션에는 시스템의 DBSPACE를 구성하는 청크에 관한 정보가 표시됩니다. 청크 정보는 기존의 청크에 사용되는 공간과 관련됩니다. 첫 번째 컬럼은 공유 메모리의 청크 정보 주소입니다. 다음 컬럼에는 행에 표시된 청크 번호와 DBSPACE 번호가 들어 있습니다. DBSPACE 번호는 첫 번째 섹션의 DBSPACE 번호를 참조합니다. 다음 컬럼은 장치에 대한 청크의 오프셋(같은 라인의 경로 이름 출력에서 참조)입니다. 청크의 크기는 다음 컬럼에 표시되며, 그 다음에 청크의 사용 가능한 바이트가 표시됩니다. 다음 컬럼에는 청크의 BLOBPAGE가 표시되고 그 다음 컬럼에는 청크의 상태를 나타내는 16진 플래그가 표시됩니다. 16진 플래그는 다음과 같습니다.

위치	설명
위치1	설명 P- 기본 M- 미러
위치2	O- 온라인 D- 다운 X- 새로 미러됨 I - 불일치
위치3	B- BLOBSPACE - - DBSPACE T-임시 DBSPACE

마지막 컬럼에는 청크를 나타내는 장치의 경로 이름이 표시됩니다. 앞에 표시된 출력에서, 청크의 크기는 2120K로 이 중 33K가 사용 가능하며, 이 크기가 현재 온라인 상태인 기본 청크입니다.

데이터베이스 관리자는 테이블에 대한 디스크 사용 상태를 면밀하게 모니터해야 합니다. onstat 명령은 테이블의 공간 사용도를 보고하지 않으며 청크에 의한 공간 사용도를 보고한다는 점에 유의하

십시오. 따라서 테이블에 대한 공간 사용 상태를 식별하는 최상의 경로는 테이블이 상주하는 청크를 식별하고 해당 청크의 공간 사용 상태를 추적하는 것입니다. 이 정보는 다음 구문과 같이 oncheck 명령을 사용하여 알 수 있습니다.

```
oncheck -pt <database_name:table_name>
```

다음 명령을 실행하면 stores7 데이터베이스의 테이블 순서 목록이 생성됩니다.

```
oncheck -pt stores7:orders
```

이 명령으로 생성한 출력의 일부가 테이블의 조각이 위치한 DBSPACE를 식별합니다. 또한 출력에는 테이블에 할당된 행 수와 현재 사용되는 행 수에 관한 유용한 정보가 표시됩니다. (사용된 행 수를 알고 있으면 유용하지만, 마지막 익스텐트에 남아 있는 행 수가 아니라 마지막 조각의 크기와 다음 익스텐트에 사용할 수 있는 공간이 테이블에 대한 쓰기를 제한할 수 있다는 점에 유의하십시오.)

이러한 청크가 식별되면, 테이블의 마지막 청크에서 사용할 수 있는 공간을 결정해야 합니다.

DBSPACE의 테이블 크기가 항상 테이블의 행 수를 반영하는 것은 아닙니다. 일련의 익스텐트로 테이블을 작성하기 때문에 테이블에서 차지하는 물리 공간은 테이블의 행 수가 차지한 공간과 같거나 그 공간을 초과합니다. 따라서 테이블의 레코드를 삭제해도 계속해서 테이블이 할당된(익스텐트의 크기) 공간을 사용하기 때문에 DBSPACE의 테이블이 차지하는 공간은 줄어들지는 않습니다.

onspaces -a 명령을 사용하여 DBSPACE에 공간을 추가할 수 있습니다. 이 명령을 실행하면 다음과 같이 기존의 DBSPACE에 청크가 추가됩니다.

```
onspaces -a dbspace1 -p /dev/rdsk1 -o 0 -s 5000
```

이 명령에 의해 원시 디스크 장치 /dev/rdsk1에서 DBSPACE dbspace1로 5MB의 청크가 추가됩니다.

DBSPACE의 디스크 입출력을 모니터링하는 것이 중요합니다. 입출력 경쟁을 방지하려면, 읽기와 쓰기 작업을 가능한 한 많이 분산시켜야 합니다. -D 옵션과 함께 onstat 명령을 실행하면 데이터베이스 인스턴스의 DBSPACE, DBSPACE에 해당되는 청크, 청크의

입출력 작업 등이 나열됩니다. 다음은 이러한 명령을 실행하여 얻은 출력 샘플입니다.

```
INFORMIX-OnLine Version 7.22.WC1  - On-Line - Up 00:05:24 - 9024 Kbytes

Dbspaces
address   number   flags   fchunk   nchunks   flags    owner      name
82efa0ec    1        1       1         1        N      Informix   rootdbs
1 active, 2047 maximum

Chunks
address    chk/dbs   offset   page Rd   page Wr         path name
82efa158    1   1      0      276        705
D: \IFMXDATA\ol_ataylor_homepc\rootdbs_dat.000
1 active, 2047 maximum
```

이 출력에는 시스템의 다양한 청크에 대한 페이지 읽기 및 쓰기 정보가 표시됩니다. 그러나 이 정보는 DBSPACE 레벨이 아니라 청크별 기준 정보입니다. 출력의 첫 번째 부분에는 시스템의 DBSPACE, DBSPACE를 참조할 때 사용하는 내부 번호를 나타내는 번호, DBSPACE의 청크 수를 나타내는 컬럼 및 청크의 상태를 나타내는 플래그가 표시됩니다. 마지막으로, 맨 끝의 두 컬럼은 DBSPACE의 소유자와 DBSPACE에 지정된 이름을 나타냅니다.

표시된 두 번째 부분은 이전 절에 열거된 DBSPACE를 구성하는 청크에 관한 정보입니다. 청크의 내부 주소가 표시되고 청크 번호와 청크의 DBSPACE 번호를 나타내는 컬럼이 있습니다. 또 다른 컬럼은 장치에 대한 청크의 오프셋을 나타냅니다. page Rd와 page Wr의 컬럼은 해당 청크의 페이지 읽기와 쓰기 작업에 관한 중요한 정보입니다. 이러한 계수는 `onstat -z` 명령을 사용하거나 엔진이 시작된 공유 메모리를 초기화하여 마지막으로 계수를 0으로 설정한 때와 관련이 있습니다.

이 외에도, `-g iof` 매개변수와 함께 `onstat` 명령을 사용하여, 청크별 입출력을 출력할 수도 있으며, 정렬 작업 파일과 임시 파일에 대한 입출력도 표시됩니다.

```
INFORMIX-OnLine Version 7.22.WC1 -- On-Line - Up  00:02:50 - 9024 Kbytes
AIO global files:gfd   path name   totalops   dskread   dskwrite  io/s
  3  *D: \data\rootdbs_dat.000        25         19         6      0.1
```

이 명령을 실행하면 각 청크의 경로 이름과 해당 청크의 입출력 통계가 표시됩니다. totalops 컬럼의 청크에 대한 총 작업, dskread 컬럼의 디스크 읽기 작업 수 및 dskwrite 컬럼의 쓰기 작업 수가 표시됩니다. 마지막 컬럼에는 청크에 대한 초당 입출력이 표시되며 이는 해당 청크 입출력의 효율성을 나타내는 훌륭한 지표입니다.

물리/논리 디스크 레이아웃의 목표는 시스템을 구성하는 디스크 전체에 디스크 입출력을 균등하게 분산시키는 것입니다. 디스크의 한 분할 영역에서 대량의 디스크 입출력을 수신하면, 시스템의 활동이 많은 기간 중에는 해당 디스크에서 입출력 경쟁이 발생할 수 있습니다.

입출력의 균등 분산이 중요한 목표이기는 하지만, 시스템을 설정하거나 튜닝할 때 고려하는 요소는 이 한 가지만이 아닙니다. 다른 요인도 시스템 성능에 영향을 줄 수 있습니다. 디스크의 물리 로그나 논리 로그로 인해, 또는 특정 디스크에서 일련의 데이터 스캔을 수행하기 위해 하루에 한 번씩 실행되는 보고서로 인해, 입출력이 균등하지 않게 분산될 수도 있습니다. 이러한 문제는 특정 테이블에 데이터를 분산시키는 방법보다는 다른 솔루션으로 해결할 수 있습니다. 예를 들어, 과도한 로그 활동으로 인한 문제는 논리 로그 파일을 다른 DBSPACE에 이동하여 처리할 수 있습니다. 과도한 보고서 활동으로 발생한 문제는 OLTP 활동이 최소화되는 밤사이에 보고서를 실행하여 관리할 수 있습니다.

시스템을 튜닝하려면 여러 가지 요인을 살펴보아야 합니다. 이러한 요인으로는 시스템에서 실행되는 응용프로그램, 사용 가능한 하드웨어 자원, 사용자 로드 및 액세스가 온라인인지 배치인지 아니면 둘의 조합인가의 여부 등이 있습니다. 이러한 모든 요인은 시스템을 튜닝하는 방법을 결정할 때 가중되어야 합니다. 이 장에서는 이러한 몇 가지 주제에 대해 간략하게 소개하였으며, 튜닝에 대해서는 5장에서 더 자세하게 설명됩니다.

요약

이 장에서는 IDS 엔진이 논리 레벨과 물리 레벨에서 디스크와 어떻게 상호 작용을 하는지에 대해 살펴보았습니다. IDS 엔진은 물리 레벨에서 청크의 디스크에 할당되며 이는 디스크 장치 및/또는 파일에 직접 매핑됩니다. 논리 레벨에서 엔진은 청크 컬렉션을 DBSPACE로 봅니다. DBSPACE에 데이터베이스가 작성되고, 데이터베이스에서

테이블이 작성되고 처리됩니다.

여러 가지 도구를 사용하여 IDS 엔진의 디스크를 관리할 수 있습니다. 명령행 도구를 모든 관리 기능을 수행하는 데 사용할 수 있으며 실제적으로 모든 데이터베이스 유지보수 테스크에 대한 쉘 스크립트를 작성할 수 있습니다. 또한, onmonitor와 같은 메뉴 구동의 문자 기반 도구와 Space Explorer 및 Informix Enterprise Command Center와 같은 GUI 도구를 사용하여 데이터베이스를 유지보수하고 모니터링할 수 있습니다.

IDS 엔진을 사용하여 테이블의 데이터를 다중 DBSPACE에 분산시킬 수 있으며 이러한 프로세스를 분할화라고 합니다. IDS 옵티마이저는 데이터의 분할화를 이해하며, 분할화를 통해 쿼리를 처리하는 동안 다중 조각의 병렬 스캔을 수행합니다. 또한 분할화를 사용하여 백업과 복구 목적으로 DBSPACE 전반에 데이터를 분산시킬 수 있습니다.

onstat 명령과 데이터 분산 등의 다양한 기능을 사용하여 디스크를 모니터링할 수 있습니다. onstat 명령은 DBSPACE에 대한 디스크 사용 및 DBSPACE를 구성하는 청크를 제공합니다. dbschema 명령을 사용하면 테이블에 저장된 데이터의 분산을 출력할 수 있습니다. 이러한 도구를 사용하여 테이블의 분할화 스키마와 관련된 정보에 따라 의사 결정을 내릴 수 있습니다.

다음 장 미리보기

다음 장에서는 이 장에서 설명한 내용을 확장하여 IDS 데이터베이스의 모니터링하고 튜닝하는 방법에 대해 논의합니다. onstat 명령과 확장 출력 옵션을 설명하고 이러한 정보를 사용하여 Informix 데이터베이스를 튜닝하는 방법에 대한 팁을 제공합니다. 엔진이 온라인 상태일 때 onmode 명령을 실행하여 튜닝 가능한 많은 엔진을 처리함으로써, 반복적인 벤치마크가 실행되는 동안 신속하게 매개변수를 변경할 수 있습니다.

onstat 명령의 출력을 검토하고 적절한 구성 매개변수를 변경하여 다양한 튜닝 방법을 구현함으로써 IDS 엔진의 전반적인 성능과 효율성을 높일 수 있습니다. 다음 장에서는 이 여러 가지 튜닝 방법에 대해 논의합니다.

IDS 엔진 모니터링 및 튜닝

5장

Informix는 데이터베이스 IDS 엔진을 모니터하고 튜닝하는 많은 도구를 제공합니다. 모니터링 도구를 사용하여 데이터베이스 작업의 많은 측면을 모니터하고 제어할 수 있습니다. 전체 시스템을 통해 개별 쿼리의 진행 상태를 추적할 수 있고 시스템의 작업 전체를 모니터할 수도 있습니다.

이 장에서는 IDS 데이터베이스 성능상의 문제를 파악하고 이를 해결하는 방법에 대해 설명합니다. 4장에서 제시된 구성 매개변수를 사용한 IDS의 모니터링 및 튜닝 과정도 설명됩니다.

IDS를 위한 튜닝 방법

IDS 엔진을 튜닝하는 것은 그리 간단하고 직접적인 프로세스가 아닙니다. 마법의 탄환처럼 모든 플랫폼의 응용프로그램 모두에 적용되어 한 번에 간단하게 해결되는 튜닝 솔루션은 없습니다. 그러나 플랫폼 자원의 가용성과 응용프로그램의 필요에 따라 적용되어야 하는 방법은 있습니다. 데이터베이스를 튜닝하는 관리자는 시스템에 영향을 줄 수 있는 기타 요인 뿐 아니라 실행 중인 응용프로그램, 사용 가능한 플랫폼 및 자원, 머신의 로드를 평가해야 합니다.

많은 요인이 시스템 성능에 영향을 줄 수 있습니다. 낮은 성능은 CPU 또는 디스크와 같은 일부 자원의 한계가 원인이 될 수 있습니다. 그러나 낮은 성능은 또한 쿼리의 특성이나 실행 중인 쿼리와 직접 관련이 될 수도 있습니다. 특정 쿼리는 디스크 테이블에서 데이터

검색을 시도할 때 테이블의 모든 레코드 검색, 즉 테이블의 연속 스캔을 요구합니다. 이 문제의 솔루션은 전체 테이블 스캔을 방지하는 인덱스를 작성하는 것입니다.

일반적으로 시스템을 튜닝하려는 경우 우선 사용 가능한 자원이 부담을 받는지 여부를 결정해야 합니다. 특정 디스크에서 디스크 입출력이 초과되었는지, 입출력 채널의 한계에 근접하고 있는지, 또는 CPU 자원이 제한되었는지 확인하십시오.

엔진 자원에 대한 일반적인 제한 요인은 CPU입니다. IDS 엔진에서는 충분한 CPU 자원이 효율적으로 작동되어야 하며 이 자원의 가용성은 엔진의 튜닝을 통해 평가되어야 합니다.

CPU 사용이 100%에 근접하고 있습니까? 여러 CPU를 사용할 수 있는 경우(SMP 플랫폼에서와 마찬가지로) 모든 CPU에서 시스템 로드가 균형을 이루고 있습니까? 엔진에서 사용 가능한 모든 CPU를 사용하고 있습니까? 불충분한 CPU 자원은 IDS 엔진 성능에 심각한 영향을 줄 수 있으며 주로 시스템 성능의 병목 현상을 가져옵니다.

튜닝의 첫 단계는 CPU나 디스크 처리량과 같은 사용 가능한 머신의 자원과 관련된 IDS 데이터베이스의 성능을 평가하는 것입니다. 많은 onstat 명령과 몇몇의 운영 체제 명령을 사용하여 자원 사용을 결정하고 잠재적 병목 현상을 파악할 수 있습니다. 이러한 명령은 시스템의 유휴시간 동안(IDS 시스템이 온라인이지만 시스템을 사용하는 사용자가 없을 때) 그리고 시스템이 작동 중이고 사용자 활동이 전체 로드를 하고 있을 때 실행되어야 합니다.

시스템의 튜닝에는 또한 데이터베이스에 대해 실행 중인 응용프로그램의 평가가 포함되어야 합니다. 쿼리가 느리게 실행되는 원인은 시스템을 제한하는 자원 때문이 아니라 실행되고 있는 쿼리 때문입니다. 쿼리가 잘못 작성되어 대형 테이블을 스캔해야 할 수도 있고 스캔해야 하는 레코드 수를 줄이기 위하여 인덱스가 필요할 수도 있습니다. 때로는 데이터에 대해 더 빠른 액세스 경로를 제공하기 위해 관계형 테이블의 구조를 변경해야 합니다.

간단히 말해, 낮은 성능에 대한 모든 잠재적인 문제 영역을 살피는 균형 잡힌 접근이 데이터베이스 튜닝에 대한 최선의 접근 방법입니다. 잠재적 문제 영역으로는 머신 자원의 병목 현상, IDS 구성 매개변수, 데이터베이스의 논리 구조 및 물리 구조, 쿼리 및 데이터베이스에 대해 실행 중인 응용프로그램 및 데이터베이스의 사용자 로드 등이 있습니다. 다음 절에서는 이러한 문제 영역에 대해 살펴보고 존재할 수 있는 문제에 대한 솔루션을 제안합니다.

onstat 유틸리티를 사용하여 IDS 모니터링

onstat 유틸리티를 사용하여 IDS 데이터베이스의 다양한 작업에 대한 통계 정보를 검색할 수 있습니다. onstat 명령은 문자 기반이며 명령행에서 실행됩니다.

이 유틸리티의 많은 옵션은 초기 제품 릴리스부터 사용되었으며 현재에도 유효합니다. 6.x와 7.x 엔진 제품의 다중 스레드 기능 외에도, 점점 증가하는 엔진의 복잡성과 엔진의 모니터링에 사용할 수 있는 정보를 수용하기 위하여 -g 매개변수 아래에 서브 옵션 세트가 작성되었습니다. 사용 가능한 onstat 전체 세트 옵션이 표 5-1에 제시되어 있습니다. 이러한 옵션 중 몇 가지는 불명확하기도 하며, 매우 한정된 모니터링 작업(성능 문제의 추적 또는 엔진 디버깅)에 유용한 정보만 포함하고 있습니다. 이 장에서는 데이터베이스의 모니터링 및 튜닝에 유용한 옵션만을 논의하겠습니다.

Informix 서버의 버전 6.0 릴리스부터 멀티스레딩과 메모리의 내부 관리와 같은 많은 중요한 기능이 엔진에 추가되었습니다. 이 기능을 모니터할 수 있도록 onstat 유틸리티에 -g 옵션 및 해당 매개변수가 추가되었습니다. 표 5-2는 -g 옵션에 대한 매개변수의 목록입니다.

표 5-1 onstat 유틸리티의 옵션 및 매개변수

옵션	목적
C	B 트리 클리너 요청
B	현재 할당된 모든 버퍼(사용 여부에 관계없음)
b	현재 사용 중인 모든 버퍼
X	<버퍼> 대기 스레드의 주소를 포함하는 버퍼
h	버퍼 해시 체인 정보
c	구성 파일 정보 표시
f	데이터 스킵 옵션 정보
d	청크 및 DBSPACE에 대한 정보
D	DBSPACE 청크 및 청크별 페이지 읽기 및 쓰기에 대한 정보 표시
i	onstat를 대화형 모드에 저장
s	래치 정보 표시

<계속>

표 5-1 onstat 유틸리티의 옵션 및 매개변수

옵션	목적
k	보유 로크 표시
l	페이지 주소를 포함한 물리 및 논리 로그에 대한 로깅 정보 표시
R	LRU 큐 표시
g	모니터링 정보 표시
m	IDS 메시지 로그 표시
p	시스템 통계 정보 표시
O	IDS 또는 온라인 광 메모리 캐시 정보 및 스테이징 영역 BLOBSPACE 정보표시
r <seconds>	지정된 시간(초)만큼 onstat 명령 반복
o <file name>	공유 메모리 세그먼트를 지정된 파일에 출력
a	사용자 중심 옵션의 요약 표시
t	활성 TABLESPACE 출력
x	트랜잭션 정보 출력
u	사용자 스레드 및 트랜잭션 표시
F	페이지를 플러시할 때 수신한 페이지 쓰기 통계 표시
i	onstat를 대화형 모드로 저장
z	모든 통계 카운터를 영(0)으로 설정

표 5-2 onstat -g 매개변수

옵션	목적
act	활성 스레드 인쇄
afr<pool name> |<session id>	세션 또는 지정된 메모리 풀에 할당된 메모리 조각
all	다중 스레딩 정보 표시
ath	모든 스레드 표시
con	조건 및 대기자 인쇄
dic<table>	매개변수 없이 공유 메모리에 캐시로 저장된 각 테이블마다 한 라인의 정보 인쇄. 테이블 매개변수와 함께 사용하면 지정된 테이블에 대한 정보만 표시
dri	데이터 복제 정보
dsc	데이터 분산 캐시
ffr <pool name> |<session id>	공유 메모리 풀 또는 지정된 세션에 사용 가능한 조각 표시

<계속>

표 5-2 onstat -g 매개변수

옵션	목적
glo	전역 다중 스레딩 정보 표시
iob	입출력 가상 프로세서 클래스별로 big-buffer 사용 인쇄
iog	비동기 입출력 전역 정보
ioq	비동기 입출력 큐 통계
iov	가상 프로세서별 비동기 입출력
lmx	로크된 모든 뮤텍스 표시
mem <pool name> |<session id>	풀 또는 지정된 세션에 대한 메모리 통계 인쇄
mgm	MGM(Memory Grant Manager) 자원 정보 표시
nbm	상주하지 않는 세그먼트에 대한 블록 비트맵 표시
nsc <client id>	클라이언트 ID별 공유 메모리 상태 표시
nsd	폴 스레드에 대한 네트워크 공유 메모리 데이터 표시
nss <session id>	세션 ID별 공유 메모리 상태 표시
ntd	서비스별 네트워크 통계
ntm	네트워크 메일 통계
ntt	네트워크 사용자 시간
ntu	네트워크 사용자 통계
pos	infos 및 DBSERVERNAME 파일 인쇄
ppf <partition> | 0	파티션 번호에 대한 파티션 프로파일 인쇄. 0은 모든 파티션의 프로파일을 인쇄
prc	내장 프로시저 정보 인쇄
qst	큐 통계 인쇄
rbm	상주 세그먼트에 대한 블록 비트맵 인쇄
rea	준비된 스레드 인쇄
sch	세마포어 작업 수 인쇄
seg	공유 메모리 세그먼트 통계 인쇄
ses <session id>	세션 ID별 세션 정보
sle	휴면 중인 스레드 인쇄
spi	스핀 로크 인쇄
sql <session id>	세션 ID별 SQL 정보 인쇄
stl <tid>	특정 스레드의 스택 덤프

<계속>

표 5-2 onstat -g 매개변수

옵션	목적
sts	스레드별 최대 및 현재 스택 사용 표시
fpf <tid>	tid에 대한 스레드 프로파일 표시. 0은 모든 스레드에 대한 프로파일을 인쇄
ufr <pool name> |<session id>	사용별로 할당된 조각 표시
wai	대기 스레드 인쇄. 조건, 뮤텍스 또는 조건 대기 중
wmx	모든 뮤텍스 및 대기자 인쇄
wst	대기 통계 인쇄

onstat 명령은 명령행에서 실행됩니다. 명령을 실행하면 결과는 명령을 실행하는 터미널 장치(stdout)로 리턴됩니다.

매개변수를 사용하지 않고 onstat 명령을 실행하는 경우 onstat 명령은 모든 사용자 스레드 및 프로파일 통계를 나열하면서 onstat -up 명령의 출력을 표시합니다. 위의 특정 매개변수 세트를 사용하여 실행하는 경우, onstat 명령은 해당 매개변수에 대한 출력을 생성합니다. 다음 명령은 현재 서버에 대한 프로파일 통계를 표시합니다.

```
onstat -p
```

청크 및 DBSPACE에 대한 정보를 표시하려면, 다음 명령을 실행하십시오.

```
onstat -d
```

onstat 명령을 실행하면 옵션을 입력 매개변수로 지정하여 다수의 옵션이 함께 실행될 수 있습니다. 다음 명령은 현재 데이터베이스 인스턴스에 대한 사용자 스레드(-u) 및 테이블 정보(-t)를 표시합니다.

```
onstat -ut
```

Informix 모니터링 도구는 엔진 및 사용자 스레드의 활동에 대한 풍부한 정보를 제공합니다. 이 출력을 이해하는 것은 쉬운 일은 아닙니다. 다음 절에서는 엔진의 모니터링 및 튜닝을 위한 onstat 출력의 평가 및 해석에 대한 방법을 제시합니다.

onstat 유틸리티를 사용하여 자원 병목 현상 식별

엔진에는 운영 중인 머신으로부터 디스크와 메모리의 두 가지 기본 자원이 필요합니다. onstat 유틸리티와 몇 가지 운영 체제 유틸리티는 데이터베이스 엔진에 적용 중인 자원을 모니터링하는 방법을 제공합니다. onstat 유틸리티는 엔진의 디스크 활동, 디스크 읽기 및 쓰기 횟수, 논리 및 물리 로그에 대한 로그 활동을 표시하는 데 사용할 수 있습니다. 또한 이 유틸리티는 공유 자원 경쟁, 보유 중인 로크 수, 스레드가 로크를 기다리고 있는지 여부, LRU 큐의 활동 및 다른 경쟁을 검색하기 위해 사용되기도 합니다. 검토할 엔진 자원과 이러한 자원을 검토하는 데 사용되는 onstat 명령이 표 5-3에 나열되어 있습니다. 다음 절에서는 엔진을 튜닝하기 위해 명령을 사용하는 프로세스에 대해 다룹니다.

표 5-3 onstat를 사용하여 엔진 자원 검토

자원	onstat 명령
버퍼/메모리	-p, -P
CPU	-g, glo
디스크 활동	-P, -g iof
로그 활동	-l, -L
로크 활동	k
LRU 큐	-R, -F
체크포인트	-m
물리 로깅	-l

onstat -g glo를 사용하여 CPU 사용 모니터링

IDS 엔진의 작업 수행은 CPU에 따라 달라집니다. 특히 실행 시 충분한 CPU 자원을 필요로 하는 병렬 처리에서는 더욱 그렇습니다. onstat -g glo 명령은 전체적으로 엔진이 사용하는 CPU 자원을 나열하는 데 사용할 수 있습니다. 이 명령은 엔진의 다중 스레딩 작업에

대한 정보를 나열하므로, 엔진의 CPU 활용에 대한 중요한 정보를 제공합니다. 다음은 이 명령에서 나온 샘플 출력입니다.

```
INFORMIX-OnLine  Version  7.22.WC1      - On-Line - Up 00:02:15 - 9024 Kbytes

MT  global  info:
sessions   threads    vps      lngspins
0          10         7        0
      sched calls      thread switches    yield 0    yield n    yield forever
total:   318751             598            312823      225           84
per  sec:  0                 0               0          0             0

Virtual processor summary:
class     vps      usercpu        syscpu          total
 cpu       1         0.00          0.00            0.00
 aio       1         0.00          0.00            0.00
 lio       1         0.00          0.00            0.00
 pio       1         0.00          0.00            0.00
 adm       1         0.00          0.00            0.00
 soc       1         0.00          0.00            0.00
 msc       1         0.00          0.00            0.00
 total     7         0.00          0.00            0.00

Individual  virtual  processors:
  vp            pid      class    usercpu       syscpu      total
   1       fff80c29       cpu        0.00         0.00       0.00
   2       fff91265       adm        0.00         0.00       0.00
   3       fff910cd       lio        0.00         0.00       0.00
   4       fff90f55       pio        0.00         0.00       0.00
   5       fff90a3d       aio        0.00         0.00       0.00
   6       fff90885       msc        0.00         0.00       0.00
   7       fff9076d       soc        0.00         0.00       0.00
                          tot        0.00         0.00       0.00
```

출력의 처음 두 섹션은 엔진의 모든 가상 프로세서 작업에 대한 정보를 나타냅니다. MT global info 섹션은 VP에서 실행 중인 활성 세션(sessions) 수와 엔진의 총 스레드(threads) 수를 표시합니다. 이 출력에서 vps 컬럼은 엔진에서 사용 중인 VP 수를 표시합니다

(구성 매개변수를 사용하여 조정할 수 없는 관리 VP 포함). lngspins 컬럼은 VP를 사용하기 위해 대기 중인 스레드별로 긴 스핀의 수를 표시합니다.

```
          sched calls   thread switches    yield 0    yield n   yield forever
total:    318751        598                312823     225       84
per sec:  0             0                  0          0         0
```

출력의 다음 섹션은 VP의 스레드 관리에 대한 초 단위 계수와 합계(최종 계수를 정리하였기 때문에)의 목록입니다. 이 출력의 sched calls 컬럼은 스케줄의 스레드 호출 목록입니다. thread switches 컬럼은 스레드 간의 스위치 목록입니다. 몇몇의 컬럼은 한 개의 스레드가 산출한 횟수의 목록에 사용됩니다.

출력의 마지막 섹션은 다음과 같은 클래스별 가상 프로세서 활동의 요약 목록입니다.

```
Virtual  processor  summary:
class          vps        usercpu        syscpu         total
cpu             1           0.00          0.00           0.00
aio             1           0.00          0.00           0.00
lio             1           0.00          0.00           0.00
pio             1           0.00          0.00           0.00
adm             1           0.00          0.00           0.00
soc             1           0.00          0.00           0.00
msc             1           0.00          0.00           0.00
total           7           0.00          0.00           0.00
```

VP의 각 클래스마다 이 출력은 vps 컬럼의 해당 클래스에 대해 실행 중인 VP 수, usercpu 컬럼의 해당 VP에 대한 사용자 CPU 시간의 양, 그리고 syscpu 컬럼의 해당 VP에 대한 시스템 CPU 시간의 양이 나열됩니다. 최종 컬럼에는 클래스 각각의 'usercpu'와 'syscpu' 둘 모두의 총계가 나열되어 있습니다. 출력의 마지막 라인에는 모든 VP의 총계가 표시되어 있습니다.

이 출력은 sar와 vmstat와 같은 Unix 유틸리티의 출력과 유사합니다. 사용자 CPU 시간(초) 출력은 IDS 엔진에 대한 프로그램 명령을 실행하는 데 사용되는 CPU 시간(초)을 표시합니다. 시스템 CPU 시간(초) 출력은 CPU가 IDS 프로그램의 작업을 수행하기 위해 시스템 호출을 실행하는 시간을 표시합니다. IDS의 예측은 IDS가 많은

시스템 호출을 사용하지 않으므로 사용자 CPU 시간(초)은 시스템 CPU 시간(초)보다 더 길어야 한다는 것입니다. 출력에서 표시된 CPU 클래스는 표 5-4에 나열되어 있습니다.

표 5-4 CPU 클래스

CPU 클래스	설명
cpu	범용 CPU VP, 대부분의 사용자 스레드가 이 VP에서 실행
aio	비동기 입출력 작업 수행
lio	논리 로그 VP
pio	물리 로그 수행
adm	관리 기능 수행
shm	SHM 리스너 스레드 실행
tli	TLI 리스너 스레드 실행
soc	리스너 스레드 실행
msc	기타 스레드 실행

onstat VP 출력의 해석

CPU VP 클래스는 사용자 스레드의 실행에 사용되는 범용 VP입니다. 따라서 디스크의 섹션을 스캔하고 있는 스레드는 cpu 클래스 VP에서 실행됩니다. aio 클래스의 VP는 비동기 입출력을 수행하는 스레드를 실행합니다. 커널 aio가 사용 가능하면(kaio), 원시 장치에 대한 입출력은 kaio 스레드가 처리하며, 이는 궁극적으로 입출력을 시스템 호출로 전달하여 비동기 읽기/쓰기 작업을 수행하게 됩니다.

lio VP와 pio VP는 각각 논리 로깅과 물리 로깅을 수행합니다. 이러한 스레드는 관리자 제어 하에 있지는 않지만 유용한 정보를 제공합니다. 제공된 세션의 경우 CPU VP가 사용했던 것 이상으로 상당히 많은 CPU 시간(초)이 lio와 pio에 적용된 것으로 나타나면, 초과 로깅으로 CPU 자원이 제한될 수 있습니다.

adm과 msc VP 클래스는 각각 관리 기능과 기타 스레드를 실행하는 데 사용됩니다. 이 VP 클래스의 작업은 시스템 관리자의 제어 하에

있지 않습니다. soc VP 클래스는 연결용으로 Unix 소켓의 리스닝에 사용됩니다.

운영 체제 자원 모니터의 사용

sar 또는 vmstat 명령의 실행은 시스템 전체에 대한 CPU 사용을 나타냅니다. 이 명령의 출력은 VP에 사용할 수 있는 자원의 양을 나타냅니다.

대칭 멀티프로세서 플랫폼에서 모든 CPU에 대해 80~100%의 CPU 사용을 나타냈던 sar 출력은 사실상 플랫폼의 모든 CPU가 사용 중이라고 표시합니다. 엔진이 부담을 받지 않을 때(시스템 활동이 느린 기간 중) 이 출력이 생성된 경우, IDS의 시스템 로드가 피크일 때 엔진에 사용 가능한 CPU 자원이 거의 없거나 없다고 표시합니다. 즉, 엔진은 CPU 자원의 제한을 받습니다. CPU 자원이 없는 경우, CPU VP에서 처리가 실행될 때까지 기다려야 하며, 이로 인해 IDS 엔진의 처리가 느려집니다.

sar 또는 vmstat 실행으로 전체 시스템에 대한 CPU 사용이 30~50%에서 실행되고 있다고 표시되면, 엔진에 사용 가능한 추가 CPU 자원이 있다는 것입니다. 이 자원은 CPU VP를 추가함으로써 엔진에 사용할 수 있습니다. CPU VP는 매우 유용한 기능인 onmode 명령을 사용하여 동적으로 추가할 수 있습니다. CPU VP를 동적으로 추가하고 삭제함으로써 CPU 자원을 피크 시스템 로드에 적용할 수 있고 이후에 다른 목적으로 운영 체제로 리턴할 수도 있습니다. 또한 CPU VP는 벤치마크 실행용으로 추가되어 특정 쿼리의 엔진 실행에 대한 추가 CPU 자원의 효과를 결정합니다.

CPU 자원 문제의 해결

시스템 성능이 기대치 이하이며 머신에서 사용 가능한 CPU 자원이 있는 경우, onmode 유틸리티를 사용하여 CPU VP를 추가하고 엔진에서 사용 가능한 CPU 자원의 양을 늘릴 수 있습니다. 응용프로그램에 사용 가능한 여러 개의 CPU가 있는 SMP 머신에 별도의 CPU VP를 추가하는 것은, 사용 가능한 CPU 자원을 흡수함으로써 IDS

엔진이 필요할 때 스케일할 수 있다는 것입니다. onmode 명령은 다음과 같이 CPU VP를 동적으로 추가하는 데 사용할 수 있습니다.

```
onmode -p +/-<number of VPs to add> <VP class>
```

VP class는 표 5-4의 목록 클래스 중 하나일 수 있습니다. 이 명령은 VP 추가 시에도 사용할 수 있습니다. 따라서 IDS 엔진에는 엔진 처리에 사용 가능한 수 이상의 VP가 존재할 수 있습니다. VP의 수는 엔진을 시작했을 때 할당된 VP의 수까지 감소할 수 있지만 이 숫자 이하로 감소되지는 않습니다. 따라서 다섯 개의 CPU VP로 엔진을 시작하고 두 개의 VP가 추가되었으면 onmode 명령을 사용하여 감소할 수 있는 최대의 VP 수는 2개가 됩니다. 다음 명령을 사용하면 IDS 엔진에 2개의 CPU VP가 추가됩니다.

```
onmode -p +2 CPU
```

다음 명령을 사용하면 2개의 CPU VP를 뺄 수 있습니다.

```
onmode -p -2 CPU
```

만일 onstat -g glo 출력 상에서 많은 CPU 시간(초)이 논리 로깅에 사용되고 있는 것으로 나타나거나 많은 로깅 활동을 생성할 몇 가지 배치 프로세스가 실행될 것으로 알려져 있으면, 추가 로깅 VP는 로그 관리를 돕도록 추가될 수 있습니다. 그리고 나서 시스템 로깅 활동이 줄어들면, 로깅 VP의 수를 감소시킬 수 있습니다. 다음 명령은 두 개의 lio VP를 IDS 엔진에 추가합니다.

```
onmode -p -2 LIO
```

IDS 메모리 사용 모니터링

비용이 많이 드는 디스크 입출력 작업은 가능한 많이 줄일수록 좋으므로, IDS 엔진의 메모리 버퍼 튜닝은 디스크 튜닝과 일부 중복되는 부분이 있습니다. 좋은 디스크 성능을 높이려면 이 두 작업을 모두 튜닝하는 것이 필수입니다.

IDS 엔진은 버퍼 캐시 또는 공유 메모리 버퍼 캐시라고 알려진 것

에서 공유 메모리의 부분을 정렬합니다. 이 버퍼 캐시는 디스크에서 검색된 데이터를 캐시에 저장하는 데 사용됩니다. 모든 IDS 읽기 작업에서는 페이지가 메모리에 없는 경우, 디스크에서 직접 읽지 않고 버퍼 캐시 페이지에서 데이터를 읽습니다. 읽기 명령은 디스크에서 페이지를 얻기 위해 발행됩니다. 따라서 디스크의 모든 데이터는 IDS 작업이 작동하기 전에 공유 메모리로 읽어야만 합니다.

가능하면 IDS는 디스크 작업이 발생하는 횟수를 줄이려고 합니다. 일단 페이지를 메모리에 읽고 나면, IDS 엔진은 다른 자원이 페이지를 사용하고 있지 않는 경우(자원 경쟁이 발생), 메모리에서 페이지를 읽습니다. IDS가 메모리에서 페이지를 찾는 것을 캐시 적중률이라고 합니다. 높은 숫자 또는 높은 비율의 캐시 적중률('onstat -p' 출력으로 표시됨)은 엔진이 데이터 읽기 작업을 효율적으로 수행하고 있다는 것을 나타냅니다.

'onstat -p' 옵션을 사용하면 엔진에 대한 프로파일 계수가 나열됩니다. 이 명령의 출력은 다음과 같습니다.

```
INORMIX-OnLine  Version  7,22,WC1     - On-Line - Up 00:04:55 - 9024  kbytes

Profile
dskreads   pagreads   bufreads  %cached   dskwrits  pagwrits  bufwrits  %cached
258        276        28280     99.09     608       705       5302      88.53

isamtot   open     start    read    write    rewrite    delete    commit    rollbk
20056     3014     3213     5950    2739     105        51        230       0

ovlock    ovuserthread      ovbuff     usercpu     syscpu     numckpts    flushes
0         0                 0          0.00        0.00       2           4

bufwaits   lokwaits   lockreqs   deadlks  dltouts   ckpwaits   compress   seqscans
23          0          6203       0        0         1          2          172

ixda-RA   idx-RA    da-RA     RA-pgsused       lchwaits
25        0         1         22               1
```

표 5-5에 이러한 출력이 설명되어 있습니다.

표 5-5 엔진 프로파일 계수 설명

계수	설명
dskreads	디스크로부터의 실제 읽은 수
pagreads	읽은 페이지의 수
bufreads	공유 메모리로부터의 읽은 수(캐시 적중률)
%cached	캐시된 읽기 비율. 100*(bufreads-dskreads)/bufreads
dkswrits	디스크에 쓴 수(물리 로그 및 논리 로그 포함)
pagwrits	쓴 페이지의 수
bufwrits	공유 메모리에 쓴 수(더티 페이지 작성)
%cached	캐시된 쓰기 비율. 100*(bufwrits-dskwrits)/dskwrits

경험에 따른 일반적인 규칙은 캐시 적중률(캐시된 %)이 80% 이하이면 더 높은 캐시 적중률을 얻기 위해 더 많은 메모리를 추가함으로써 엔진을 튜닝할 수 있습니다. 그러나 이것은 단지 경험상의 법칙입니다. 메모리가 제한되어 있고/또는 응용프로그램이 공유 메모리에 적합하지 않은 대량의 데이터를 읽는 경우(데이터 웨어하우스/DSS 응용프로그램에서는 일반적임), 이 규칙은 적용되지 않습니다.

관리자는 시스템에 적절한 내용을 결정해야 합니다. 일반적으로 더 높은 캐시 적중률이 데이터 검색을 향상시킨다는 사실을 유념하십시오.

사용자가 언제든지 어떤 응용프로그램 옵션이라도 실행할 수 있는 개방 환경에서 액세스는 가변성을 가지므로, IDS 서버에서 요구하는 정확한 메모리를 결정하는 것은 어렵습니다. 관리자는 시스템 활동을 모니터해야 하고 시스템의 피크 활동 로드를 결정해야 합니다.

IDS 모니터를 위한 onstat 옵션의 반복

옵션 세트를 반복하고 옵션을 파일로 출력하는 데 onstat 명령을 사용하여 시스템을 모니터할 수 있습니다. 파일은 효과적으로 일정 기간 동안의 시스템 활동 로그가 될 수 있습니다. 그런 다음 시스템에 대한 피크 활동을 결정하기 위하여 이 로그를 검토할 수 있습니다.

onstat 유틸리티는 -r 옵션을 사용하여 옵션 세트를 반복합니다.

-r 옵션에 제공되는 인수는 명령을 다시 실행하기 전에 대기해야 하는 시간(초)입니다. 이러한 방식으로 onstat를 실행하는 경우, onstat는 영구 루프의 형태로 실행되며, 대화형 명령행에서 Ctrl-C 키를 누르거나 onstat 명령을 실행하는 프로세스를 종료하기 위해 Unix kill 명령을 사용하여 유틸리티가 중단될 때까지 이 루프를 반복합니다. 명령을 반복하려면, 다음과 같이 스크립트를 작성할 수 있습니다.

```
onstat -p -r 300 -z >> output.log
```

이 스크립트는 300초마다(5분) -p 옵션을 반복 사용하여 onstat 명령을 실행합니다. '-z' 옵션은 '-p' 옵션 이후에 실행되어 통계 계수를 0으로 만들거나 정리하고, 시스템 활동의 5분간의 스냅샷을 output.log 파일에 누적하게 합니다.

메모리의 특정 부분은 엔진에서 의사 결정 지원 쿼리라고 인식하는 처리를 위해 남겨둡니다. 이 쿼리는 쿼리의 테이블 간 정렬 및 조인을 처리하기 위한 메모리를 필요로 하며 대형 테이블을 스캔하는 쿼리입니다. 의사 결정 지원 메모리의 총량은 DS_TOTAL_MEMORY 구성 매개변수를 사용하여 설정됩니다. DS_TOTAL_MEMORY 구성 매개변수는 -M 매개변수가 있는 onmode 명령을 사용하여 엔진을 온라인으로 튜닝할 수 있습니다.

메모리 문제 해결

낮은 버퍼 캐시 적중률의 문제에 대한 명백한 해결책은 메모리를 IDS 인스턴스로 할당하는 것입니다. 그러나 이 문제는 공유 메모리가 필요한 플랫폼 상에서 다른 응용프로그램 요구와 균형을 맞추어야 합니다.

물리적 메모리를 사용할 수 있는 경우 플랫폼에서 물리적 메모리가 공유 메모리로 할당되어 있는 지를 확인해야 합니다. 물리적 메모리를 사용할 수 있고 공유 메모리로 할당되어 있으면, 변경을 인식할 수 있도록 공유 메모리의 IDS 구성 파일 매개변수를 조정할 수 있습니다.

디스크 활동 모니터링

IDS 서버는 매우 유연하므로 데이터가 여러 디스크에서 물리적으로 분산될 수 있습니다. 데이터를 적절하게 분산시킴으로써 디스크 입출력 병목 현상을 완화할 수 있습니다. 그러나 IDS 데이터베이스에서 데이터의 분할화라고 알려진 데이터의 적절한 분산을 결정하는 것은 어려운 작업이 될 수 있습니다.

데이터 입출력 병목 현상이 있는지 여부를 결정하려면 DBSPACE 또는 청크별 읽기와 쓰기를 모니터해야 합니다. -D 매개변수와 함께 onstat 명령을 사용하여 이를 결정할 수 있습니다. 이 명령은 다음과 같이 청크 단위의 읽기와 쓰기를 나타냅니다.

```
INFORMIX-OnLine  Version  7.22.WC1     - On-Line - Up 00:05:24 - 9024  Kbytes
Dbspaces
address   number    flags     fchunk      nchunks    flags   owner    name
82efa0ec  1         1         1           1          N      Informix  rootdbs
1 active,  2047  maximum
Chunks
address  chk/dbs  offset  page Rd  page Wr   pathname
82efa158  1   1   0       276      705       D:\IFMXDATA\ol_ataylor_homepc\
                                             rootdbs_dat.000
1 active, 2047 maximum
```

이 명령의 출력에서 처음 부분에는 시스템의 DBSPACE에 대한 정보가 나열되어 있습니다. 이 섹션에 대한 출력에서 첫 번째 컬럼은 공유 메모리의 DBSPACE 정보 위치를 표시하는 내부 번호인 DBSPACE 주소를 표시합니다. 다음 컬럼은 DBSPACE의 번호를 표시합니다. 이 번호는 출력의 다음 섹션에서 DBSPACE를 참조하는데 사용됩니다. 다음 컬럼은 이 DBSPACE에 대한 플래그를 표시합니다. 이 플래그는 16진수로서 그 의미가 표 5-6에 나와 있습니다.

fchunk 컬럼은 DBSPACE의 최초 청크의 청크 번호를 표시합니다. 이는 DBSPACE의 초기 청크지만 DBSPACE에서 삭제할 수는 없습니다. 다음 컬럼은 DBSPACE의 상태를 표시하는 플래그 세트를 제공합니다. 의미는 표 5-7에 설명되어 있습니다.

표 5-6 DBSPACE 플래그 값

값	설명
0x0001	미러 없음
0x0002	미러
0x0004	다운
0x0008	새로 미러됨
0x0010	BLOBSPACE

표 5-7 DBSPACE 상태 플래그 값

포지션	설명
	M-미러됨
1	N-미러되지 않았음
2	X-새로 미러됨 P-물리적으로 복구됨, 논리적 복구 대기 L-논리적으로 복구 중 R-복구 중
3	B-BLOBSPACE

마지막 두 컬럼은 DBSPACE의 owner(DBSPACE를 작성한 사용자)와 DBSPACE에 지정된 name을 표시합니다.

이 onstat 출력의 최종 섹션은 다음과 같습니다. 이 섹션은 시스템에서 DBSPACE를 구성하는 청크에 관한 정보를 표시합니다. 이 정보는 기존의 청크에서 사용된 공간과 관련이 있습니다.

```
...
Chunks
address  chk/dbs   offset   size   free   bpage   flags    pathname
83198158  1   1    0        2120   33              -PO     rootdbs_dat . 000
1 active,  2047  maximum
```

위의 출력에서 첫 번째 컬럼은 공유 메모리의 청크 정보 주소입니다. 컬럼 번호 및 DBSPACE 번호 또한 행에 표시되어 있습니다. DBSPACE 번호는 첫 번째 섹션의 DBSPACE 번호를 참조합니다.

이 행의 다음 컬럼은 장치로의 청크 offset입니다(동일한 라인의 경로 이름 출력에서 참조). 청크의 크기는 청크에서 사용 가능한 바이트 다음에 오는 컬럼에 표시됩니다.

다음 컬럼은 bpage입니다. 표 5-8에서 표시와 같이, 청크의 상태를 표시하는 16진수 플래그로 청크의 BLOBPAGE를 나타냅니다.

표 5-8 청크 상태 플래그

플래그	설명
0x001	미러 없음
0x002	미러
0x004	다운
0x008	새로 미러됨
0x0010	BLOPSPACE

마지막 컬럼은 청크를 보유한 장치의 pathname을 나타냅니다. 이전에 표시된 출력의 청크 크기는 사용 가능한 33KB의 여분을 포함하여 2120KB입니다.

디스크 문제의 해결

IDS 엔진의 디스크 관리 목표는 가능한 균등하게 데이터 액세스를 분산시키는 것입니다. 논리 로그, 물리 로그 및 변하기 쉬운 모든 데이터를 동일한 물리적 디스크에 위치 지정하지 않는 일반적인 접근이 좋은 접근 방식입니다. 액세스 비율이 높은 데이터 테이블과 로그는 가능한 많이 복수 디스크에 분산시켜야 합니다. 데이터베이스 관리자는 DBSPACE에 매핑된 디스크의 물리적 위치를 알아야 합니다. 가능하면 대형 테이블의 조각은 다중 DBSPACE에 지정하여 여러 디스크 전체에 분산시켜야 합니다. 그러나 실제로 이 DBSPACE가 동일한 디스크에 청크를 가지고 있으면, 디스크 입출력 로드를 분산시키는 목적을 달성할 수 없습니다.

onstat -D를 사용하면 출력에 읽기와 쓰기가 표시됩니다(Page Rd 및 Page Wr). 이 명령을 출력했을 때 높은 비율의 디스크 입출력이 적은 DBSPACE 세트로 지정되어 있으면, alter fragment 명령을 사용하여 테이블의 부분을 또 다른 DBSPACE로 재지정할 수

있습니다. 이 장의 뒷부분에서는 IDS 엔진의 적절한 데이터 분산으로 중요한 부분인 여러 DBSPACE에 데이터를 분산하는 분할화 스키마에 대해 설명합니다.

IDS 엔진의 로그 활동

Informix는 디스크에 쓰기 전에 공유 메모리에서 물리 로그 및 논리 로그 출력을 버퍼에 지정 기능을 제공합니다. 이 버퍼의 크기를 올바르게 결정하면 로그를 디스크에 작성하는 데 필요한 디스크 쓰기의 횟수를 줄일 수 있으므로 성능을 향상시킬 수 있습니다.

"-l" 매개변수를 사용하는 onstat 명령은 물리 로그 및 논리 로그에 대한 정보를 인쇄합니다. 표시된 출력의 첫 번째 섹션은 표 5-9에 표시된 물리 로그에 대한 정보를 나타냅니다.

출력의 두 번째 섹션은 표 5-10에 표시된 논리 로그에 대한 정보를 표시합니다.

각 로그 파일의 상태를 설명하기 위해 추가 출력을 제공하지만 로그 입출력의 튜닝은 각 로그의 상태와는 관련이 없습니다.

표 5-9 onstat 물리 로그 출력

항목	설명
buffer	물리 로그 버퍼의 수
bufused	물리 로그 버퍼의 페이지 수
bufsize	각 물리 로그 버퍼의 크기
numpages	물리 로그에 쓴 페이지 수
numwrits	디스크에 쓴 수
pages/io	쓰기별 물리 로그 페이지 수(numpages/numwrits)
phybegin	로그 시작의 물리적 페이지 수
physize	페이지의 물리 로그 크기
phypos	다음 로그 레코드 쓰기가 발생할 물리 로그의 현재 포지션
phyused	로그에서 사용된 페이지 수
%used	사용된 페이지의 비율

표 5-10 onstat 논리 로그 출력

항목	설명
buffer	논리 로그 버퍼의 수
bufused	논리 로그 버퍼에서 사용된 페이지 수
bufsize	각 논리 로그 버퍼의 크기
numrecs	쓴 레코드의 수
numpages	쓴 페이지의 수
numwrits	논리 로그에 쓴 수
recs/pages	페이지별 레코드 수(numrecs/numpages)
pages/io	페이지 입출력의 수(numpages/numwrits)

로그 성능 문제의 해결

논리 로그를 튜닝하려면, 로그 쓰기 입출력별 페이지를 최대화함으로써 로그 파일을(물리 로그 및 논리 로그) 디스크로 가져오기 위해 엔진이 수행해야 하는 디스크 입출력의 양을 줄일 필요가 있습니다. 이는 로그 파일을 저장하는 데 사용한 버퍼 크기를 변경함으로써 수행할 수 있습니다. 이 작업은 버퍼 크기가 onstat -l 출력에서 보고된 입출력별 페이지 수에 가능한 가까워질 때까지 구성 매개변수 LOGBUFF와 PHYBUFF를 변경함으로써 수행됩니다.

많은 논리 로그의 부분이 입출력 요청으로 포화된 디스크에서 지속되고 있는 경우(onstat -D와 onstat -g iof 명령을 실행하여 나타난 것처럼), 일부 또는 모든 논리 로그 파일을 덜 경쟁적인 디스크 부분으로 이동해야만 할 수도 있습니다. 이는 onparams 명령을 사용하여 수행할 수 있습니다.

물리 로그 활동의 평가 및 튜닝

물리 로그는 업데이트할 페이지의 사전 이미지를 저장하는 데 사용합니다. 이 사본은 디스크의 물리 로그에 쓰여지고 시스템 복구를 위하여 업데이트된 페이지를 이전의 상태로 복구해야 할 필요가 있는

경우(급속 복구 경우와 같음)에 사용됩니다. 그러나 '사전 이미지'는 디스크의 물리 로그에 쓰기 전에 우선 메모리의 물리 로그에 작성됩니다. 메모리의 이 버퍼 위치는 특정 간격으로 디스크에 쓰여집니다.

메모리에는 2개의 물리 로그 버퍼가 있습니다. 첫 번째 물리 로그 버퍼가 찼으면, 메모리의 두 번째 물리 로그 버퍼가 사용됩니다. 따라서 시스템 성능을 방해하지 않고 물리 로그 버퍼에 대한 쓰기를 계속 수행할 수 있습니다. 물리 로그가 75%에 이르면 체크포인트가 트리거되고 물리 로그의 내용이 디스크에 쓰여집니다.

물리 로그는 두 가지 방식으로 성능에 영향을 줄 수 있습니다. 첫 번째로 IDS 엔진은 연속적으로 물리 로그에 쓰기를 수행하므로, 물리 로그를 포함하는 디스크에는 많은 양의 디스크 입출력이 발생합니다. 두 번째로 물리 로그가 75%까지 차면, 앞의 절에서 언급한 대로 시스템 성능을 지연시킬 수 있는 체크포인트를 강제로 실행시킵니다.

물리 로그 위치

디스크의 섹션에서 물리 로깅의 초과 디스크 입출력을 방지하도록 물리 로그의 위치를 변경할 수 있습니다. 엔진 생성 시, 물리 로그는 대량의 디스크 활동이 발생할 수 있는 DBSPACE인 root DBSPACE에 배치됩니다. 이 물리 로그는 PHYDBS 구성 매개변수를 변경한 후 엔진을 재시작하여 이동시킬 수 있습니다.

PHYDBS 매개변수는 물리 로그가 위치할 DBSPACE를 나타냅니다. 물리 로그는 물리 로그를 인접 공간으로 보유할 정도로 충분히 큰 DBSPACE 모두에 위치할 수 있습니다. 로그는 로그 상에서 수행되는 대량의 디스크 입출력에 의하여 영향을 받지 않는 DBSPACE에 배치하십시오. (이는 onparams 명령 또는 onmonitor 유틸리티를 사용하여 물리 로그를 이동할 수 있습니다.) 일단 PHYDBS 매개변수를 변경한 경우, 매개변수 변경을 인식하려면 엔진을 재시작해야 합니다.

물리 로그 크기

물리 로그가 연속적으로 75%에 도달하여 설정된 체크포인트 간격 사이에서 체크포인트를 강제 실행하는 경우, 물리 로그의 크기를 증가시켜서 체크포인트의 수를 줄일 수 있습니다. 이는 체크포인트

사이의 시간을 늘려주지만, 디스크로 플러시할 물리 로그의 페이지가 더 많아지므로, 체크포인트의 길이도 늘어나게 됩니다.

그러나 체크포인트 간격이 짧아지면 더 적은 물리 로그로 인해 더 빠른 체크포인트 처리가 가능해집니다. 따라서 궁극적으로 필요한 것은 물리 로그 크기와 체크포인트 간격 간에 균형을 이루는 것입니다. 즉, 이는 최소의 체크포인트 오버헤드를 산출하고 시스템 장애 시 데이터 손실을 최소화하는 균형입니다.

BLOBSPACE에 저장된 BLOB은 물리 로그에 쓰여지지 않습니다. (BLOBSPACE BLOB의 데이터 무결성은 또 다른 메커니즘을 통해 관리됩니다.) 그러나 테이블의 데이터 레코드로 저장된 BLOB인 TABLESPACE BLOB은 물리 로그에 쓰여집니다. 따라서 TABLESPACE BLOB은 가중된 물리 로그 활동을 가져오는 테이블 업데이트를 방지하기 위하여 신중하게 사용되어야 합니다.

물리 로그는 시스템의 동시 사용자 수와 이 사용자가 주어진 시점에서 업데이트할 수 있는 추정된 페이지 크기를 기준으로 예측한 크기로 초기 설정할 것을 권장합니다. 설정된 수의 동시 사용자가 보유하는 페이지 수를 추정하는 것이 어렵다는 가정 하에, 또 다른 접근 방식은 특정 물리 로그 크기에서 시작하여 시스템 활동 중 물리 로그를 모니터하는 것입니다. 물리 로그가 가득 차고 체크포인트 간격 사이에서 체크포인트를 강제로 수행하면 물리 로그의 크기는 늘어나야 합니다.

물리 로그 크기는 PHYSFILE 매개변수를 사용하여 변경할 수 있습니다. 일단 이 매개변수를 변경하고 나면 엔진을 재시작하여 이 변경사항을 인식합니다. 또는 `onparams` 명령 또는 `onmonitor` 프로그램을 사용하여 이 매개변수를 변경할 수 있습니다.

LRU 큐의 평가

앞에서 언급한 대로 IDS 엔진은 공유 메모리의 모든 입출력을 버퍼에 지정합니다. 데이터베이스 테이블에 대한 모든 쓰기는 먼저 공유 메모리 버퍼의 페이지에 수행됩니다. 정기적으로 공유 메모리에서 업데이트된 페이지(더티 페이지라고 함)는 디스크에 쓰여져야 합니다. 이 프로세스는 때때로 버퍼를 디스크에 플러시한다고 합니다. 페이지는 페이지 플러셔 스레드고 알려진 스레드에 의해 디스크에 쓰여집니다.

업데이트된 LRU 버퍼는 Least Recently Used 버퍼로 알려진 매

핑 메커니즘을 통해 관리됩니다. 이 메커니즘은 공유 메모리 버퍼에서 페이지로 매핑합니다.

LRU 큐는 업데이트 집중 시스템에서 최대의 처리량을 제공하기 위해 설계되었습니다. 적절하게 튜닝된 LRU 큐를 사용하여 곧 사용할 예정인 페이지를 메모리에 보관하므로 공유 메모리를 채우기 위해 디스크에서 페이지를 다시 읽지 않아도 됩니다.

LRU 성능의 평가

2개의 onstat 명령을 사용하여 LRU 관리 시스템의 성능을 평가할 수 있습니다. 즉, '-F' 명령을 사용하여 페이지 플러셔 명령을 표시할 수 있고, '-R' 명령을 사용하여 LRU 큐에 대한 정보를 출력할 수 있습니다. '-F' 옵션을 사용하면 다음과 같이 출력됩니다.

```
INFORMIX-OnLine  Version  7.22.WC1     - On-Line - Up 07:00:59 - 9024  Kbytes

Fg Writes           LRU Writes              Chunk Writes
0                   0                       31

address             flusher     state       data
82efc448            0           I           0           = 0X0
states: Exit Idle Chunk Lru
```

명령의 처음 부분은 표 5-11과 같은 LRU 큐의 누적 정보를 나타냅니다.

표 5-11 LRU 큐

Fg writes	포그라운드 쓰기 수
LRU writes	LRU 쓰기 수
Chunk writes	청크 쓰기 수

이들 각 컬럼의 정보는 페이지 클리너가 더티 버퍼를 정리하기 위하여 웨이크업된 이유를 설명해 줍니다. 포그라운드 쓰기 수(Fg writes) 컬럼의 출력은 큐에 쓸 필요가 있지만 더티 버퍼로 인해 쓸 수 없으므로 버퍼를 정리하기 위해 페이지 클리너를 웨이크업해야 하는 작업이 트리거한 쓰기 수를 나타냅니다.

LRU 쓰기 수(LRU writes) 컬럼의 출력은 제공된 큐에 대해 LRU_MAX_DIRTY 비율에 도달했다고 클리너가 결정했을 때 LRU 메커니즘이 클리너를 웨이크업한 횟수를 나타냅니다.

청크 쓰기 수(chunk writes) 컬럼의 출력은 체크포인트 동안 버퍼를 정리하기 위해 클리너를 웨이크업한 횟수를 나타냅니다. 이때, 버퍼는 버퍼가 속한 청크별로 정렬되고 페이지 클리너는 청크와 출력에 정렬된 큐에 지정됩니다. (따라서 청크 쓰기라는 용어가 나옵니다.)

명령 출력의 두 번째 부분은 다음과 같이 시스템의 페이지 클리너(플러셔)에 대한 정보를 표시합니다.

```
address   flusher   state   date
82efc448  0         I       0       = 0X0

states: Exit Idle Chunk Lru
```

address 컬럼은 출력의 행에서 참조하고 있는 페이지 클리너에 대한 주소(공유 메모리)를 표시합니다. flusher 컬럼은 참조하고 있는 플러셔(클리너)의 번호를 표시합니다. state 컬럼은 상태가 Exit, Idle, Chunk, LRU 중 하나인 페이지 클리너의 상태를 표시합니다. 상태가 Exit이면, 페이지 클리너는 종료되고 더 이상 실행되지 않습니다. 상태가 Idle을 표시하는 경우, 페이지 클리너는 웨이크업되기를 기다리고 있고(휴면) 정리할 큐로 지정됩니다. 페이지 클리너가 Chunk 상태이면, 체크포인트 시간이 chunk write를 수행하고 있을 때 웨이크업된 것입니다. 페이지 클리너의 상태가 LRU이면, 정리할 LRU가 지정된 LRU 큐에 의해 웨이크업되고 LRU write를 수행 중인 것입니다.

LRU 큐를 적절하게 튜닝하기 위해서는 발생 중인 청크 쓰기의 양을 줄여야 합니다. 청크 쓰기는 체크포인트 시간에 발생하고 시간이 너무 많이 걸리는 경우 체크포인트의 길이가 늘어날 수 있기 때문입니다. 이전에 표시된 출력의 LRU 쓰기 컬럼에 표시한 대로 버퍼는 LRU 메커니즘을 사용하여 지우려고 합니다. '-R' 명령은 표 5-12에 표시된 LRU 큐에 대한 정보를 표시합니다.

표 5-12 -R LRU 큐 출력

#	큐 번호 f/m 큐 유형
f	사용 가능 LRU 큐
F	요소가 가장 적은 사용 가능 큐
m	MLRU 큐
M	플러시 중인 MLRU 큐
length	버퍼로 측정된 큐 길이
% of	서브큐의 큐 비율
pair total	LRU 큐의 버퍼 총 수

출력의 이 섹션은 각 개별 큐의 상태에 대한 정보를 표시합니다. 출력의 pair total에는 큐의 버퍼 총 수에 대한 정보가 포함되어 있습니다. 이 정보는 페이지 플러시 작업 중에 큐의 크기가 손쉽게 쓸 수 있을 정도로 충분히 큰지 여부를 결정할 때 유용합니다. 또한 컬럼의 % 출력으로 큐가 채워지는 속도에 대해 알 수 있습니다.

출력의 다음 섹션은 표 5-12에서 표시된 대로 큐에 대한 요약 정보를 표시합니다.

표 5-13 - LRU 큐 요약 정보

dirty	모든 LRU 큐에서 수정된 총 버퍼 수
queued	LRU 큐의 총 버퍼 수
hash buckets	해시 버킷의 수
buffer size	각 버퍼의 크기
start clean	LRU_MAX_DIRTY의 값
stop at	LRU_MIN_DIRTY의 값

출력의 마지막 섹션은 LRU 활동의 총 수를 표시합니다. 더티 또는 수정된 페이지의 총 수가 LRU 큐의 총 수 및 각 버퍼의 크기와 함께 나열됩니다. 해시 버킷 수가 나열되며 마지막으로 구성 매개변수 LRU_MAX_DIRTY 및 LRU_MIN_DIRTY의 값이 나열됩니다.

LRU 큐는 특정 간격으로 명령을 반복하기 위하여, -r 옵션이 있는

onstat 명령을 사용하여 시스템 활동 중에 모니터해야 합니다. 관리자는 매우 빠르게 채워져서 페이지 클리너가 체크포인트 이전에 더티 페이지를 플러시할 시간이 없는 큐를 찾아야 합니다. 다음 절에서는 LRU 큐에 대한 튜닝 방법에 대해 설명합니다.

LRU 관리 튜닝

LRU 큐의 관리는 영향이 간접적이라 하더라도 시스템 성능에 대하여 중대한 영향을 줄 수 있습니다. LRU 관리가 신호를 보냈을 때 수정된 페이지(더티 페이지)를 페이지 플러셔가 디스크에 쓰지 않았으면, 이 페이지는 체크포인트 시간에 쓰여집니다. 페이지를 체크포인트 시간에 썼을 때의 문제는 체크포인트가 작업하는 동안에 실제로 모든 중요 시스템 활동을 중단(대부분의 시스템 활동이 중지되는)한다는 것입니다. 체크포인트 간격 중 대량의 더티 페이지를 써야 하는 경우, 체크포인트는 어느 정도의 시간이 소요될 수 있습니다.

따라서 LRU 큐 관리의 튜닝에서 기본적인 작업은 일반 시스템 처리 중 LRU 큐를 관리하는 데 수행하는 작업 양을 최대화하고 체크포인트에서 수행하는 작업을 최소화하는 것입니다. 이 작업은 체크포인트 발생 전에 LRU 큐 관리 기능이 더 많은 페이지 쓰기를 트리거하도록 함으로써 수행될 수 있습니다. 이러한 페이지 쓰기를 트리거하는 데 도움을 주는 사용 가능한 네 개의 매개변수로는 LRU_MIN_DIRTY, LRU_MAX_DIRTY, LRUS 및 CLEANERS가 있습니다. 이러한 매개변수에 대한 설명은 표 5-14와 같습니다.

표 5-14 - LRU 구성 매개변수

매개변수	설명
LRUS	생성할 공유 메모리 LRU 큐의 수
LRU_MIN_DIRTY	이 더티 페이지 비율에서 LRU 정리를 중지
LRU_MAX_DIRTY	이 더티 페이지 비율에서 LRU 큐 정리를 시작
CLEANERS	지정된 페이지 클리너 스레드의 수

LRUS 매개변수는 생성할 LRU 큐의 수를 나타냅니다. 선택된 큐의 수는 큐의 크기에 영향을 주고 큐의 크기는 큐를 디스크에 쓰는 데 소요되는 시간에 영향을 줍니다. 따라서 LRU 매개변수는 CLEA

NERS 매개변수를 사용하여 조정해야 합니다. 이 CLEANERS 메커니즘은 LRU 메커니즘에 의해 더티 페이지를 디스크에 쓰도록 지정된 클리너 스레드의 수를 설정합니다. 경험에 따른 일반적인 규칙은 LRU의 수가 CPU VP의 수와 동일하게 설정되어야 하며, CLEANERS의 수는 청크의 수로 설정되어야 한다는 것입니다.

IDS 엔진은 각 큐의 LRU 버퍼 페이지를 모니터합니다. 큐가 더티 페이지의 LRU_MAX_DIRTY 비율에 도달하면, 클리너 스레드는 해당 큐의 더티 페이지를 디스크에 쓰도록 지정됩니다. 이 클리너 스레드는 해당 큐의 더티 페이지 비율이 페이지 쓰기를 중지하는 지점인 LRU_MIN_DIRTY 비율에 도달할 때까지 계속해서 디스크에 페이지를 씁니다.

앞서 언급한 대로, LRU 큐 튜닝의 목표는 체크포인트에서 쓰여진 페이지의 수를 최소화하는 것입니다. 따라서 체크포인트 사이에 가능한 많은 페이지를 쓰는 것이 필요합니다. 이 체크포인트 사이에서 쓰여진 페이지 쓰기는 주로 LRU 메커니즘에 의해 트리거됩니다. 따라서 목표는 LRU 메커니즘이 쓰는 페이지의 수를 최대화하는 것입니다.

LRU 큐가 더 많은 페이지 쓰기를 트리거하게 하려면, LRU_MAX_DIRTY 매개변수를 더 낮게 설정해야 합니다. 이 매개변수를 더 낮게 설정하면 LRU 메커니즘은 페이지 쓰기를 더 빨리 트리거합니다.

페이지 클리너 스레드를 웨이크업하고 정리할 LRU 큐를 지정하면, LRU_MIN_DIRTY 비율에 도달할 때까지 큐를 정리하게 됩니다(업데이트된 페이지를 디스크에 쓰기). 클리너 스레드가 체크포인트 사이에서 더 많은 페이지를 디스크에 강제로 쓰게 하려면, LRU_MIN_DIRTY 비율을 더 낮게 설정해야 합니다. 즉, LRU 관리에서 페이지 클리너를 웨이크업하면 더 많은 페이지를 쓰게 됩니다.

LRU 매개변수 설정 시, 델타 즉 LRU_MIN_DIRTY와 LRU_MAX_DIRTY 설정의 차이는 페이지 클리너가 수행해야 하는 작업량에 직접적인 영향을 줍니다. 예를 들어, 시스템에서 공유 메모리 버퍼에 사용 가능한 메모리가 1GB이고 20개의 LRU 큐를 지정한 경우, 각 LRU 큐는 50,000,000byte(1,000,000,000/20)의 큐로 지정됩니다. LRU 매개변수 설정이 LRU_MIN_DIRTY=10이고 LRU_MAX_DIRTY=30이면, 버퍼를 정리하기 위해 웨이크업되었을 때 델타는 20이며 페이지 클리너는 50MB 버퍼의 20%인 10MB(.2*50,000,000)를 쓰게 됩니다. 실제로 IDS가 생산 중이고 일정한 비율로 업데이트를 수행하고 있는 경우, 더티 버퍼를 정리용으로 일단 한 개의 페이지 클리너를 웨이크업했을 때 나머지도 곧 웨이크업될 가능성이 높습니다. LRU 설정이 최적인지 여부는 디스크 드라이브의 속도

와 수(입출력 처리량) 및 사용 가능한 CPU 자원의 양(일부 CPU 사이클은 쓰기 수행에 필수임)과 같은 IDS에서 사용 가능한 머신 자원에 따라 달라집니다. 일부를 검토했을 때(LRU_MIN_DIRTY와 LRU_MAX_DIRTY간의 델타 설정에 따라) 클리너 번호당 10MB가 너무 큰 것으로 결정되었으면, CLEANERS 구성 매개변수를 사용하여 페이지 클리너의 수를 증가함으로써 수를 더 적게 만들 수 있습니다.

IDS 시스템에 대한 올바른 LRU 설정을 결정하는 데에는 관찰과 실험이 중요합니다. IDS 시스템의 프로파일은 업데이트 로드가 가중되었는지, 경미한지 또는 이 두 가지가 정기적으로 조합되었는지에 상관없이 이러한 매개변수의 설정 시에 주요한 고려사항이 됩니다. 다음 절에서는 이러한 고려사항과 LRU 큐의 튜닝에 미치는 영향에 대해 설명합니다.

가중된 업데이트 로드에 대한 LRU 매개변수 설정

OLTP 시스템의 공통적인 트랜잭션 프로파일인 많은 동시성과 가중되고 일정한 업데이트 로드를 지닌 시스템은 체크포인트 시간을 최소화하고 체크포인트 사이에 쓴 더티 버퍼 페이지의 양을 최대화해야 합니다.

이와 같은 시스템에서 LRU_MAX_DIRTY를 10으로, LRU_MIN_DIRTY를 2로 설정하면 효과적인 설정이 됩니다. 이렇게 밀접한 델타는 다수의 페이지를 체크포인트 사이에 쓰게 만들고 체크포인트 시간에서 최소의 디스크 쓰기 작업을 수행하도록 합니다. 머신 자원이 체크포인트 사이의 디스크 쓰기 로드를 관리할 수 있다고 확신할 때 체크포인트는, 체크포인트 시간에서 엔진이 수행해야 하는 작업을 지연시키는 동시성 작업을 신속히 수행하면서 각 LRU 큐의 2%~10% 내에서만 디스크를 써야 합니다.

경미한 업데이트 로드에 대한 LRU 매개변수 설정

업데이트 로드가 가중된 시스템에 대한 좋은 설정이 경미한 업데이트 로드 및 가중된 업데이트 로드가 모두 있는 시스템에 대해서도 좋은 설정이 될 수는 없습니다. 약간의 업데이트 로드와 넉넉한 디스크 및 CPU 자원을 사용하는 경우, LRU_MAX_DIRTY를 40%로, LRU_MIN_DIRTY를 15%로 설정하는 것이 적절합니다. 이는 대부분의 경우에 시스템이 체크포인트 시간에서 단지 15%만 채운 큐를 작성하고 체크포인트 사이에서 25%의 큐를 정리한다는 것이라는

의미입니다. 이 경우, 업데이트 로드가 가볍기 때문에 LRU 알고리즘은 효과적으로 업데이트 로드를 버퍼 큐 전체에 확산시킵니다. 그리고 체크포인트 사이에는 소수의 큐만 정리되어야 합니다. 체크포인트가 발생하면 15%~40% 사이에서 정리해야 할 큐가 일부 있을 수도 있습니다. 넉넉한 시스템 자원이 사용 가능한 상태일 경우, 이는 체크포인트 중에 관리될 수 있습니다. (체크포인트에 너무 오랜 시간이 소요되는 것 같으면, LRU_MAX_DIRTY 설정을 30 또는 25로 줄여 더욱 밀접한 델타를 생성할 수 있습니다.)

의사 결정 지원 시스템에 대한 LRU 설정

의사 결정 지원 시스템에는 OLTP 시스템과는 다른 트랜잭션 프로파일이 있습니다. 의사 결정 지원 시스템에는 업데이트가 발생하지 않고 뒤에 긴 실행 쿼리가 수행되는 시간이 오는 데이터 로드 작업에 대한 매우 거대한 업데이트 트랜잭션이 있습니다. 이러한 다수의 시스템에서 사용자가 시스템에 없을 때, 긴 체크포인트가 그리 중요하지 않은 시점에서 업데이트가 발생할 수 있습니다. 데이터 로드 프로세스 중 긴 체크포인트가 사용자(시스템에 있지 않는)에게 영향을 주지 않는 경우에도, 이러한 튜닝이 처리량을 개선시키고 로드 시간을 최소화하므로 효과적으로 필수 작업을 수행하기 위해 LRU 버퍼 관리를 튜닝하는 것이 중요합니다.

DSS 시스템에서는 LRU_MAX_DIRTY를 60%로, LRU_MIN_DIRTY를 20%로 설정해도 충분합니다. 이는 가중된 업데이트를 수행 중에도 잠재적으로 20%에서 60% 더티 사이의 체크포인트에서 많은 큐가 정리되도록 합니다. 이것은 길지만 관리 가능한 체크포인트를 정기적으로 생성할 수 있으나, 자원에 대한 경쟁이 거의 없는 상태에서 처리량에는 거의 영향을 주지 못합니다.

이 설정을 통해 처리량을 조절한 경우, LRU_MAX_DIRTY를 더욱 낮은 수준으로 설정하면 더 많은 버퍼 페이지가 체크포인트 사이에서 쓰여지게 됩니다. LRU_MAX_DIRTY를 40%로 설정하면 더 많은 클리너 스레드 활동이 실행되며 잠재적으로 더 짧은 체크포인트를 제공합니다.

LRU 튜닝을 위한 반복 벤치마킹

많은 다른 시스템 구성 매개변수와 마찬가지로 LRU 매개변수 설정을 위한 간단한 공식은 없습니다. LRU 설정은 분량을 정하기가

어려운 플랫폼 고유의 변수(디스크 수, 디스크 속도, 사용 가능한 메모리 양)에 따라 다릅니다. 플랫폼 요인의 가변성에는 반복 벤치마킹을 통한 플랫폼 기능을 발견하려고 시도하는 튜닝에 대한 접근이 필요합니다. 벤치마크 프로그램은 LRU 매개변수의 영향을 받는 엔진 자원에 부담을 주도록 개발되어야 합니다. LRU 매개변수가 업데이트를 수행하는 방식에 영향을 주기 때문에 벤치마크 프로그램은 엔진에 부담을 주는 방식으로 엔진을 업데이트해야 합니다.

벤치마크 프로세스는 벤치마크를 실행한 후 결과를 평가하는 식의 반복적인 프로세스여야 합니다. 필요한 경우 엔진 매개변수를 조정하고, 벤치마크를 재실행하며, 결과를 다시 한번 평가합니다. 이 프로세스는 성능이 최적화되었다고 간주될 때까지 반복됩니다.

이 벤치마크 프로세스는 중요합니다. 엔진 매개변수는 변경의 영향을 평가하지 않은 상태에서 변경해서는 안 됩니다. 반복적인 벤치마크 실행은 매개변수의 작고 섬세한 변경을 수행하며, 벤치마크를 수행한 후 결과를 평가하여 최선의 결과를 산출합니다.

체크포인트 평가 및 튜닝

앞서 언급한 대로, IDS 엔진은 공유 메모리 버퍼를 디스크와 동기화하고 고정된 복구 지점을 제공하기 위하여 체크포인트 작업을 수행합니다. 체크포인트 작업은 시스템의 무결성을 위해 필수이지만, 체크포인트가 발생하는 동안 체크포인트로 인해 엔진 코드가 중요 영역으로 들어갈 수 없습니다. 이는 실제로 체크포인트 수행 중 업데이트를 중단시키므로 긴 체크포인트가 엔진을 멈춘 것처럼 보여질 수 있습니다. 과도한 체크포인트 작업 활동은 시스템 처리를 느리게 하고 성능을 지연시킬 수 있습니다.

실제로 체크포인트 간격은 보통 OLTP 시스템에서, 5분에서 30분 사이로 설정되지만(CKPTINTVL 구성 매개변수로 설정한 대로), 확정값이 아닙니다. 특정 시스템의 값 설정은 해당 시스템 요구에 따라 달라져야 합니다.

체크포인트는 머신 고장 시 일관성 있는 상태로 시스템을 복구할 수 있는 시점을 표시합니다. 시스템이 예기치 않게 다운된 경우 짧은 체크포인트 간격은 더 긴 체크포인트 간격보다 더 많은 작업을 보존합니다. 시스템이 5분의 체크포인트 간격을 제공하는데 체크포인트 시간 전 1분 동안 실패한 경우, 4분간의 시스템 활동만이 잠재적으로 상실될 수 있습니다. 그러나 시스템의 체크포인트 간격이 30

분이고 체크포인트 전 1분 동안 실패한 경우, 시스템 활동의 29분이 잠재적으로 상실됩니다. (시스템 활동은 로그되지 않은 데이터베이스에서는 완전히 상실됩니다. 트랜잭션 로깅이 있는 데이터베이스에서는 단지 완료되지 않은 트랜잭션만 상실됩니다.) 체크포인트 간격이 더 길어지면 급속 복구가 최종 체크포인트에서 복구하므로 급속 복구에 필요한 시간을 늘리는 영향도 갖게 됩니다.

그러나 5분마다 체크포인트가 발생하는 시스템에서는 잠재적으로 체크포인트가 발생할 때 매 5분의 시스템의 저하가 발생합니다. (적절한 LRU 관리는 체크포인트의 오버헤드를 상당히 줄일 수 있지만 일부 성능 저하가 발생할 수도 있다는 점에 유의하십시오.)

배치 시스템에는 OLTP 시스템과는 상당히 다른 트랜잭션 프로파일이 있으므로, 온라인 시스템과 동일한 복구 문제는 발생하지 않습니다. 배치 프로세스는 종종 배치 프로세스를 단순히 재실행함으로써 복구할 수 있습니다. 30분 이상의 체크포인트 간격은 이러한 유형의 시스템에 적합할 수 있습니다.

업데이트에 집중적인 모든 시스템은 사전 이미지를 물리 로그에 쓰도록 함으로써 IDS에 부담을 줍니다. 물리 로그를 75%까지 채우면 체크포인트가 강제 실행되므로, 가중된 업데이트 로드와 협력하는 긴 체크포인트 간격은 많은 체크포인트를 실행시키고 긴 체크포인트 간격(구성 매개변수 CKPTINTVL에서 설정한 대로)의 문제를 발생시킵니다. (물리 로그를 75% 채움으로써 가중된 업데이트 로드가 5분마다 체크포인트를 실행하는 경우, 30분의 체크포인트 간격에는 결코 도달할 수 없습니다.)

쿼리 및 응용프로그램의 튜닝

특정 쿼리의 성능을 추적하는 것은 응용프로그램의 성능 병목 현상을 찾는 매우 효과적인 방법입니다. 쿼리 성능을 추적에는 쿼리를 실행하고 onstat 유틸리티를 사용하여 쿼리의 실행을 추적하는 것이 포함됩니다(onstat 출력에서 세션으로 식별됨).

onstat 명령을 사용하여 쿼리의 실행을 추적하면 쿼리가 액세스한 테이블, 테이블 및/또는 인덱스에서 데이터를 읽기 위해 사용한 스캔 스레드의 수, 사용된 임시 테이블 및 할당된 메모리 양을 식별할 수 있습니다. 이 정보를 사용하여 다음 절에서 설명하는 대로 이 쿼리의 최적화 방법을 결정합니다.

‘onstat -g ses’를 사용하여 쿼리 세션 추적

쿼리 세션의 활동을 추적하기 위해 두 가지의 onstat 매개변수를 사용할 수 있습니다. ‘-g ses <session id>’ 매개변수를 사용한 onstat 명령은 특정 세션에 대한 정보를 나열하며, onstat ‘-g sql <session id>’는 처리 중인 SQL문에 대한 정보를 제공합니다.

‘onstat -g ses’ 를 사용하여 세션 활동 상태 보기

-g ses 옵션을 사용한 onstat 명령은 해당 세션 또는 실행 중인 특정 세션에 대한 정보를 표시합니다. 이 명령에 대한 출력은 다음과 같습니다.

```
INFORMIX-OnLine  Version  7.22.WC1     - On-Line - Up 00:05:07 - 9024  Kbytes
session                                          #RSAM     total       used
id        user      tty      pid     hostname    threads    memory     memory
13       informix    -       0        -            0          8192       4628
3        informix    -       0        -            0         16384      12256
2        informix    -       0        -            0          8192       4628
```

명령의 출력은 세션이 활성 여부에 상관없이 모든 세션의 세션 정보를 표시합니다. 각 라인은 세션 ID, 세션을 시작한 사용자, 사용자가 시스템에 액세스한 위치, tty(터미널 ID), 연결 프로세스의 프로세스 ID, 호스트 프로세스를 위한 머신의 호스트 이름, 사용자가 사용 중인 스레드의 수, 사용자에게 할당된 전체 메모리, 현재 사용 중인 총 메모리를 표시합니다. 이 명령에 표시된 컬럼이 표 5-15에 나열되어 있습니다.

이 명령이 생성한 세션 출력은 쿼리의 상태, 현재 사용되는 자원의 양에 대한 정보를 제공합니다. 터무니없는 메모리 양을 참조하는 모든 쿼리는 더욱 면밀하게 검토해야 합니다. 세션에 대한 추가 정보는 세션 ID를 매개변수(onstat -g ses <session id>)로서 onstat 명령에 전달하여 얻을 수 있습니다. 다음은 이 명령의 샘플 출력입니다.

표 5-15 onstat -g ses 명령 출력

컬럼	목적
Session	사용자 세션에 대한 내부 IDS 세션 ID
user	연결을 위한 운영 체제 사용자 이름
tty	사용자 터미널(존재하는 경우)의 운영 체제 터미널 ID
pid	사용자 클라이언트 세션의 운영 체제 프로세스 ID
hostname	사용자 세션이 시작된 호스트 이름
#RSAM threads	사용자 세션에 대하여 현재 작동 중인 스레드의 수
total memory	사용자 세션에 할당된 메모리
used memory	사용자 세션이 현재 사용 중인 메모리

```
INFORMIX-OnLine  Version  7.22.WC1     - On-Line - Up 00:08:03 - 9024  Kbytes

session                                          #RSAM       total       used
id       user      tty          pid    hostname threads   memory     memory
14     informix   ATAYLOR_    fff9b3c7    ataylor_ 1       49152      30432

tid      name     rstcb        flags     curstk      status
27     sqlexec   82efd0e4      Y-P-      756      82efd0e4 cond wait(netnorm)

Memory pools    count  1
name        class  addr      totalsize   freesize   #allocfrag   #freefrag
14          V      82ffa014  49152       18720      276          12

name          free          used          name          free          used
overhead       0            108           scb           0             80
opentable      0            3508          filetable     0             628
log            0            4180          temprec       0             1596
gentcb         0            7660          ostcb         0             2008
sort           0            52            sqscb         0             7520
rdahead        0            116           hashfiletab   0             276
osenv          0            1192          sqtcb         0             1348
fragman        0            160
```

```
Sess   SQL          Current          Iso  Lock        SQL    ISAM    F.E.
Id     Stmt type    Database         Lvl  Mode        ERR    ERR     Vers
14      -           testit           NL  Not Wait     0      0       7.22

Last  parsed  SQL  statement  :
  select  *  from  t1
```

-g ses <session_id> 명령의 출력에서 첫 번째 부분은 -g ses 명령과 동일합니다. 세션, 연결 터미널, 운영 체제의 프로세스 ID, 할당된 메모리 양 및 세션이 사용한 메모리 양에 대한 정보를 표시합니다.

출력의 다음 부분은 해당 세션에 대해 현재 실행 중인 스레드에 대한 정보를 표시합니다. 스레드 ID는 tid 컬럼과 스레드 이름(예: sqlexec)의 순서로 나열됩니다. restcb 컬럼은 스레드의 control block에 대한 주소를 나열합니다. 다음 네 개의 컬럼은 스레드가 사용 중인 메모리 양에 대한 정보를 제공합니다.

출력의 다음 섹션은 다음과 같습니다. 이 섹션은 해당 세션에 사용 중인 메모리에 대한 정보를 표시합니다. 나열된 다양한 이름은 IDS 메모리 구조의 내부 이름입니다.

```
Memory  pools   count  1
name            class addr       totalsize  freesize  #allocfrag  #freefrag
14              V     82ffa014   49152      18720     276         12

name            free           used          name          free         used
overhead        0              108           scb           0            80
opentable       0              3508          filetable     0            628
log             0              4180          temprec       0            1596
gentcb          0              7660          ostcb         0            2008
sort            0              52            sqscb         0            7520
rdahead         0              116           hashfiletab   0            276
osenv           0              1192          sqtcb         0            1348
fragman         0              160

Sess   SQL          Current             Iso  Lock        SQL    ISAM    F.E.
Id     Stmt type    Database            Lvl  Mode        ERR    ERR     Vers
14     -            testit              NL  Not Wait     0      0       7.22
```

```
Last parsed SQL statement :
  select * from t1
```

출력의 마지막 섹션은 실행 중인 SQL문에 대한 정보를 표시합니다. 출력에는 SQL문의 유형, 현재 데이터베이스, 격리 수준(Iso Lvl), 로크 모드, SQL문에 생성된 오류(존재하는 경우)가 포함되어 있습니다. 마지막 컬럼은 연결을 설정한 Informix 프론트엔드 클라이언트 소프트웨어 구성 요소(ESQL/C, Informix-4GL) 버전을 나타냅니다.

이 명령 출력은 세션에서 사용 중인 IDS 자원의 양을 결정하는 데 유용합니다. 이 정보는 문제 쿼리와 일반 자원 사용에 대해 IDS 시스템을 모니터하는 데 사용할 수 있습니다. 이 명령 출력의 첫 번째 섹션은 세션에 할당된 메모리와 스레드의 수를 표시합니다. 메모리와 같은 대량 시스템 자원을 사용하는 세션은 전체 엔진 성능을 제한하기 시작하는 런어웨이 세션(지옥의 쿼리)이 될 수 있습니다. onstat -g ses 명령을 실행하면 세션과 이 세션에 사용되고 있는 메모리 및 스레드를 식별할 수 있습니다. 그런 후 onstat -g ses <session id> 명령을 실행하면 실행 중인 쿼리가 나타납니다.

관리자는 이 쿼리를 검토하여 적합한지 여부를 결정합니다. 예를 들어, 6백만 행의 테이블에서 모든 행을 선택하는 쿼리는 의심을 받을 수 있습니다. 필수 쿼리일 수는 있으나 쿼리를 실행하게 되면 쿼리를 실행할 충분한 자원이 있는지, 언제 쿼리를 실행해야 하는지, 사용 가능한 다양한 기능을 사용하는 쿼리에 할당할 자원을 제한해야 하는지 여부 등의 많은 튜닝 문제가 제기됩니다. 다음 절에서는 이러한 주제에 대해 다룹니다.

'onstat -g sql' 명령

-g sql <session id> 매개변수와 함께 onstat 명령을 실행하면 현재 실행 중인 SQL문에 대한 정보가 표시됩니다. 이 명령은 시스템에서 실행 중인 SQL문에 대한 추가 정보를 제공합니다. 다음은 이 명령의 샘플 출력입니다.

```
INFORMIX-OnLine  Version  7.22.WC1     - On-Line - Up 00:07:19 - 9024  Kbytes
Sess   SQL          Current          Iso  Lock        SQL    ISAM    F.E.
Id     Stmt type    Database         Lvl  Mode        ERR    ERR     Vers
14     -            testit           NL   Not Wait    0      0       7.22

Last parsed SQL statement :
  select * from t1
```

이 명령의 출력은 표 5-16에 설명되어 있습니다.

onstat -g ses 명령에서와 마찬가지로 이 명령은 시스템에서 실행 중인 쿼리를 모니터링하는 방법과 이 쿼리의 현재 상태, 가장 중요한 쿼리의 오류 발생 여부에 대한 정보를 제공합니다. 관리자는 사용자가 실행 중인 쿼리에 오류가 발생했는지 여부와 오류가 발생한 경우 그 원인을 확인하기 위해 이 명령을 사용할 수 있습니다.

표 5-16 onstat -g sql 명령 출력

Session id	내부 IDS 세션 ID
SQL stmt type	SQL문의 유형
current database	명령문을 실행하는 현재 데이터베이스
Iso Lvl	SQL문에 대한 로깅 격리 수준
Lock Mode	명령문에 대한 로크 모드
SQL Err	명령문으로 생성된 최종 SQL 오류
ISAM Err	명령문으로 생성된 최종 ISAM 오류
F.E. Vers	세션에 대한 프론트엔드(클라이언트) 버전

'onstat -g act'를 사용하여 활성 스레드 모니터링

'-g act' 옵션을 사용하여 실행한 onstat 명령은 쿼리에 대한 활성 스레드의 유용한 정보를 표시합니다. 이 정보는 쿼리에서 발생한 병렬 처리의 양을 결정할 때 중요한 정보가 됩니다. 다음은 이 명령에 대한 출력입니다.

```
INFORMIX-OnLine  Version  7.22.WC1     - On-Line - Up 07:00:38 - 9024  Kbytes

Running  threads:
tid      tcb         rstcb      prty       status      vp-class  name
7        82f7d944    0          2          running     7soc      soctcppoll
35       8305a764    82efccb0   2          running     1cpu      sqlexec
```

이 예에서 엔진은 연결을 위한 TCP/IP 소켓을 리스닝하고 폴링하는 soctcpoll 스레드와 쿼리를 실행하는 sqlexec 스레드, 두 가지

활성 스레드를 관리합니다. 이 출력의 각 행은 특정 스레드에 대한 정보를 표시합니다. 각 행에 표시된 컬럼은 표 5-17에 나와 있습니다.

VP 클래스 정보는 스레드를 실행하는 가상 프로세서를 표시합니다. 이 명령은 IDS 시스템 전체가 CPU 자원을 사용하는 방법을 결정하는 데 유용합니다. 쿼리의 실행 중에 이 명령을 실행하면 쿼리(및 동시에 실행 중인 다른 시스템 활동)를 처리하는 데 사용되는 활성 스레드(현재 IDS가 실행하는 스레드)를 표시합니다. 예를 들어, 병렬 처리를 사용했던 쿼리를 실행한 경우, 이 명령을 실행하면 쿼리가 사용한 활성 스캔 스레드가 표시됩니다.

표 5-17 onstat -g act 명령 출력

tid	내부 IDS 스레드 ID 번호
tcb	스레드 제어 블록 주소
status	스레드 상태(-g act 옵션은 실행 중인 스레드만 표시)
vp-class	스레드를 실행하는 VP 클래스
name	스레드 유형의 이름

'onstat -g iof'를 사용하여 비동기 입출력의 모니터링

-g iof 옵션과 함께 onstat 명령을 실행하면 시스템에서 실행 중인 비동기 입출력(AIO) 스레드에 대한 정보가 표시됩니다. 다음은 이 명령의 출력 예입니다.

```
INFORMIX-OnLine  Version  7.22.WC1     - On-Line - Up 00:01:24 - 9024  Kbytes

AIO  global  files:
gfd  pathname                 totalops       dskread     dskwrite     io/s
3    *D:\rootdbs_dat.000       23             17          6            0.3
4    *d:\chunk1                5              5           0            0.1
```

이 명령은 매우 유용한 초당 입출력 출력(io/s)을 포함한 청크당 읽기 및 쓰기 작업(onstat -D 출력과 유사)에 대한 정보를 표시합니다. io/s 컬럼은 디스크의 특정 부분에 대한 처리량을 잘 나타냅니다. 특정 청크가 과다한 처리량을 발생하는 것처럼 보이면, 디스크

의 내용(청크/DBSPACE)을 검토해야 합니다. 대량의 입출력에 대한 타당한 이유가 있을 수도 있습니다. 예를 들어, 물리 로깅 또는 논리 로깅을 청크에서 수행하고 있거나 일정하게 액세스된 테이블이 청크에 위치할 수 있습니다. 그러나 상위 레벨의 입출력 또한 테이블을 이동 또는 분할하거나, 논리 로그 파일을 이동, 또는 물리 로그 위치를 다시 지정함으로써 해결할 수 있는 문제를 나타낼 수도 있습니다.

과다한 입출력은 또한 입출력 서브시스템이 처리량을 효율적으로 처리할 수 있는 지점 이상인 포화된 디스크의 부분을 나타낼 수도 있습니다. 이것이 경우에 해당하는지 여부는 사용 중인 입출력 서브시스템에 따라 달라지며, 해당 경우로 보이면 IDS 하드웨어 플랫폼에 대한 시스템 관리자가 검토해야 합니다.

시스템 자원 문제 해결

성능의 저하를 가져오는 시스템 자원 문제를 해결하는 가장 분명한 방법은 머신 자원을 추가하는 것입니다. 이것은 반드시 머신에 별도의 디스크나 CPU를 추가하는 것만을 의미하는 것이 아닙니다. 단지 시스템에서 이 자원을 사용할 수 있으면 성능 문제가 있는 IDS 인스턴스에서도 이 자원이 사용 가능해야 한다는 의미입니다.

데이터베이스 엔진에는 작업을 수행하는 데 필요한 상당한 양의 CPU 자원이 필요할 수 있습니다. 엔진에서 사용 가능한 CPU 자원은 사용 가능한 디스크 자원만큼 중요합니다. 실제적으로 엔진이 CPU에 제한되어 있는 경우가 많으므로, 사용 가능한 CPU 자원이 사용 가능한 디스크 자원보다 더 중요하다는 의미입니다. 머신의 CPU 자원은 테이블의 적절한 분할화를 통해 병렬 처리를 수행하는 경우 수요가 많을 수 있습니다.

현재 사용 가능한 가장 일반적인 하드웨어 플랫폼 중 하나는 응용 프로그램에 처리 능력을 제공하기 위해 여러 개의 CPU를 사용하는 대칭 멀티프로세서 플랫폼입니다. 사용되는 CPU의 수는 유연하므로 사용자가 지정된 한도만큼 필요한 CPU를 추가할 수 있습니다.

쿼리가 4개의 CPU만 있는 머신에 5개의 스캔 스레드가 필요한 경우, 쿼리를 완전히 병렬 처리할 수 없습니다. 이 경우, 추가 CPU를 머신에 추가하고 CPU를 데이터베이스 엔진에서 사용할 수 있게 하여 성능을 향상시킬 수 있습니다.

IDS 엔진은 CPU 가상 프로세서(VP)를 통해 CPU 자원을 인식합

니다. CPU VP는 onconfig 구성 파일에서 추가하거나 onmode 유틸리티를 통해 동적으로 추가할 수 있습니다. CPU VP는 NUMCPUVP 구성 매개변수를 변경하여 onconfig 파일에 추가할 수 있습니다. 이 매개변수는 시스템 시작 중에만 읽습니다. 따라서 엔진이 실행된 이후에 변경하여 CPU VP를 추가할 수 없습니다. NUMCPUVP는 다음에 엔진이 시작될 때 엔진에 대한 CPU VP의 수를 변경하는 숫자 매개변수입니다.

CPU VP의 추가

엔진이 온라인 상태일 때 동적으로 CPU VP를 추가하는 능력은 새로운 차원의 IDS 엔진에 대한 확장 기능을 첨가했습니다. 이 기능을 사용하여 컴퓨터 시스템의 로드에 따라 엔진을 확장할 수 있습니다. 다른 응용프로그램에 의한 시스템의 로드가 시스템 관리자가 로드의 균형을 이루고 다른 응용프로그램의 사용자에게 추가 시스템 자원을 제공해야 할 지점까지 증가한 경우, IDS가 사용하는 CPU VP의 수를 줄일 수 있으므로, 응용프로그램에 더 많은 CPU를 동적으로 제공하게 됩니다. 사용자 로드가 낮을 것으로 예상될 때(야간 처리 중에 가능) OLTP 시스템에서와 같이, IDS 엔진의 로드가 낮은 경우 이를 수행할 수 있습니다. onmode 명령을 사용한 VP 추가 구문은 다음과 같습니다.

```
onmode -p +/- <number to add/subtract> <VP Class>
```

VP class 매개변수는 표 5-18에 나열된 내용 중 하나입니다.

표 5-18 onmode로 전달된 VP 클래스

NET	네트워크 VP
AIO	비동기 입출력 VP
SHM	공유 메모리 리스너 VP
TLI	TLI 리스너 VP
SOC	TCP 소켓 리스너 VP
CPU	CPU VP

매개변수를 추가하거나 빼는 수는 정수여야 하며 64 이하이어야 합니다. 빼는 수로 인해 해당 유형의 VP 수가 엔진의 초기화 때 엔진을 사용한 수의 미만이면 안 됩니다.

엄격한 규칙은 아니지만 일반적으로 CPU VP의 수가 플랫폼에서 사용 가능한 실제 CPU 수를 초과해서는 안 됩니다. 어떤 경우(CPU VP의 수를 약간 초과하는 것이 시스템의 성능을 향상시키는 경우도 있습니다.)

CPU VP를 추가했을 때의 이익을 측정하려면, 벤치마크를 수행해야 합니다. 벤치마크 응용프로그램은 시스템 처리 중에 엔진에 지정된 쿼리의 근사치를 내도록 개발되어야 합니다. 응용프로그램은 쿼리 및/또는 업데이트 세트를 무작위로 서버로 보내야 하고, 최소한 5분 간격(더 길면 좋음)으로 실행할 수 있어야 합니다.

CPU VP는 이 장에서 이전에 표시된 대로 onmode 명령을 사용하여 조금씩(한 개씩 또는 두 개씩을 선호) 추가되어야 합니다. CPU VP를 추가한 이후 벤치마크 프로그램을 다시 실행하여 추가 CPU 자원이 영향을 주었는지 여부를 결정해야 합니다. 효과가 더 이상 늘어나지 않으면(CPU VP 추가가 영향을 미치지 않으면), 최적의 CPU VP 수를 찾은 것입니다.

일단 최적의 CPU 수를 찾았으면 IDS 시스템은 온라인이 되어야 하고 새 구성 설정을 사용하여 생산 모드로 실행되어야 합니다. 또한 CPU 사용을 모니터하여 시스템에서 CPU VP에 대해 균등하게 분산되었는지를 결정해야 합니다.

쿼리 튜닝

응용프로그램을 구성하는 쿼리는 응용프로그램이 데이터에 어떻게 논리적으로 액세스하는지에 대한 가장 분명한 그림을 제시합니다. 일단 논리적 데이터 액세스가 알려지면 Informix 서버는 데이터에 대한 물리적 액세스에서 일정 수준의 제어를 허용합니다. 이 제어를 개발하는 첫 번째 단계는 SQL 확장 set explain 명령을 사용하여 논리 데이터 액세스를 결정하는 것입니다.

set explain 명령은 SQL문으로서 실행됩니다. 일단 이 명령을 실행하면 데이터베이스 세션의 기간(사용자가 데이터베이스에 연결되는 기간) 동안 명령이 활성화됩니다.

'set explain' 명령은 사용자 세션의 현재 작업 디렉토리에서 'sqexplain.out'이라는 ASCII 파일에 쿼리를 처리하는 데 사용할 쿼리

계획에 대한 정보를 작성합니다. 세션이 'set explain off'를 사용하여 'set explain' 옵션을 해제할 때까지, 현재 작업 디렉토리를 사용하는 모든 사용자 세션은 이 파일을 계속 추가합니다. 다음은 'set explain' 출력을 작성하는 SQL문의 예입니다.

```
set   explain  on;

select  *
from   sales_facts  s,  time_dim  t,  region  r
where  s.month   =  t.month  and
       s.region  =  r.region
order  by  s.region  ;

set  explain  off;

select  *
from   po_facts  p,  time_dim  t,  region  r
where  p.month   =  t.month  and
       p.region  =  r.region
order  by  p.region  ;
```

위의 일련의 SQL문은 sales_facts 테이블에서 데이터를 검색하는 첫 번째 SQL문에 대한 set explain 출력을 생성합니다. 이 명령문을 실행하면 즉시 sqexplain 출력이 해제되기 때문에 po_facts 테이블에서 선택한 목록의 두 번째 SQL문은 set explain 출력을 생성하지 않습니다.

set explain 명령의 출력에는 IDS 옵티마이저가 쿼리에 대해 생성했던 계획의 설명이 들어 있습니다. sqexplain.out 파일의 출력에는 실행되었던 쿼리, 쿼리 처리에 사용할 스레드의 수, 각 테이블에 액세스하는 데 사용할 필터 기준 및 인덱스가 포함되어 있습니다. 이 출력을 사용하여 다음 사항을 결정할 수 있습니다.

1. 병렬 처리량
2. 데이터 테이블에 대한 액세스 경로
3. 조인 필드
4. 집계 필드
5. 정렬 필드

쿼리에 대한 병렬 처리 레벨은 시작된 스레드의 수에 따라 결정됩니다. 시작된 스레드의 수는 다음을 포함한 많은 요인에 의해 결정됩니다.

1. 쿼리의 테이블 수
2. 쿼리를 구성하는 테이블에서의 분할화 양
3. 쿼리의 필터
4. 쿼리의 집계
5. 현재 유효한 PDQ 우선 순위

IDS 옵티마이저는 타당하다고 간주되는 병렬 처리만을 수행합니다. IDS 옵티마이저는 2개의 읽기 프로세스 사이에서 단순히 디스크 경쟁을 발생시킨다는 가정 하에, 같은 DBSPACE에 위치한 동일한 테이블 및/또는 조각에서 읽어 두 개의 병렬 스캔 스레드를 시작하지 않습니다. 그러나 테이블이 여러 개의 DBSPACE에 걸쳐 분할화된 경우, IDS 옵티마이저는 이러한 여러 개의 조각에서 병렬로 읽을 때 디스크 경쟁이 없다고 가정합니다.

동일한 디스크로 분할화

IDS는 여러 개의 조각이 물리적으로 별도의 디스크를 표시한다고 가정합니다. 또한 IDS는 디스크의 기본적인 물리 구조를 검토하지 않습니다. 관리자에게 동일 디스크에 위치하는 청크에 매핑하는 별도의 DBSPACE가 두 개 있으면, 디스크 경쟁을 유발할 수 있습니다. 예를 들어, 어떤 시스템에 대해 다음과 같이 DBSPACE가 정의되어 있는 경우를 살펴봅시다.

```
dbspace1       chunk1 (/dev/disk1 - offset 0)
dbspace2       chunk2 (/dev/disk1 - offset 50K)
```

관리자는 다음과 같이 `table1`이라는 테이블을 작성하고 `dbspace1`과 `dbspace2`에 테이블을 분할합니다.

```
 create table table1
(col1 int,
 col2 char(10),
 col3 char(10))
```

```
fragment by round robin in
    dbspace1, dbspace2;
```

다음 select문은 테이블 전체를 효과적으로 강제 스캔하는 table 1에서 모든 행을 검색하도록 실행됩니다.

```
select *
from table1
order by col1
```

PDQPRIORITY를 충분히 설정하여 옵티마이저는 이 쿼리에서 테이블 table1로부터 데이터를 읽기 위하여 두 개의 스레드를 시작합니다. 한 스캔 스레드는 dbspace1에서 읽고 다른 한 스캔 스레드는 dbspace2에서 읽습니다. DBSPACE가 별도의 청크로 매핑하는 경우에도, 청크는 동일한 물리적 디스크로 매핑하기 때문에 이 작업은 디스크 경쟁을 유발할 수 있는 읽기 작업입니다. 이 문제를 방지하기 위하여 분할화 계획에서는 가능하면 별도의 디스크에서 DBSPACE를 사용해야 합니다.

필터와 조각 제거

Informix 서버는 필터 기준을 사용하여 어떤 조각을 스캔할 것인지 결정할 수 있습니다. 이 과정을 조각 제거라고 합니다. 몇 개의 DBSPACE에 대한 표현식으로 테이블이 분할화된 경우, 조각 표현식에서 컬럼을 사용하는 필터 기준으로 옵티마이저가 조각을 제거할 수 있습니다. 다음은 이에 대한 예입니다.

다음 SQL문을 사용하여 테이블이 작성된 경우를 예로 들어 봅시다.

```
create tablea  ( col1 int, col2 char(20) )
fragment  by  expression
           col1  >  0 and col1  <  200 in dbspace1,
           col1  >  200 and col1  <  300 in dbspace2,
           col1  >  300 and col1  <  400 in dbspace3,
           remainder  in  dbspace4;
```

다음 SQL문이 테이블에 대해 실행된 경우, 제거 가능한 조각이 있습니다.

```
select *
from tablea
where col1 in (10, 20, 210, 230);
```

이 쿼리에서는 dbspace3과 dbspace4의 조각이 검색에서 제거될 수 있습니다.

이전 예에서는, dbspace1의 컬럼 col1에 대해 값이 10 또는 20인 레코드와 dbspace2의 컬럼 col1에 대해 값이 210 또는 230인 레코드를 배치한 표현식이 테이블을 분할화했고, 쿼리가 특히 해당하는 컬럼 값만 있는 행을 요청했기 때문에, 검색에서 다른 모든 조각을 논리적으로 제거할 수 있습니다.

테이블이 표현식으로 분할화된 후 실행 중인 쿼리의 필터 기준을 사용하여 적절하게 참조된 표현식을 사용한 경우에만 조각 제거가 가능합니다. 조각 표현식에 두 개의 컬럼이 있으면 두 컬럼은 모두 조각 제거가 가능하도록 분할화 기준에서 참조되어야 합니다.

PDQPRIORITY 설정

PDQPRIORITY 설정은 무엇보다도 쿼리에 적용될 병렬 처리의 레벨에 영향을 줍니다. 옵티마이저는 이 값을 사용하여 쿼리에 적용될 자원을 결정합니다. 이 값을 낮게 설정하면 더 적은 자원을 할당하고 높게 설정하면 더 많은 자원을 할당합니다. 자원은 병렬 쿼리 수행에 사용되는 스캔 스레드의 형식과, 메모리 부여 관리자(MGM)가 연계된 메모리 형식으로 할당됩니다. 따라서 PDQPRIORITY가 0이면 병렬 처리를 산출하지 않고 쿼리의 모든 부분은 연속적으로 실행됩니다. 스캔, 조인, 정렬, 집계 및 그룹화를 포함한 여러 쿼리 작업은 병렬로 실행될 수 있습니다. 병렬 처리의 영향은 중요할 수 있습니다.

예를 들어, 임의의 쿼리가 어떤 필터 기준에 따라 테이블 스캔을 포함시킨 후 결과를 정렬하고 집계한 경우, 쿼리의 스캔, 정렬, 집계 부분을 병렬로 처리할 수 있습니다. 그러나 PDQPRIORITY가 0으로 설정된 경우 IDS 옵티마이저는 먼저 실행한 데이터를 스캔했던 쿼리 경로를 작성합니다. 그런 후 스캔이 완료되면 결과를 정렬하고, 이 정렬이 완료되면 결과를 집계합니다. 각 단계를 연속적으로 실행

하면 다음과 같은 시간이 산출됩니다.

프로세스	시간(분)
스캔	3
정렬	1
집계	1
총계	5

각 단계는 연속적으로 실행되므로 처리를 위한 총 시간은 각 단계에 필요한 시간의 합이 됩니다. 그러나 각 단계가 병렬로 실행되면, 쿼리 완료를 위한 시간은 가장 긴 단계를 완료하는 데 필요한 시간과 대략 같습니다. 따라서 선행 쿼리의 경우 각 단계에 대해 별도의 처리 스레드와 이 선행 쿼리가 병렬로 실행되면, 처리 시간은 5분이 아니라 3분이 됩니다. (병렬 처리에는 시작하고, 처리를 동기화한 후, 처리를 종료하는 오버헤드가 있으므로 이는 근사치입니다.)

예를 들면, 주문 입력 시스템은 주문 테이블과 구성 항목 저장에 사용하는 테이블 사이에 상위-하위 테이블 관계로 구성되어 있습니다. 각 주문에 대한 총 가격을 구하려면, 주문 번호별로 주문한 항목의 가격을 집계해야 합니다. 다음은 이 집계를 수행하는 쿼리입니다.

```
select orders.order_num, sum(total_price)
from order, items
where orders.order_num = items.item_num
group by 1;
```

이 쿼리를 실행하면 다음 출력에 표시된 대로 각 주문에 대한 집계 가격이 제공됩니다.

```
order_num              (sum)
     1010             $84.00
     1015            $450.00
     1006            $448.00
     1017            $584.00
     1016            $654.00
```

```
1002            $1200.00
1007             $940.00
1011              $99.00
```

이 쿼리의 set explain 출력은 쿼리를 처리하기 위해 선택한 쿼리 경로를 설명합니다. 쿼리가 읽는 테이블, 인덱스의 사용 여부, 테이블 조인에 사용되는 조인의 유형 및 기타 관련 쿼리 경로 정보에 대한 설명입니다. 다음은 이 쿼리의 sqexplain 파일에 대한 샘플 출력입니다.

```
QUERY :
select orders.order_num, sum(total_price)
from orders, items
where orders.order_num = items.order_num
group by  1

Estimated Cost:  79
Estimated # of Rows Returned: 1
Temporary Files Required For: Group By

1) informix.orders: SEQUENTIAL SCAN (Serial, fragments: ALL)

2) informix.items: SEQUENTIAL SCAN

DYNAMIC HASH JOIN (Build  Outer)
  Dynamic Hash Filters: Informix.orders.order_num = Informix.item.order_num

select orders order_num, sum(total_price)
from orders, items
where orders.order_num = item.order_num
group by 1

Estimated Cost: 94
```

이 set explain 출력은 orders 테이블과 items 테이블 간의 조인이 동적 해시 조인을 통해 수행된다는 것을 보여줍니다. orders 테이블 인덱스는 인덱스를 검색하기 위해 읽혀집니다. (쿼리에는 orders 테이블의 다른 데이터가 필요하지 않습니다.) 그런 후, 해시 테이블을 통해 인덱스에서 검색한 주문 번호를 항목 테이블의 행에 조인

합니다(이로 인해 해시 조인이라는 이름이 지정되었음).

set explain 출력의 다음 라인은 쿼리가 병렬로 처리되지 않는다는 것을 보여줍니다.

```
1) informix.orders: SEQUENTIAL SCAN (Serial, fragments: ALL)
```

병렬 처리를 지원하는 플랫폼에서 괄호 안의 정보는 쿼리가 연속으로 처리되는지 병렬로 처리되는지를 표시합니다. 이 출력은 연속 처리를 사용한다는 것을 나타냈으므로 병렬 처리는 발생하지 않습니다. PDQPRIORITY를 0으로 설정하면 강제로 모든 스캔을 연속으로 실행하게 됩니다. PDQPRIORITY를 0보다 큰 값으로 설정하면 쿼리에 생성된 쿼리 경로에 영향을 주고 스캔 작업을 포함하여 쿼리의 부분이 병렬로 실행될 수 있습니다.

PDQPRIORITY의 설정 및 제어

PDQPRIORITY의 값은 사용자 세션에 대한 설정할 수 있는 환경 변수 또는 SQL문으로 설정할 수 있습니다. 환경 또는 SQL 세션에서 설정되지 않은 경우, PDQPRIORITY의 기본값은 0입니다. 표 5-19에서 설명한 대로 많은 구성 매개변수가 PDQPRIORITY에 영향을 줍니다. (이 구성 매개변수는 4장에서 자세하게 다룹니다.)

표 5-19 PDQ 구성 매개변수

매개변수	설명
DS_MAX-QUERIES	허용되는 최대 동시 PDQ 쿼리 수
DS_MAX_SCANS	허용되는 최대 동시 PDQ 스캔 수
DS_TOTAL_MEMORY	PDQ 쿼리로 할당되는 최대 메모리 양
MAX_PDQPRIORITY	허용되는 최대 PDQPRIORITY 설정
OPTCOMPIND	옵티마이저가 사용할 선호하는 조인 유형

PDQPRIORITY가 환경 변수에서 설정된 후 SQL문에서 설정되면, SQL문 설정은 환경 변수의 설정을 무시합니다. MAX_PDQPRIORITY 설정은 개별 세션의 PDQPRIORITY 설정을 제한합니다. MAX_PDQPRIORITY가 50으로 설정되고 세션이 PDQPRIORITY를 80

으로 설정하면, 옵티마이저는 PDQPRIORITY 설정을 50으로 대체합니다.

이전에 언급한 대로 PDQPRIORITY를 0으로 설정하는 것은 병렬 기능을 사용하지 않는다는 의미입니다. 1을 설정하면 쿼리를 처리하는 데 병렬 스캔만 사용할 것임을 옵티마이저에 알리는 것입니다. 2에서 100까지 설정하면, 표 5-19에서 표시된 환경 변수 설정의 제한을 받아서 병렬 쿼리 자원의 일정 비율을 쿼리 처리에 사용하게 됩니다.

병렬 쿼리 처리의 사용은 주로 의사 결정 지원 쿼리로 인식되는 쿼리, 여러 행을 처리하고 집계를 수행하는 쿼리, 정렬과 같은 일부 쿼리에 상당한 영향을 줄 수 있습니다. OLTP 유형의 쿼리의 경우에는 병렬 처리가 거의 긍정적인 영향을 주지 않으므로 이 쿼리를 권장하지 않습니다. 이러한 이유로 인해 IDS를 사용하는 OLTP 시스템에 대해 PDQPRIORITY를 0으로 설정(기본값)할 것을 권장합니다.

의사 결정 지원 응용프로그램의 경우, PDQPRIORITY를 2와 100 사이의 값으로 설정할 것을 권장합니다. 특정 쿼리에 대한 다양한 설정의 영향은 쿼리의 테이블 분할화, 사용 가능한 IDS 자원 및 표 5-19의 구성 매개변수 설정에 따라 달라집니다.

또는 실험과 DSS 쿼리를 장기 실행하는 반복 벤치마킹을 권장합니다. DSS 쿼리를 실행하고 결과를 /dev/null로 출력하는 SQL문 외에 아무 것도 할 수 없는 벤치마크 응용프로그램 생성과 다양한 값으로 설정된 PDQPRIORITY 환경 변수를 사용한 벤치마크의 실행은 쿼리에 대한 최적의 PDQPRIORITY 설정을 나타낼 수 있습니다.

허용되는 경우, PDQ 쿼리가 상당한 부분의 IDS 자원을 사용할 수 있으며 다른 DSS 쿼리의 처리를 엄격하게 제한할 수 있다는 점에 유의하십시오. 이러한 이유로 인해 표 5-19의 구성 매개변수는 PDQ에 사용되는 자원의 양을 제한하도록 설정되어야 합니다. 이러한 매개변수는 onmode 명령을 사용하여 IDS 엔진 온라인으로 설정될 수 있으므로 특정 기간 동안(예: 생산 시간 동안) PDQ 자원의 사용을 제한하고 시간외 배치 처리 기간에는 PDQ 자원의 사용을 증가시킵니다. 표 5-20은 PDQ 자원의 조정에 사용할 수 있는 onmode 명령의 목록입니다.

표 5-20 onmode PDQ 설정

매개변수	설정
-M	DS_TOTAL_MEMORY 값 변경
-Q	DS_MAX_QUERIES 값 변경
-D	MAX_PDQPRIORITY 값 변경
-S	DS_MAX_SCANS 값 변경

OPTCOMPIND 매개변수를 사용하여 최적화에 미친 영향

Informix는 두 개 이상의 테이블 간 조인 및 병합에 관련된 옵티마이저에 대한 힌트를 주는 환경 변수를 제공합니다. 쿼리가 조인할 여러 테이블을 필요로 하는 경우(여러 테이블에서 관련 행을 검색), 조인 수행에 사용 가능한 몇 가지 방법이 있습니다. 환경 변수는 OPTCOMPIND이며, 사용 가능한 설정 및 각각의 영향에 대해서는 표 5-21에 설명되어 있습니다.

표 5-21 테이블 조인 방법

옵션	설명
0	비용에 상관없이 옵티마이저가 인덱스 스캔(중첩 루프 조인) 선택
1	격리 모드가 RR(Repeatable Read)이면, 옵티마이저는 2로 설정한 경우와 동일하게 작동. 그렇지 않은 경우, 옵티마이저는 0으로 설정된 것처럼 작동.
2	옵티마이저가 비용을 사용하여 최적의 쿼리 경로를 결정. 기본값: 2

인덱스 읽기 대 연속 읽기

테이블을 무작위로 읽을 때(외부 테이블로의 인덱스 액세스를 사용한 중첩 루프 조인에서의 경우와 같음), 디스크 읽기에는 비용이 많이 듭니다. 실제 디스크 입출력에서는 읽기 헤드가 디스크 전체를 이동해야만 하기 때문에 대기 시간이 늘어나고 읽기 수행에 필요한 시간이 늘어납니다. 비용 기반의 옵티마이저는 이 비용을 인식하고 무작위 디스크 읽기 작업에 해당하는 가중치를 추가합니다. 이러한

이유로 옵티마이저는 테이블의 일부 또는 모든 필터 기준을 만족시키기 위해 인덱스를 사용할 수 있는 경우에도 종종 테이블의 연속 스캔 수행을 선택합니다.

미리 읽기 옵션

IDS 엔진은 전체 테이블 스캔 동안 몇 페이지에 미리 읽기 기능을 제공합니다. 미리 읽기 기능은 앞의 몇 페이지를 미리 읽고, 특히 쿼리 프로세스가 요청하기 전에 이러한 페이지를 공유 메모리 버퍼에 배치합니다. (엔진은 다음 입출력 요청을 효과적으로 예측합니다.) 이로 인해 쿼리 프로세스는 디스크 입출력을 기다려야 하는 시간의 양을 감소시키게 됩니다.

미리 읽기 기능은 RA_PAGES와 RA_THRESHOLD, 두 개의 구성 매개변수를 사용하여 구성됩니다. RA_PAGES 매개변수는 엔진이 미리 읽기 작업을 수행할 때 디스크에서 읽을 페이지의 수를 지정하고, RA_THRESHOLD 매개변수는 IDS 엔진이 다른 미리 읽기를 수행하기 전에 메모리에 남아 있어야 할 처리되지 않은 페이지의 수를 지정합니다. `onstat` 명령은 `onstat -p` 옵션으로 미리 읽기 기능의 사용을 모니터하는 데 사용할 수 있습니다. 다음에 나열된 `onstat` 조각은 미리 읽기 사용을 참조합니다.

```
...
ixda-RA      idx-RA       da-RA       RA-pgsused
87           0            6           71
```

위 목록의 표제는 `onstat -p` 명령의 출력 가장 아래 행에 나열되어 있습니다. 이 표제의 의미는 표 5-22와 같습니다.

*표 5-22 **onstat** 미리 읽기 정보*

idxda-RA	인덱스로부터의 미리 읽기 계수가 데이터 페이지에 남습니다.
idx-RA	인덱스를 탐색하는 미리 읽기 계수가 남습니다.
da-RA	데이터의 계수만 스캔합니다.
RA-pgsused	사용된 IDS 미리 읽기 페이지 수입니다.

보고된 `RA-pgsused` 값이 모든 미리 읽기 페이지(`idx-RA + da-RA`)로

보고된 값보다 훨씬 적은 경우, 미리 읽기 매개변수가 너무 높게 설정되었을 수 있습니다. 이 옵션을 신중하게 사용하는 것이 중요합니다. 많은 페이지를 공유 메모리 버퍼로 읽어드리는 것은 LRU 관리에 부담을 주어 사용 가능한 버퍼 공간으로 다수의 페이지 플러시를 강요할 수 있습니다. 이로 인해 실제로 성능이 저하됩니다.

성능 향상을 위해 옵티마이저 지시문 사용:
Informix Dynamic Server 버전 7.3의 릴리스에서 옵티마이저 지시문을 사용하여 쿼리 계획의 선택에 관련한 옵티마이저에 추가 힌트를 제공할 수 있게 되었습니다. 7장 "관계형 데이터베이스 및 Informix SQL"에는 IDS 엔진에 옵티마이저 힌트를 사용하는 방법에 대한 자세한 정보가 나와 있습니다.

쿼리 및 쿼리 계획

1970년대와 1980년대에 널리 보급되었던 계층 구조 데이터베이스와 네트워크 데이터베이스와는 달리, 관계형 데이터베이스에는 디스크의 데이터 테이블 간의 물리적 관계에 대한 정의가 포함되어 있지 않습니다. 관계형 데이터베이스에서 데이터 액세스 경로는 데이터베이스의 일부로서 정의되어 있지 않습니다. 대신 이 액세스 경로는 데이터베이스에서 데이터를 요청할 때 결정되어야 합니다. IDS 데이터베이스에서 IDS 옵티마이저는 이 기능을 수행해야 합니다.

IDS 옵티마이저는 쿼리가 요청하는 데이터에 액세스하는 최선의 계획을 결정하는 데 내부 알고리즘을 사용합니다. 이 계획을 쿼리 계획이라고 합니다. 쿼리 계획을 작성하려면, 옵티마이저는 쿼리를 평가한 후 다음과 같은 몇 가지 사항을 결정해야 합니다.

- 액세스 계획
- 조인 계획
- 평가 순서

위의 목록은 짧지만 평가 수행에 포함된 논리는 매우 중요합니다. 다음 절에서는 평가 과정 전반의 자세한 정보가 나와 있습니다.

액세스 계획

옵티마이저는 테이블에 액세스하는 방법에 관한 두 가지 기본사항, 즉 테이블에 액세스하기 위해 인덱스 읽기를 사용할 것인지 또는 테이블을 연속적으로 읽을 것인지(즉, 테이블 스캔 수행)를 결정해야 합니다.

인덱스 사용 여부에 대한 결정은 많은 요인에 따라 달라질 수 있습니다. 테이블의 많은 부분을 읽어야 하는 경우, 인덱스와 데이터 페이지를 읽는 것은 단순히 실행할 입출력 작업의 수를 증가시키므로 인덱스의 사용이 반드시 유리하지만은 않습니다. 테이블에서 많은 행을 읽을 때, 인덱스 읽기를 사용하는 것보다 연속 스캔을 사용하면 더욱 효율적으로 수행할 수 있습니다.

병렬 처리(테이블이 분할되어 있는 경우) 및 디스크 미리 읽기 작업, 그리고 디스크로부터 연속 데이터 페이지 읽기의 일반적 효율성을 조합하여 IDS 엔진과 함께 매우 효율적인 연속 스캔을 수행할 수 있습니다.

조인 계획

쿼리에서 사용한 테이블은 쿼리의 where절에서 등식 명령문을 사용하여 조인됩니다. 테이블은 쿼리의 조인 계획으로 알려진 여러 가지 다른 방식으로 물리적 조인이 수행될 수 있습니다. 조인 계획은 중첩 루프 조인 또는 해시 조인 중 어느 한 조인 방식을 가리킵니다. 중첩 루프 조인은 상대적으로 적은 수의 레코드를 액세스하고 있을 때 더욱 빠른 것으로 간주되며 이와 반대로 해시 조인은 다수의 레코드를 액세스하고 있을 때 더욱 빠르다고 간주됩니다.

중첩 루프는 평가순으로 외부 테이블로도 알려진 첫 번째 테이블을 스캔함으로써 두 개의 테이블을 조인합니다. 그런 후, 외부 테이블에 필터를 전달하는 각 행을 내부 테이블에서 찾습니다. 인덱스를 사용하여 내부 테이블 및 외부 테이블을 모두 스캔할 수 있습니다. 이와 같은 방법이 필수는 아니지만 더 선호되는 액세스 방법입니다. 내부 테이블에 인덱스가 없는 경우, 쿼리 실행 시점에서 자동 인덱스를 사용하여 테이블을 생성할 수 있습니다. 자동 인덱스의 생성부는 쿼리 계획을 평가할 때 옵티마이저가 결정합니다.

해시 조인

해시 조인에는 해시 테이블을 사용하여 두 개의 테이블을 조인하는 것이 포함되며, 이 조인은 두 값 사이에서 신속하게 일치하는 값을 찾기 위한 메커니즘입니다. 이 조인 방법의 중요한 장점은 인덱스와 정렬이 필수가 아니라는 것입니다.

해시 조인은 해시 테이블이 빌드되는 빌드 단계와, 조인에서 두 번째(외부) 테이블을 읽고, 적절한 필터가 적용되며, 조인 레코드에 대해 해시 테이블을 검사하는 검사 단계의 두 가지 활동으로 구성됩니다. 더 작은 해시 테이블은 공유 메모리의 동적 부분에 위치할 수 있으므로 해시 조인의 성능이 향상됩니다.

평가 순서

쿼리에서 둘 이상의 테이블이 읽혀지고 있는 경우, 옵티마이저는 쿼리에서 테이블을 읽는 순서를 결정해야 합니다. 순서는 테이블을 읽는 비용에 따라 평가하고, 테이블의 행 수, 테이블을 읽는 컬럼 인덱스의 존재, 인덱스의 선택 빈도, 필터할 테이블의 데이터 분산 및 기타 요인으로 결정됩니다.

비용 기반의 옵티마이저 및 '**set optimization**' 명령

IDS 옵티마이저는 비용 기반의 옵티마이저로서 사용 가능한 각 대체 액세스 계획에 대한 통계적 비용을 산출합니다. 이 비용은 필수적인 입출력 작업 수, 결과 산출에 필요한 계산, 옵티마이저가 액세스할 행의 수, 필요한 모든 정렬, 사용 가능한 인덱스의 선택 빈도 및 기타 관련 기준과 같은 요소에 따라 달라집니다. 옵티마이저는 기본적으로 모든 가능한 액세스 경로를 평가하여 각 경로에 비용을 지정합니다. 그런 후 옵티마이저는 가능한 가장 낮은 비용을 사용하여 액세스 경로를 선택하고 해당 액세스 경로를 사용합니다.

set optimization 명령을 사용하여 기본 최적화를 무시할 수 있습니다. 이 명령의 구문은 다음과 같습니다.

```
set optimization high | low | first rows | all rows
```

set optimization high 옵션을 사용하면, 옵티마이저는 모든

가능한 액세스 경로를 철저히 검토합니다. 이는 기본 최적화 모드입니다.

또한 set optimization low 옵션을 사용하면, 옵티마이저는 쿼리 경로를 평가하는 데 철저한 알고리즘을 사용하지 않습니다. 대신에 가능성이 낮은 많은 액세스 후보를 최적화 프로세스의 초기에 제거합니다.

first rows와 all rows 옵션은 쿼리 목표에 적용됩니다. 쿼리 목표가 all rows를 검색하는 것이면, 최적화는 궁극적으로 모든 행을 가능한 한 효율적으로 요청자에게 리턴할 쿼리 계획을 생성합니다.

쿼리 목표가 첫 번째 행으로 설정되면, 최적화는 첫 번째 행 세트를 가능한 신속하게 최적으로 되돌리는 쿼리 계획을 형식화하려고 합니다.

쿼리 튜닝

응용프로그램에서 하나의 쿼리 또는 쿼리 세트를 튜닝하는 것은 시스템의 성능 및 효율성 모두에 상당한 영향을 줄 수 있습니다. 또한 특정 쿼리의 튜닝에는 기본적인 물리적 데이터베이스 구조와 논리적 데이터베이스 구조, 즉 쿼리의 테이블 설계와 테이블에 대한 분할화 스키마에 반영된 테이블의 물리적 레이아웃의 검토가 포함됩니다. 이러한 고려사항 외에도 테이블에 대한 인덱싱 방법도 검토해야 합니다. 종종 단일 조인 컬럼에 대한 인덱스의 추가는 몇 가지 크기의 순서에 따라 쿼리 성능을 증가시킬 수 있습니다.

전체 테이블 스캔의 식별 및 문제 해결

전체 테이블 스캔은 쿼리의 요구를 만족시키기 위해 테이블의 모든 행을 읽어야 한다고 IDS 옵티마이저가 결정했을 때 존재합니다. 전체 테이블 스캔이 필수인 몇 가지 경우가 있지만 대부분의 경우 이렇게 비용이 많이 드는 작업은 필수가 아니면 단지 쿼리를 위한 처리 시간을 연장시키기만 합니다.

전체 테이블 스캔은 쿼리 실행을 통해 생성된 sqexplain 출력에서 식별할 수 있습니다. 다음 sqexplain 출력에서는 한 예를 제시합니다.

```
QUERY:
------
select * from orders where order_date < "05/01/1995"

Estimated Cost: 2
Estimated # of Rows Returned: 3

1) informix.orders: SEQUENTIAL SCAN (Serial, fragments: ALL)
    Filters: Informix.orders.order_date < 05/01/1995
```

위의 예에서 테이블의 스캔을 위한 섹션을 보면 적용할 필터 기준은 있지만 인덱스의 목록은 없다는 것을 알 수 있습니다. 따라서 전체 테이블을 읽거나 스캔하여 쿼리를 충족시켜야 합니다. 즉 제거 프로세스로 전체 테이블의 스캔을 결정해야 합니다.

전체 테이블 스캔을 승인할 수 있는 경우

테이블의 모든 또는 거의 모든 행을 읽고 처리해야 하는 인스턴스가 있습니다. 예를 들어, 테이블에는 1년 간의 매출 거래의 자세한 정보가 포함되어 있는 경우, 쿼리는 4분기 동안의 매출 데이터 요약을 작성해야 할 수도 있습니다. 이 쿼리는 모든 이벤트에서 전체 테이블을 읽어야 하므로 이 인스턴스에서 인덱스를 사용하면 간단하게 쿼리 처리 오버헤드에 디스크 읽기를 추가하게 됩니다. 다음의 예를 살펴보십시오.

```
create index sales_qtr_idx on sales_facts ( sales_qtr ) ;
create index cust_id_idx   on sales_facts ( cust_id ) ;

select sales_qtr, sum(sales_amount)
from sales_facts
where sales_qtr in (1, 2, 3, 4)
group by sales_qtr;
```

이 예에서 sales_facts 테이블에는 sales_qtr 컬럼이 있으며, 인덱스가 이 컬럼에 빌드되어 있습니다. 실행 중인 쿼리는 컬럼에서 필터 기준을 사용합니다. 언뜻 보기에는 sales_qtr 컬럼에서 인덱스를 사용하는 것이 유용한 것 같습니다. 그러나 테이블 내용을 읽고 나면, 테이블에는 4분기만 저장되어 있기 때문에 전체 테이블을

읽어 쿼리 결과가 나왔음을 알 수 있습니다. sales_facts 테이블에 200,000개의 행이 있고 옵티마이저가 sales_qtr_idx 인덱스를 사용하여 테이블에 액세스하려고 결정한 경우, 쿼리는 200,000개의 인덱스 행을 읽고 테이블에서 200,000개의 행을 더 읽게 됩니다. 즉, 200,000개의 행을 추가로 읽는 것입니다.

이 쿼리는 요약과 집계를 위해 많은 행을 스캔하는 전형적인 DSS 유형의 쿼리입니다. 이러한 유형의 쿼리가 전체 테이블 스캔을 요구하는 것은 일반적입니다. 전형적인 OLTP 유형의 쿼리는 더 적은 행을 읽어야 하며 집계를 수행하지 않을 가능성이 높습니다. 다음은 sales 테이블에 대한 OLTP 쿼리의 예입니다.

```
select sales_qtr,  sales_id,  sales_name,  sales_amount
from sales_facts
where sales_qtr in  (1, 2)    and
       sales_id = 1234       and
       sales_amount > 10000;
```

이 쿼리는 ID가 1234인 영업 사원에 대해 1/4분기와 2/4분기의 $10,000 이상인 매출을 알아보려는 것입니다. 이는 상대적으로 적은 수의 행(전체 연도의 행을 스캔해야 할 DSS 쿼리에 대해 상대적)을 선택하는 매우 선택적인 쿼리입니다. 그러나 적절한 인덱스가 없는 경우, 엔진은 전체 테이블을 스캔해야 합니다. 이 쿼리는 적은 수의 행만을 검색하기 때문에 전체 테이블을 스캔하는 것은 효율적이 아닙니다. 그러나 적절한 인덱스가 없기 때문에 옵티마이저는 어쩔 수 없이 전체 테이블을 스캔했던 쿼리 경로를 선택해야만 합니다. 다음은 이전에 표시된 쿼리를 만족시키기 위해 사용할 수 있는 인덱스를 생성합니다.

```
create index sales_idx1 on sales(sales_id);
```

이 인덱스는 영업 사원 ID 번호(sales_id), 즉 영업 사원을 고유하게 식별하지만 회사의 모든 영업 사원에 대한 매출 목록인 sales_facts 테이블에 중복되어 있는 식별자를 사용하여 테이블에 액세스합니다. 그러나 중복된 식별자를 사용하더라도 주어진 분기에 주어진 영업 사원에 대해 관리 가능한 레코드가 수가 있습니다. sales id에 대한 인덱스를 사용하여, 쿼리 경로는 주어진 매출 ID의 모든 sales_facts 레코드를 읽을 수 있고 원하는 분기에 대한 레코드를 필터링할 수 있습니다. 만일 sales_facts 테이블과 충돌하는 쿼리가

보통 sales_id와 sales_qtr을 선택한다고 알려져 있으면, sales_id와 sales_qtr에 대한 복합 인덱스 생성으로 다음과 같이 쿼리에 대한 더욱 선택적인 인덱스를 제공할 수 있습니다.

```
create index sales_idx2 on sales( sales_id, sales_qtr );
```

데이터 분산 통계 및 IDS 옵티마이저

인덱스의 연속 스캔을 선택하기 위한 스레시홀드는 20에서 40%이며 쿼리가 테이블의 20에서 40%를 읽어야 하는 경우, 인덱스에 대해 연속 스캔을 선택해야만 한다는 의미입니다. 옵티마이저는 자체적으로 이 결정을 해야 합니다. 하지만, 실제로 테이블을 스캔하기 전에 얼마나 많은 테이블의 행을 읽을 것인지를 결정하는 것은 어렵습니다. IDS 옵티마이저는 update statistics 명령이 생성하고 저장한 통계를 따릅니다. 4장에서 자세히 설명한 대로 update statistics 명령은 테이블의 각 컬럼을 스캔하고 컬럼의 값 범위에 대한 통계를 저장합니다. 그러나 update statistics 명령이 저장한 자세한 정보의 레벨은 update statistics 명령을 실행할 때 선택한 자세한 정보의 레벨에 따라 달라질 수 있습니다. 자세한 정보의 레벨로는 HIGH, MEDIUM, LOW가 있습니다.

update statistics 명령이 LOW 레벨에서 실행되면 옵티마이저가 올바른 결정을 하는 데 부적절한 통계가 저장될 수 있습니다. 또한 어떤 테이블에서는 데이터의 변하기 쉬운 특성으로 인해(데이터가 광범위하게 연속적으로 업데이트되는 경우), 최종 통계 세트가 생성된 이후 데이터 분산이 변경되었을 수 있으므로, 옵티마이저는 더 이상 유효하지 않은 통계를 사용하여 작업하고 잘못된 결정을 내리게 됩니다.

어느 경우든지 옵티마이저가 쿼리 경로 생성에 있어 잘못된 결정을 내린 것으로 보일 경우, HIGH 또는 MEDIUM은 레벨에서 update statistics 명령을 사용하여 통계를 다시 생성함으로써 옵티마이저에 유용한 자세한 정보가 생성되게 합니다. 데이터베이스에 일정하게 업데이트된 테이블 세트가 포함되어 있으면, 주별 또는 어떤 경우에는 매일 밤 정기적으로 update statistics 명령을 실행하는 것이 최선일 수 있습니다.

쿼리 튜닝을 위해 테이블 분할화 사용

Informix 엔진의 가장 강력한 기능 중 하나는 데이터의 정적 분할화를 제공하면서 이를 지원하는 병렬 처리 기능입니다. 의사 결정 지원 쿼리 동안 많은 양의 데이터를 스캔 중일 때 병렬 처리에서 얻는 성능상의 이점은 상당히 클 수 있습니다. 이와 마찬가지로, 데이터 경쟁이 문제가 될 수 있는 OLTP 시스템에서 정적 분할화 기능으로 단일 테이블의 데이터를 여러 디스크 및/또는 여러 개의 제어기에 분산할 수 있습니다. 이러한 분산은 특정 디스크 또는 제어기에 대한 부담을 줄이고 여러 장치로 입출력을 분산시킵니다. 정적 분할화가 없으면, 동일한 목표를 위해 디스크 스트라이핑과 같은 덜 강제적인 메커니즘을 사용해야 합니다.

분할화 스키마

분할화를 작업하려면 적절한 분할화 스키마를 채택해야 합니다. 분할화 스키마는 테이블의 데이터 위치를 지정합니다. 스키마는 표현식 기반의 분할화(가장 일반적이고 종종 가장 효과적임)와 마찬가지로, 테이블에서 컬럼을 포함하는 표현식에 기반을 두거나, 라운드로빈 분할화에서처럼 지정된 디스크 세트에 걸쳐 균등한 행 분산에 기반을 둘 수 있습니다.

대부분의 경우, 성능 향상을 위해 분할화를 사용하지만, 일부의 경우에는 보관 및 복구 목적으로 데이터를 분산시키는 데 사용합니다. 이는 테이블 내의 정적인 이전 연도 데이터가 단일 DBSPACE로 분산되고 변하기 쉬운 현재 데이터가 다른 DBSPACE에 보관된 경우에 해당할 수 있습니다. 이전 연도의 정적인 데이터 보관 및 복구가 월별 기준일 경우(데이터베이스의 전체 보관 동안), 현재 데이터는 일별 그리고 주별로 보관 할 수 있습니다. 그러나 보다 일반적인 분할화 사용은 주로 장기 실행 쿼리에서 성능을 향상시키는 것입니다. 이 경우, 데이터는 보통 데이터 검색을 위한 필터 기준의 일부인 컬럼을 포함하는 분할화 스키마에 기반을 두고 여러 디스크에 걸쳐 분산되어 있습니다. 목표는 여러 디스크에 걸쳐 데이터를 균등하게 분산시키는 것이며 입출력 경쟁을 가능한 한 많이 줄이는 것입니다. 다음의 데이터 테이블은 분할화 스키마 예를 기초로 구성된 것입니다.

```
create table invoice_facts(
    inv_number int,
    inv_date date,
    inv_flag char(2),
    ship_to_code char(5),
    inv_amount decimal(12,2));
```

이와 같은 축약된 사실 테이블에는 데이터 웨어하우스의 스토리지에 대해 OLTP 시스템에서 가져온 송장에 대한 정보가 포함되어 있습니다. 표현식 기반의 분할화 스키마에서 사용할 후보 컬럼으로 inv_date, inv_flag, ship_to_code 컬럼이 있습니다. 이러한 컬럼에는 다수의 중복된 값이 있으므로 표현식 기반의 분할화 스키마에 사용할 수 있습니다. 많은 다른 스키마가 가능하며 다음 절에는 이러한 스키마에 대한 예가 나와 있습니다.

inv_date별 분할

송장 월별 데이터 분할화에는 여러 장점이 있습니다. 첫 번째는 데이터 웨어하우스 시나리오에서 시스템의 DBSPACE 대부분은 상대적으로 정적인 데이터를 포함한다는 점입니다. 현재 월 데이터가 현재 월 DBSPACE로 이동하고 있기 때문에 이전 월 데이터는 주별 또는 월별 기준으로 보관 가능한 상대적으로 정적인 DBSPACE에 상주합니다. 다음은 invoice_date를 기준으로 한 분할화 스키마를 보여줍니다.

```
create table invoice_facts(
    inv_number int,
    inv_date date,
    inv_flag char(2),
    ship_to_code char(5),
    inv_amount decimal(12,2));

fragment by expression

  month(inv_date) in (1,2,3) in dbspace1,
  month(inv_date) in (4,5,6) in dbspace2,
  month(inv_date) in (7,8,9) in dbspace3,
```

```
month(inv_date) in (10,11,12) in dbspace4,

remainder in dbspace6;
```

이 예에서, 분할화 스키마는 송장 날짜의 월을 기준으로 합니다. month(inv_date) 함수는 해당 연도의 월에 대하여 1과 12 사이의 숫자를 분석합니다. 분할화 스키마는 특정 DBSPACE에 주어진 월 세트 데이터를 배치합니다. 이 접근 방식의 추가적인 이점은 성능입니다. 몇 개의 DBSPACE에 걸쳐 데이터를 분산시킴으로써 병렬 처리가 가능해집니다. 여러 개의 스캔 스레드를 시작하여 제공된 쿼리에 대한 데이터를 리턴하기 위해 각각의 DBSPACE에서 데이터를 읽을 수 있습니다. 표현식 기반의 분할화 스키마를 사용했으므로, 쿼리 경로로부터 필요 없는 조각을 제거하기 위하여 조각 제거 기능을 사용할 수 있습니다.

inv_flag별 분할

inv_flag 필드에서 분할 표현식 사용을 포함하는 분할 방법으로 시스템의 DBSPACE 전반에 데이터를 분산시킬 수 있습니다. 이 방법에서는 DBSPACE 전반에 걸쳐 데이터를 균등하게 분산시키는 것이 어렵습니다. 이를 위해서는 시스템의 데이터 특성을 알고 있어야 합니다. 다음은 분할화 스키마의 한 예입니다.

```
create table invoice_facts(
    inv_number int,
    inv_date date,
    inv_flag char(2),
    ship_to_code char(5),
    inv_amount decimal(12,2));

fragment by expression
    inv_flag = "A" in dbspace1,
    inv_flag = "B" in dbspace2,
    inv_flag  in ( "C", "D", "E") in dbspace3,
    inv_flag = ( "F", "G", "H") in dbspace4,
    inv_flag = "I" in dbspace5,
    remainder in dbspace6;
```

이 분할화 방법은 inv_flag별로 DBSPACE 전반에 걸쳐 데이터를 분산시킵니다. 즉, 한 개의 DBSPACE에 둘 이상의 inv_flag 설정을 할당하여 균등한 데이터 분산을 생성하는 것입니다. 이 방법은 데이터의 균등한 분산이 inv_flag를 통해 이루어질 수 있고, 대부분의 쿼리가 필터 기준으로서 inv_flag를 사용하여 데이터를 요청한다는 확신을 갖고 있습니다. dbschema 명령(dbschema -d <data base name> -hd <table name>)을 사용하여 데이터 분산을 실행함으로써 관리자는 테이블에서 데이터 분산을 결정할 수 있고 실제로 균등한 데이터 분산을 제공하는 그룹화로 분할화 스키마를 개발할 수 있습니다.

ship_to_code별 분할

또 다른 분할화 방법은 ship_to_code의 표현식으로 분할하는 것입니다. 이 방법은 ship_to_code 컬럼의 내용에 따라 지정된 DBSPACE 세트에서 invoice_facts 테이블의 데이터를 분산시킵니다. 다음은 이 방법의 예입니다.

```
create table invoice_facts(
    inv_number int,
    inv_date date,
    inv_flag char(2),
    ship_to_code char(5),
    inv_amount decimal(12,2));

fragment by expression

    ship_to_code = "345" in dbspace1,
    ship_to_code = "456" in dbspace2,
    ship_to_code in ( "789", "888", "888") in dbspace3,
    ship_to_code = ( "921", "831", "366") in dbspace4,
    ship_to_code = "987" in dbspace5,
    remainder in dbspace6;
```

이 방법에서는 특정 Ship_to_codes를 사용하여 식별된 DBSPACE 전체에 invoice_facts table의 레코드를 분산시킵니다. 이 분산 스키마는 ship_to_code 값의 특성에 대한 비즈니스 지식에 기반을 두고 있거나

또는 dbschema 유틸리티(dbschema -d <database name> -hd <table name>)를 사용하여 데이터 분산을 실행했다고 가정하며, 이 분산은 스키마가 테이블에서 데이터 레코드의 상대적으로 균등한 분산을 생성할 수 있다는 것을 보여줍니다.

분할화와 RAID

RAID는 일련의 디스크 드라이브가 컴퓨터 시스템에 하나의 디스크 드라이브로 나타날 수 있는 하드웨어 기능입니다. 이 기능은 RAID1~5로 이름이 지정된 여러 구현 방식을 제공하며, 이 구현은 동일한 방법의 서로 다른 구현입니다. 하나의 디스크의 오류가 시스템에 투명하게 드러나기 때문에 오류 복구는 이러한 시스템에서 큰 장점입니다. 디스크 읽기 및 쓰기는 인터럽트되지 않고 계속됩니다.

그러나 RAID에는 디스크 스트라이핑(물리적으로 분리된 여러 디스크 부분에 데이터를 무작위로 분산시키는 방법)과 동일한 구현상의 문제가 있습니다. IDS는 데이터의 물리적 분산을 인식하지 못하므로 이러한 방식으로 분산된 데이터를 병렬 처리할 수 없습니다. 따라서 RAID를 사용해도 IDS에 대해 특별한 이점은 없습니다.

그러나 이것이 IDS에서 RAID의 사용을 방해하지는 않습니다. RAID를 투명하게 사용하여 일반 디스크와 마찬가지로 RAID 디스크에 디스크 청크를 매핑하면, RAID를 사용한 병렬 처리가 가능합니다. 일반 디스크인 경우에도 논리 디스크(DBSPACE)에 걸쳐 데이터가 분산되어야 하므로 IDS가 데이터에 대해 병렬 데이터 스캔을 제공할 수 있습니다.

요약

이 장에서는 IDS 데이터베이스의 모니터링 및 튜닝 프로세스에 대해 자세하게 다루었습니다. 전 세계의 Informix 데이터베이스 관리자가 애용하는 onstat 유틸리티는 엔진 운영에 많은 기본 정보를 제공합니다. 일련의 onstat -g 명령을 포함한 많은 기본 정보 화면을 사용함으로써, 다중 스레드와 내부 메모리 할당 작업에 더욱 구체적인 정보를 제공할 수 있습니다.

엔진 튜닝의 기본적인 두 가지 접근 방법은 엔진의 실제 작업을 튜닝하는 것과 특정 쿼리를 튜닝하는 것입니다. 이 두 접근 방식이

분명히 겹치기는 하지만 상당한 차이도 있습니다.

LRU 큐는 공유 메모리 버퍼에 배치되어야 하는 새 데이터 페이지에 대한 공간을 제공하기 위해 더티 페이지를 디스크에 쓰는 프로세스를 관리합니다. 이 장에서 설명한 LRU 큐의 튜닝 프로세스는 업데이트가 필요한 시스템에 상당한 성능의 향상을 가져옵니다.

onmode 명령을 사용하면 엔진이 작동 중일 때 특정 엔진의 튜닝 가능한 매개변수를 변경할 수 있습니다. 엔진을 재시작하는 오버헤드를 발생시키지 않고 튜닝 가능한 매개변수의 변경을 평가할 수 있는 편리한 방법을 제공합니다. 또한 시스템 로드와 일치시킬 수 있는 시스템 자원 사용을 변경하는 방법도 제공합니다.

IDS 엔진은 쿼리의 분할화, 즉 여러 논리 디스크(DBSPACE)에 걸친 쿼리의 분산에 따른 쿼리의 병렬 처리를 제공합니다. 테이블의 적절한 분할화는 성능상 상당한 이점을 가져옵니다.

다음 장 미리보기

다음 장에서는 IDS 데이터베이스의 관리 프로세스를 검토합니다. 엔진의 시작 및 중지, 논리 로그의 추가 및 삭제, 데이터의 보관 및 복구, 보안 관리 작업에 대해 다룹니다. IDS 엔진은 엔진이 온라인 상태일 때 여러 구성 매개변수를 변경하는 강력한 기능을 제공합니다. 이러한 변경 프로세스도 다음 장에서 다루게 됩니다.

6장

Informix Dynamic Server의 데이터베이스 관리

실제로 모든 상업용 데이터베이스 제품에서는 일정 양의 데이터베이스 관리가 필수이며, IDS 엔진도 예외가 아닙니다. 이러한 관리 노력의 일환으로 IDS에는 엔진의 시작 및 중지, 데이터베이스 및 로그의 보관 수행, 보안 관리 등의 기본적인 작업이 있습니다. 이 외에도 IDS에는 데이터베이스 구성 매개변수를 동적으로 변경하고 데이터베이스가 온라인일 때 엔진 자원을 제어하기 위한 기능도 포함되어 있습니다. 이러한 관리 기능을 이해하고 사용하는 것은 IDS 엔진의 활용도를 최대화하는 중요한 구성 요소입니다. 다음 절에서는 이러한 기능을 비롯한 기타 관리 주제에 대해 자세하게 논의하겠습니다.

데이터베이스의 시작 및 중지

IDS 데이터베이스 인스턴스는 `oninit` 명령을 사용하여 시작됩니다. 이 프로그램과 많은 내부 스레드는 IDS 엔진에서 완료된 대량의 작업을 수행합니다. `oninit` 명령은 사용자 `informix` 또는 `root`(Unix 수퍼 사용자)만이 실행할 수 있습니다. 표 6-1에는 이 명령으로 전달할 수 있는 옵션이 나열되어 있습니다.

표 6-1 oninit 명령 옵션

매개변수	설명
-p	임시 테이블 삭제
-s	대기 모드로만 가져옴
-i	디스크 초기화
-y	모든 질문에 예라고 응답

엔진 운영 모드

oninit 명령은 데이터베이스를 시작하여 작업 모드로 가져옵니다. IDS 데이터베이스에는 오프라인 모드, 대기 모드(단일 사용자), 온라인 모드(다중 사용자), 세 개의 작업 모드가 있습니다. 데이터베이스 오프라인 모드를 사용하여 수행할 수 있는 데이터베이스 작업의 수는 제한되어 있습니다. 또한 대기 모드 상태의 데이터베이스를 요구하는 작업(예: 논리 로그 파일의 삭제 작업)도 일부 있습니다. 여러 사용자가 데이터베이스에 액세스하고 유용한 작업을 수행하려면 데이터베이스는 반드시 온라인 상태이어야 합니다.

oninit 명령은 공유 메모리를 초기화하고 기존의 데이터베이스 인스턴스를 시작할 수 있습니다. 또는 공유 메모리와 디스크를 초기화하고 새 데이터베이스 인스턴스를 작성할 수 있습니다.

oninit 명령이 기존의 데이터베이스를 시작하면, 이 명령은 환경 변수와 구성 파일을 읽고, 엔진을 위한 공유 메모리 버퍼를 초기화하며, 엔진이 사용한 제어 테이블 및 시스템 예약 페이지를 공유 메모리에 배치합니다. 그런 후 급속 복구를 실행하고 엔진에서 필요한 기타 추가 하우스 키핑 및 시작 작업을 수행합니다. 모든 시작 작업이 완료되면, oninit로 전달된 매개변수가 지정한 작업 모드로 엔진을 가져옵니다. 기존의 데이터베이스를 시작하는 데 사용되는 명령에서는 다음과 같이 매개변수가 없이 oninit를 호출합니다.

```
oninit
```

선택적으로 oninit에서는 다음 명령을 사용하여, 모든 임시 테이블을 삭제하고 엔진을 대기 모드로 가져오도록 지시할 수 있습니다.

```
oninit -ps
```

새 데이터베이스 인스턴스를 초기화하기 위해, oninit 명령은 디스크(특히, root DBSPACE)의 초기화를 지시하는 -i 매개변수로 전달되어 비어 있는 새 데이터베이스 인스턴스를 작성합니다. -y 매개변수를 oninit로 전달되지 않았으면, 이 명령은 새 인스턴스를 계속 작성하기 전에 사용자에게 프롬프트를 표시합니다. 이 프롬프트를 표시하는 이유는 oninit 디스크 초기화에는 파괴적 속성이 있기 때문입니다. 즉, 이 명령을 사용하면 시스템 예약 페이지, 시스템 카탈로그 테이블 및 모든 구성 데이터베이스를 비롯한 기존의 모든 데이터베이스 인스턴스가 제거됩니다. 다음은 oninit를 사용한 데이터베이스 인스턴스 초기화의 예입니다.

```
oninit -i
```

oninit를 사용하여 디스크를 초기화하고 새 데이터베이스 인스턴스를 작성하려면, oninit -i 명령을 사용하십시오. -y 매개변수와 함께 사용하는 경우, oninit는 아무런 표시없이 모든 기존의 데이터베이스 인스턴스를 삭제하고 새 데이터베이스 인스턴스를 작성합니다. 말할 필요도 없이 이러한 사항은 생각지도 않은 데이터베이스 인스턴스에 커다란 해를 미칠 수 있으므로, 매우 신중하게 사용해야 합니다.

시작 문제

엔진 시작 도중 문제가 발생할 수 있습니다. 엔진을 시작하는 중, 엔진이 구성 파일의 내용을 이해할 수 없거나, 구성 파일을 찾을 수 없거나, sqlhosts 파일의 내용을 이해할 수 없는 경우, 시작이 진행되지 않습니다.

문제는 데이터베이스를 구성하는 DBSPACE에서 발생할 수도 있습니다. 시작 중 엔진은 데이터베이스 인스턴스의 모든 DBSPACE로부터 데이터를 항상 읽지는 않습니다. 즉, 시작에 필요한 정보 및 스토리지를 포함하는 DBSPACE(물리 로그, 논리 로그, 시스템 예약 페이지)만 읽습니다. 엔진 시작에 필요한 정보가 들어 있는 위치는 주로 root DBSPACE입니다. root DBSPACE를 읽을 수 없는 경우, 엔진을 시작할 수 없으므로, 백업에서 데이터베이스를 복구해야 합니다.

온라인 엔진 수정

엔진이 온라인 모드일 때 엔진 작업의 많은 면을 수정할 수 있습니다. onmode 명령은 이러한 옵션에 대한 명령행 액세스를 제공합니다. onmode 명령으로 전달할 수 있는 매개변수가 표 6-2에 나와 있습니다. 다음 절에는 이러한 명령이 설명되어 있습니다.

표 6-2 IDS onmode 옵션

옵션	설명
-a <kbytes>	공유 메모리 세그먼트 크기 증가
-b <version>	IDS 디스크 구조를 이전 버전으로 되돌림
-c	체크포인트 수행
-D <max PDQ priority allowed>	MAX_PDQPRIORITY 구성 매개변수를 지정된 값으로 설정
-d {standard|{primary|secondary <servername>}}	이 서버의 데이터 복제 서버 유형 설정
-m	엔진을 온라인 모드로 가져옴
-F	사용하지 않은 메모리 세그먼트를 해제
-k	엔진을 오프라인으로 가져오기
-l	다음 논리 로그 시작하기
-M <decision support memory in kbytes>	의사 결정 지원에 사용할 메모리를 킬로바이트로 설정
-n	공유 메모리 버퍼를 비상주로 설정(공유 메모리가 스왑되도록 허용).
-O	DBSPACEDOWN 매개변수가 완료되지 않도록 지정한 경우에도 체크포인트가 완료되도록 허용
-p <+ -#><class>	지정된 클래스의 가상 프로세서 추가 또는 삭제
-Q <max # decision support queries>	DS_MAX_QUERIES 구성 매개변수 지정
-z <sid>	지정된 세션 ID 제거

<계속>

표 6-2 IDS onmode 옵션

옵션	설명
-Z <address>	지정된 트랜잭션을 경험적으로 완료
-y	지정된 명령에 대해 확인을 요구하지 않음
-u	모든 세션을 죽이고 종료
-s	엔진을 단일 사용자 모드로 가져오기
-S <max # decision support scans>	DS_MAX_SCANS 구성 매개변수 설정
-r	공유 메모리 버퍼 캐시를 상주로 설정(공유 메모리를 디스크로 스왑할 수 없음)
-R	/INFORMIXDIR/etc/.infos.DBSERVERNAME 파일 재빌드

엔진을 오프라인 모드로 가져오기

-k 매개변수와 함께 onmode 명령을 사용하여 데이터베이스 엔진을 오프라인 모드로 가져올 수 있습니다. 이 명령(명령을 실행한 사용자에게 프롬프트 표시)은 실행 시, 모든 실행 중인 데이터베이스 트랜잭션을 중단하고 오프라인 모드의 엔진을 가져옵니다. 사용자 informix 또는 root만이 이 명령을 실행할 수 있고, 이는 일반적으로 다음과 같이 실행됩니다.

```
onmode -ky
```

위의 구문을 사용하여 onmode 명령을 실행하면 즉시 모든 실행 중인 스레드가 죽으며 엔진은 오프라인 모드로 바뀝니다. -y 옵션을 사용하면 사용자에게 작업 유효화를 요청하는 질문을 생략하고 엔진을 즉시 오프라인 모드로 가져갑니다. -y 매개변수가 생략되면 모든 사용자 세션을 중단한 다음, 계속하도록 사용자에게 다음과 같은 프롬프트가 표시됩니다.

```
onmode -k

This will take INFORMIX-OnLine OFF-LINE -
Do you wish to continue (y/n)?
```

사용자가 계속하도록 요청하는 프롬프트에 사용자가 “y”(예)라고 응답하는 경우, 엔진은 오프라인 모드로 이동하는 프로세스를 시작합니다. 사용자 스레드가 실행 중이면 사용자에게는 다음과 같이 해당 스레드를 중단하라는 프롬프트가 표시됩니다.

```
onmode -k
This will take INFORMIX-OnLine OFF-LINE -
Do you wish to continue (y/n)? y
There are 1 user threads that will be killed.
Do you wish to continue (y/n)?
```

프롬프트에 “y”(예)라고 응답하면, 사용자 스레드가 죽이고 엔진을 오프라인 모드로 가져옵니다.

단일 사용자 모드로 엔진 가져오기

‘-s’ 매개변수와 함께 onmode 명령을 사용하여 엔진을 온라인 모드에서 단일 사용자 모드로 가져올 수 있습니다. 일단은 단일 사용자 모드인 경우, 사용자 informix 또는 root만이 이 모드에서 엔진에 액세스할 수 있습니다. 다른 사용자는 onstat 명령과 같은 유틸리티를 사용하여 엔진 활동을 볼 수는 있지만 데이터베이스에 연결할 수는 없습니다. 이 데이터베이스 모드는 일반적으로 논리 로그의 추가 및 제거에 사용되거나 다른 관리 기능을 수행하는 데 사용되며, 이는 다음과 같이 입력하면 됩니다.

```
onmode -s
```

이 명령은 먼저 사용자에게 다음과 같이 명령을 실행하라는 프롬프트를 표시합니다.

```
onmode -s
This will perform a GRACEFUL SUHTDOW -
Do you wish to continue (y/n)?
```

사용자가 “y”(예)라고 입력하면, 엔진은 대기 모드로 바뀝니다. 그렇지 않으면 onmode 명령을 -y 옵션(onmode -sy)과 함께 입력할 수 있습니다. 그러면, 엔진은 프롬프트없이 대기 모드로 이동하는 프

로세스를 시작합니다.

-u 옵션은 현재 실행 중인 모든 사용자 세션을 죽입니다. 이 옵션은 "-s" 옵션과 함께 사용하는 유용한 매개변수로서 모든 실행 중인 세션이 죽었는지 확인합니다. 이는 다음과 같이 사용됩니다.

```
onmode -suy
```

이 명령을 실행하면, 실행 중인 모든 세션이 죽으며 엔진은 대기 모드로 바뀝니다.

온라인(다중 사용자) 모드로 엔진 가져오기

-m 매개변수와 함께 onmode 명령을 사용하여 엔진을 대기 모드에서 온라인 모드로 가져올 수 있습니다. 이 명령에서는 공유 메모리가 oninit 명령을 사용하여 이미 초기화되었다고 가정한다는 사실에 유의하십시오. 공유 메모리가 초기화되지 않았을 경우, onmode 명령은 오류를 리턴합니다. 엔진을 대기 모드에서 온라인 모드로 가져오기 위해 onmode 명령을 다음과 같이 사용할 수 있습니다.

```
onmode -m
```

위의 명령은 엔진을 대기 모드에서 온라인 모드로 가져옵니다.

tip *로그를 추가 또는 삭제하려면, 엔진은 대기 모드여야 합니다. 엔진을 온라인 모드에서 대기 모드로 가져와 로그를 추가 또는 삭제한 뒤, 다중 사용자 즉 온라인 모드로 다시 가져오는 것이 일반적입니다. 다음은 이러한 작업의 실행하는 데 사용되는 일련의 명령입니다.*

```
onmode -suy
onparams -a -d dbspace
onmode -m
```

이 예의 처음에서는 onmode -suy 명령을 사용하여 엔진을 단일 사용자 모드로 가져옵니다. 다음으로 onparams 명령을 사용하여 논리 로그를 추가합니다. 일단 로그가 성공적으로 추가되었으면, onmode -m 명령을 사용하여 엔진을 다중 사용자 모드로 리턴합니다.

CPU 자원의 추가 및 제거

-p 옵션과 함께 onmode 명령을 사용하여 엔진에 추가 CPU 자원을 적용하거나 제거할 수 있습니다. 이전에 설명한 대로 IDS 엔진은 가상 프로세서(VP)를 할당하는 방법을 통해 CPU 자원을 간접적으로 다양한 작업에 할당되게 합니다. 이러한 VP는 다음과 같은 onmode 구문을 사용하여 할당되거나 할당해제됩니다.

```
onmode -p +/- <number of VPs to add> <VP type>
```

VP type은 표 6-3에 표시된 유형 중 하나일 수 있습니다.

표 6-3 VP 유형

유형	설명
NET	네트워크 VP
AIO	비동기 입출력 VP
SHM	공유 메모리 리스너 VP
TLI	TLI 리스너 VP
SOC	TCP 소켓 리스너 VP
CPU	CPU VP

위의 구문을 사용하여 단일 CPU VP를 다음과 같이 추가할 수 있습니다.

```
onmode -p +1 CPU
```

이 명령 실행으로 CPU VP의 수가 엔진을 시작했을 때의 적절한 수 이하로 감소하지 않을 경우, 다음 명령을 사용하면 두 개의 CPU VP가 제거됩니다.

```
onmode -p -2 CPU
```

DSS 매개변수 변경

장기 실행 DSS 유형 쿼리의 실행에 영향을 주기 위해 여러 엔진 매개변수를 사용할 수 있습니다. DSS 쿼리의 최대 수, 동시에 실행 가능한 DSS 스캔, DSS 쿼리에 적용할 메모리 양을 조정하기 위해 매개변수를 사용할 수 있습니다. 허용된 최대 PDQPRIORITY 값 또한 onmode 명령을 사용하여 설정할 수 있습니다. 이 DSS 매개변수는 표 6-4에 나와 있습니다.

표 6-4 DSS 매개변수

매개변수	설명
-D <max PDQ priority allowed>	MAX_PDQPRIORITY 구성 매개변수를 지정한 값으로 설정
-M <decision support memory in kbytes>	의사 결정 지원에 사용할 메모리를 킬로바이트로 설정
-Q <max # decision support queries>	DS_MAX_QUERY 매개변수 설정
-S <max # decision support scans>	DS_MAX_SCANS 매개변수 설정

추가 CPU 자원을 사용할 수 있는 경우, IDS 엔진을 이 자원에 액세스하도록 허용함으로써 잠재적으로 성능에 상당한 영향을 줄 수 있습니다. 여러 개의 CPU를 사용할 수 있는 대칭 멀티프로세서 머신에서 이 CPU의 일부를 IDS 엔진에 할당할 수 있습니다. SMP 머신에 6개의 CPU가 있고 초기에 3개가 엔진에 할당되었으면, 두 개 또는 세 개의 추가 CPU를 엔진에 할당할 수 있습니다. (대부분의 플랫폼에서 운영 체제에는 하나의 CPU를 사용할 수 있고 나머지 CPU는 해당 플랫폼에서 실행 중인 응용프로그램에 할당됩니다.)

다음 명령에서와 같이 onmode 명령을 사용하여 MAX_PDQPRIORITY를 증가시킬 수 있습니다.

```
onmode -D 40
```

이 명령을 실행하면 MAX_PDQPRIORITY 값이 40으로 설정됩니다. 의사 결정 지원 쿼리에 할당할 메모리 양을 조정하기 위해 다음

명령을 실행할 수 있습니다.

```
onmode -M 20000
```

이 명령은 의사 결정 지원 쿼리에 사용할 최대 메모리 양을 20,000KB로 설정합니다.

onmode를 사용하여 MAX_PDQPRIORITY 설정을 변경하는 것은 DS_TOTAL_MEMORY 구성 매개변수를 변경하는 것과 동일한 효과를 갖습니다. 이 매개변수의 설정을 사용하면 온라인 트랜잭션 처리와 의사 결정 지원 모두에 사용되는 시스템을 조정하는 데 도움을 줄 수 있습니다. 사용 중인 OLTP 활동 기간 동안 MAX_PDQPRIORITY를 더 낮은 값으로 설정하여, 의사 결정 지원이 엔진에 대하여 갖는 과중한 부담을 줄이고 OLTP 처리를 위한 추가 자원을 사용하게 할 수 있습니다. OLTP 처리가 줄어들거나 존재하지 않는 경우, 의사 결정 지원 쿼리에 더 많은 자원을 할당하기 위해 MAX_PDQPRIORITY를 증가시킬 수 있습니다.

공유 메모리 매개변수의 변경에 onmode 사용

공유 메모리 매개변수를 변경하는 데 여러 onmode 옵션을 사용할 수 있습니다. 이러한 옵션은 표 6-5에 나와 있습니다.

표 6-5 공유 메모리 매개변수의 변경을 위한 onmode 옵션

매개변수	설명
-a <kbytes>	지정된 크기의 공유 메모리 가상 세그먼트 추가
-r	공유 메모리 버퍼 캐시를 상주로 설정(공유 메모리를 디스크로 스왑할 수 없음)
-F	사용하지 않은 메모리 세그먼트 해제
-n	공유 메모리 버퍼를 비상주로 설정(공유 메모리를 스왑할 수 있음)

‘onmode -a’ 를 사용하여 가상 공유 메모리 추가

-a 옵션과 함께 onmode 명령을 사용하여 가상 공유 메모리 세그먼트를 의사 결정 지원 쿼리 처리를 위해 IDS 인스턴스에 추가할 수 있습니다. 이 값은 양의 정수여야 하며 엔진에서 사용할 수 있는 메모리를 초과해서는 안 됩니다.

일반적으로 IDS 엔진은 사용 가능한 공유 메모리 세그먼트 수에 대한 운영 체제 한계에 도달하는 것을 방지하기 위해, DS_TOTAL_MEMORY가 지정한 정의된 한계까지 필요한 만큼의 가상 공유 메모리 세그먼트를 할당합니다. 이 매개변수는 엔진이 의사 결정 지원 쿼리에 다수의 더 작은 세그먼트(SHMADD 구성 매개변수에서 정의된 대로)를 할당하는 것을 방지할 수 있는 정도의 충분히 큰 세그먼트를 할당하는 데 사용되어야 합니다. 이 매개변수는 인스턴스에 추가될 메모리에 킬로바이트 값으로 전달됩니다. 다음은 이 옵션의 예입니다.

```
onmode -a 10000
```

이 명령은 데이터베이스 인스턴스에 10MB의 공유 메모리(10,000 KB)를 추가합니다.

‘onmode -r’을 사용하여 공유 메모리 버퍼를 상주 상태로 설정

-r 옵션을 사용하여 실행된 onmode 명령은 공유 메모리 버퍼 캐시를 상주로 설정합니다. 기본적으로 공유 메모리는(모든 Unix 메모리에서와 마찬가지로) 디스크로 스왑 가능합니다. 이 명령을 실행하여 공유 메모리 세그먼트를 디스크로 스왑할 수는 없습니다. 다음은 이 명령의 예입니다.

```
onmode -r
```

이 명령을 사용하면 공유 메모리 버퍼가 상주로 설정되지만, 공유 메모리 버퍼가 디스크로 스왑될 수는 없습니다.

-F 옵션과 함께 onmode 명령을 사용하여 사용하지 않은 메모리 세그먼트를 IDS 엔진에서 해제할 수 있습니다. 다음은 이 옵션의 실행 예입니다.

```
onmode -F
```

이 명령은 IDS 인스턴스의 가상 메모리 부분에 획득한 메모리 세그먼트를 해제하여 정기적으로 실행되어야 합니다. 이 명령이 실행 중일 때 일반적으로 약 1초 또는 2초 정도 성능 저하가 발생한다는 점에 유의하십시오.

-n 옵션을 사용하여 실행된 onmode 명령은 IDS 인스턴스 공유 메모리 버퍼를 비상주로 설정하므로 이 세그먼트는 디스크로 스왑될 수 있습니다. 메모리가 제한을 받는 서버에서 이 명령은 IDS 엔진에 사용 중인 메모리를 디스크로 제거할 수 있으며 다른 응용프로그램이 사용할 공유 메모리를 해제할 수도 있습니다. 다음은 이 명령의 예입니다.

```
onmode -n
```

이 명령을 실행하면 공유 메모리 버퍼 캐시가 즉시 비상주로 설정됩니다.

공유 메모리가 디스크로 스왑 중이면, 데이터베이스 엔진 및 다른 실행 중인 응용프로그램을 포함한 시스템의 성능은 현저하게 영향을 받습니다. Informix 공유 메모리를 상주로 설정하면 공유 메모리의 스왑을 방지할 수 있고 엔진에 대한 성능을 향상시킬 수 있습니다.

그러나 공유 메모리가 스왑 중이라는 사실은 전체적인 시스템에 대해 메모리의 제한이 있다는 뜻입니다. IDS 공유 메모리 버퍼의 로크는 시스템상의 다른 응용프로그램에 사용 가능한 메모리를 더 적게 남기고 이 메모리를 디스크로 스왑되도록 하며 다른 응용프로그램의 성능을 저하시킬 수 있습니다. 메모리 스왑 문제에 대한 유일한 해결은 시스템에 더 많은 물리 메모리를 추가하는 것입니다.

기타 onmode 명령

onmode 명령은 여러 추가 관리 작업을 수행하는 데 사용할 수도 있습니다. 이러한 작업은 다음과 같습니다.

옵션	설명
-l	다음 논리 로그로 이동
-Z	지정된 트랜잭션을 경험적으로 완료
-z	지정된 프로세스 죽이기
-R	이 서버의 복제 유형 설정
-d	/INFORMIXDIR/etc/infos.DBSERVERNAME 파일 재빌드

이러한 명령 옵션에 대해서는 다음 절을 참조하십시오.

'onmode -l' 명령

-l 옵션과 함께 onmode 명령을 사용하여 IDS 엔진을 다음 논리 로그로 강제 이동할 수 있습니다. 특정 SQL DDL(데이터 정의 언어)문은 다음 논리 로그를 사용할 때까지 인식되지 않습니다. 이 onmode 명령을 실행하면 IDS가 다음 로그로 이동되므로 엔진이 DDL문의 결과를 인식할 수 있습니다.

'onmode -Z' <transaction id> 명령

-Z 옵션이 있는 onmode 명령과 트랜잭션 ID를 포함한 추가 매개변수는 매개변수로서 전달된 공유 메모리 주소가 지정한 분산된 트랜잭션을 논리적으로 제거합니다. (공유 메모리 주소는 트랜잭션의 onstat -x 출력에서 사용 가능합니다.) TXTIMEOUT 매개변수가 지정한 시간이 트랜잭션에 대해 만료된 경우에만 이 명령을 사용할 수 있습니다. 사용자 informix 또는 root만이 이 명령을 실행할 수 있습니다.

'onmode -z' <pid> 명령

-z 옵션과 함께 onmode 명령을 사용하여 특정 프로세스를 제거할 수 있습니다. 이 명령에는 제거할 프로세스의 프로세스 ID를 표시하는 추가 인수가 필요합니다. (onstat -u 출력에서 사용 가능합니다.)

‘onmode -R’ 명령

-R 옵션과 함께 onmode 명령을 사용하여 특정 데이터베이스 유틸리티가 사용하는 Informix 정보 파일을 다시 작성할 수 있습니다. 인스턴스가 오프라인일 때 일부 유틸리티에서는 이 파일을 사용하여 IDS 인스턴스의 구성을 결정해야 합니다. 파일이 손상되었거나 부주의로 삭제된 경우, 이 명령을 사용하여 다시 작성할 수 있습니다.

‘onmode -d’ 명령

-d 옵션과 함께 onmode 명령을 사용하여 보조 옵션을 사용할 때에 지정한 데이터베이스 인스턴스 또는 현재 데이터베이스 인스턴스에 복제 유형을 설정할 수 있습니다. 이 명령의 구문은 다음과 같습니다.

```
onmode -d {standard|{primary|secondary <servername>}}
```

이 명령에서는 참조한 데이터베이스 인스턴스가 IDS 복제를 실행하고 있다고 확신합니다. 다음은 이 명령의 예입니다.

```
onmode -d standard
```

이 명령을 실행하면 현재 데이터베이스 인스턴스가 기본 복제로 설정됩니다. 현재 실행 중인 기본 인스턴스를 보조 인스턴스로 변경하려면 다음 명령을 사용하십시오.

```
onmode -d secondary remote_ids_instance
```

IDS 데이터베이스 보안

원래 Informix 엔진에서 사용한 보안 모델은 사용자 레벨이었습니다. 사용자 로그인 이름을 사용하여 테이블 액세스를 사용자에 지정했고, 사용자에게 테이블 및/또는 테이블 컬럼의 특정 작업을 수행하도록 사용 권한을 부여했습니다. 이 보안 모델에는 사용자 그룹

또는 역할이 포함되지 않았습니다.

이후의 IDS 버전은 사용자 역할이라는 사용 권한 유형을 추가했습니다. 특정 테이블과 관련된 사용 권한에 역할을 지정한 뒤, 사용자에게 역할을 지정합니다. 역할을 사용하면 데이터베이스에 대한 보안을 관리하는 프로세스를 매우 단순화할 수 있습니다.

IDS의 보안 또는 권한은 데이터베이스와 테이블에 존재합니다. 데이터베이스와 데이터베이스 내의 테이블과 관련된 사용 권한을 사용자에게 지정할 수 있습니다. 데이터베이스 레벨 권한으로는 resource, dba, connect가 있습니다. 테이블 레벨 권한으로는 select, update, delete, alter, index 또는 references가 있습니다. 이러한 사용 권한은 다음 절에서 자세하게 설명합니다.

데이터베이스 레벨 사용 권한

데이터베이스와 직접 관련된 사용 권한에는 여러 가지가 있습니다. 이러한 사용 권한으로 사용자는 데이터베이스에 연결하여 테이블을 데이터베이스에 추가하거나, 수퍼 사용자가 되어 데이터베이스에 대한 데이터베이스 관리자 권한을 가질 수 있습니다. 이러한 권한은 다음과 같습니다.

- Connect
- Resource
- dba

이러한 사용 권한은 사용자 또는 역할에 지정됩니다. 다음 절에서는 이 사용 권한에 대해 설명합니다.

데이터베이스 connect 권한

데이터베이스 connect 권한을 사용하여 사용자는 데이터베이스에 연결할 수 있는 사용 권한을 갖습니다. 그러면 사용자는 데이터베이스의 테이블에 액세스할 수 있고, 테이블에 대해 사용자에게 부여된 사용 권한에 따라 테이블에서 사용 권한을 추가, 업데이트, 삭제, 변경 또는 부여할 수 있습니다. 이 권한이 있는 사용자라고 해서 테이블, 인덱스, 또는 내장 프로시저를 데이터베이스에 추가할 수 있는 것은 아닙니다.

데이터베이스 resource 권한

resource 권한이 있는 사용자는 새 테이블, 인덱스 및 프로시저를 작성할 수 있는 권한과 connect 권한을 사용하여 사용자의 모든 기능을 수행할 수 있습니다. 이 권한 레벨은 개발자 또는 데이터베이스 관리자에게 적절한 권한입니다.

데이터베이스 dba 권한

dba 권한이 있는 사용자는 여러 가지 추가할 수 있는 권한과 resource 권한을 사용하여 사용자의 모든 기능을 수행할 수 있습니다. dba 권한이 있는 사용자는 다른 사용자에게 어떤 권한이라도 부여할 수 있고, 시스템 카탈로그 테이블의 어떤 작업이라도 수행할 수 있으며, 테이블 또는 뷰를 작성하여 다른 사용자를 소유자로 지정할 수 있고, drop database, drop distributions, start database 및 roll forward 데이터베이스 명령문을 실행할 수도 있습니다.

다음과 같이 grant문 구문을 사용하여 데이터베이스 레벨 사용 권한을 지정할 수 있습니다.

```
grant <database privilege> to <user name | PUBLIC>
```

일반적인 접근 방법은 public에 connect를 부여하고 난 후 지정한 사용자 또는 역할 세트에 테이블 레벨 사용 권한을 제한하는 것입니다. 따라서 각 사용자에게 단일 역할 또는 복수의 역할을 지정하거나 데이터베이스의 각 테이블마다 특정 권한을 지정할 수 있습니다. 명령문의 예는 다음과 같습니다.

```
grant connect to public ;
grant insert, delete, select on customers to jamesj, henryj;
```

이 예에서는 먼저 데이터베이스에 대한 connect 권한을 public (모든 사용자를 의미)에 부여합니다. 그런 후 다음 명령문에서는 고객 테이블에서 insert, delete 및 select 권한을 jamesj와 henryj 사용자에게 부여합니다.

사용자 역할의 작성 및 사용

현재 IDS 버전을 사용한 사용자 보안으로의 가장 일반적인 접근 방법은 사용자 역할을 작성하는 것입니다. 다음 구문에서는 create role문을 사용하여 역할을 작성합니다.

```
create role payroll;
```

이 명령문은 급여 부서 사용자에 대해 payroll이라는 역할을 작성합니다. 급여 부서의 사용자에게는 다음과 같이 grant문을 사용하여 이 역할의 사용 권한을 부여할 수 있습니다.

```
grant payroll to hannaht;
grant payroll to eric;
grant payroll to carolynt;
```

사용자에게는 둘 이상의 역할이 부여될 수 있습니다. payroll 사용자는 accounting과 personnel 역할에 액세스해야 할 수도 있습니다. 따라서 이 사용자 역할을 다음과 같이 지정할 수도 있습니다.

```
grant acctg to hannaht;
grant personnel to hannaht;
```

grant문을 실행하면 사용자는 해당하는 모든 역할을 사용할 수 있습니다. 그러나 사용하려고 한 역할을 지정해야만 역할을 사용할 수 있습니다. 사용자 역할은 set role문을 사용하여 각 데이터베이스 세션의 시작 시에 지정되어야 합니다. 사용자 역할을 payroll로 설정하고 동일 세션에서 이후에 사용자 역할을 accounting으로 전환하려면 위해 다음과 같이 명령문을 실행하십시오.

```
set role to payroll;
... <SQL statements to execute>
set role to accounting;
... <SQL statements to execute>
```

테이블 사용 권한

데이터베이스에서 작성된 각 테이블에는 이와 관련된 사용 권한 세트가 있습니다. 사용 권한은 insert, update, delete, select, index 또는 alter 중 하나입니다. 이러한 사용 권한은 표 6-6에 설명되어 있습니다.

표 6-6 테이블 사용 권한

사용 권한	설명
insert	테이블에서 행을 삽입(추가)할 수 있습니다.
delete	테이블에서 행을 삭제할 수 있습니다.
update(column list)	테이블의 모든 또는 일부 컬럼을 업데이트할 수 있습니다. 컬럼의 목록이 없는 경우 테이블의 모든 컬럼을 의미합니다.
select(column list)	테이블의 모든 또는 일부 컬럼을 선택할 수 있습니다. 컬럼의 목록이 없는 경우 테이블의 모든 컬럼을 의미합니다.
index	테이블에 인덱스를 작성할 수 있습니다. 사용자에게는 index 권한을 사용할 수 있는 resource 권한도 있어야 합니다.
alter	테이블을 변경할 수 있습니다. 사용자에게는 alter 권한을 사용할 수 있는 resource 권한도 있어야 합니다.
references(column list)	테이블에서 일부 또는 모든 컬럼에 대한 참조 제한 조건을 작성할 수 있습니다. 컬럼의 목록이 없는 경우 테이블의 모든 컬럼을 의미합니다.
all	사용자에게 모든 권한을 부여합니다.

insert와 delete문은 특정 컬럼에서 실행되지 않습니다. 이러한 명령문은 전체 행을 삽입하거나 삭제하면서 전체 행에 대해 실행되므로 사용 권한에 컬럼을 지정하지 않습니다. update, select, references문은 전체 행 또는 특정 컬럼에서 실행되므로 컬럼을 지정할 수 있습니다. 컬럼 목록이 제공되지 않으면, 모든 컬럼을 의미하는 것입니다.

grant문을 사용하여 테이블 사용 권한을 지정할 수 있습니다. grant문에 대한 구문은 다음과 같습니다.

```
grant <permission> on <table name> to <user list | role>
```

테이블 권한을 사용자 public에 부여하면 모든 사용자가 해당 테이블에서 이 권한을 사용할 수 있습니다. 예를 들어, 다음 명령문은 모든 컬럼에 대해 select 권한을 부여합니다.

```
grant select on sales_facts to public;
```

이 명령문은 sales_facts 테이블에 대한 특정 권한이 부여되었는지 여부에 상관없이 모든 사용자에게 테이블에서 행을 선택할 수 있는 권한을 주는 것입니다.

여러 사용자에게는 다음과 같이 단일 명령문으로 사용 권한을 부여할 수 있습니다.

```
grant insert on sales_facts to erict, hannaht;
```

이 명령문은 사용자 eric과 hannaht에게 sales_facts 테이블에 대한 insert 사용 권한을 부여합니다.

'with grant option' 명령문

with grant option문을 사용하여 사용 권한을 부여할 경우, 이 권한을 다른 사용자에게 부여하기 위한 사용 권한과 함께 지정된 권한이 사용자에게 부여됩니다. 다음 명령문은 사용자 johni에게 update 사용 권한을 부여합니다.

```
grant update on customers to johni with grant option;
```

사용자 johni는 다음과 같이 추가 권한에 update 권한을 부여할 수 있습니다.

```
grant update on customers to janetr, paulaj;
```

사용자 johni가 실행한 이 명령문은 사용자 janetr와 paulaj에게 customer 테이블에 대한 update 사용 권한을 부여합니다.

‘revoke’ 명령으로 사용 권한 제거

revoke 옵션을 사용하여 사용자에게 부여된 테이블 또는 데이터베이스 레벨 사용 권한을 제거할 수 있습니다. 사용자는 다른 사용자에게 사용 권한을 부여하거나 취소할 수 있는 dba 권한이 없는 경우, 사용자가 부여했던 권한만 취소할 수 있습니다. 다음 구문과 같이 revoke문은 특정 권한을 지정하고 사용자를 식별할 수 있습니다.

```
revoke insert on sales_facts from erict;
```

이 명령문은 사용자 erict로부터 sales_facts 테이블에 대한 insert 권한을 취소합니다. revoke문을 사용하여 다음과 같이 데이터베이스 권한을 취소할 수도 있습니다.

```
revoke connect from erict;
```

이 명령문은 사용자 erict로부터 connect 권한을 취소합니다. 사용자 erict에게 데이터베이스로 연결할 수 있는 사용 권한이 없는 경우, 데이터베이스에서 특정 테이블로 액세스하는 권한이 있다는 사실이 문제가 될 수 있습니다. 이 사용자는 데이터베이스에 액세스할 수 없으며 데이터베이스에서 테이블을 사용할 수도 없습니다.

IDS 로그 관리

Informix 데이터베이스는 데이터 복구용으로 논리 로그를 사용합니다. 논리 로그는 메모리에 버퍼가 지정된 다음 디스크 파일에 쓰여 집니다. 논리 로그를 디스크에 남겨 두면, 데이터베이스 시스템에 대한 가장 심각한 하드웨어 오류 중 하나인 디스크 오류가 발생할 때에 복구 기능을 사용할 수 없습니다. 또한, 디스크의 논리 로그 파일은 테이프로 백업될 때까지는 사용할 수 없습니다. 이러한 이유로 디스크의 논리 로그는 가능한 빨리, 되도록이면 연속적으로 테이프에 보관되어야 합니다.

IDS 데이터베이스는 두 가지 논리 로그 백업 방법을 제공합니다. 특정 시간에 관리자가 논리 로그를 백업하도록 선택할 수 있는 정기 백업과 각 로그 파일이 채워지면 디스크로 백업되는 연속 백업입니다.

디스크의 논리 로그가 채워지면 엔진은 새 트랜잭션을 시작할 수 없으며, 데이터베이스 활동은 정지될 수 있습니다. 따라서 데이터베이스 관리자가 데이터베이스에서 논리 로그 활동을 모니터하는 것은 필수입니다. 구체적으로 관리자는 얼마나 많은 논리 로그 파일을 사용할 수 있으며 로그 파일이 얼마나 빠르게 채워지고 있는지를 알고 있어야 합니다. 다음 절에서는 논리 로그 파일의 모니터링 및 보관 프로세스에 대해 설명합니다.

논리 로그 모니터링

논리 로그 파일의 수는 시스템 활동에 따라 다양할 수 있습니다. 논리 로그를 사용하는 과중한 업데이트 활동이 주를 이루므로, 집중적인 읽기 활동의 기간은 많은 논리 로그를 필요로 하지 않습니다. 대형 데이터베이스 테이블의 정기 로드와 같은 광범위한 업데이트 기간에는 로그 파일을 많이 필요로 할 수 있습니다. 관리자는 데이터베이스의 트랜잭션 프로파일을 검토하고 피크 활동을 관리하는 데 필요한 논리 로그 파일의 수를 예측해야 합니다. 논리 로그를 추가하는 프로세스는 간단하지 않기 때문에 로그 파일과 레벨 0 보관의 추가를 요구하는 로그는 미리 추가되어야 하고 피크 시스템 활동을 위해 남겨져 있어야 합니다.

onstat -l 옵션을 사용하여 논리 로그를 모니터할 수 있습니다. 아래에 표시된 대로 이 명령의 출력에는 시스템에 대한 논리 로그와 개별 로그 파일의 현재 상태가 나열되어 있습니다.

```
Physical  Logging
Buffer  bufused   bufsize   numpages   numwrits   pages/io
  P-1   0         16        0          0          0.00
        phybegin  physize   phypos     phyused    %used
        10003f    500       214        0          0.00

Logical  Logging
Buffer  bufused   bufsize  numrecs  numpages  numwrits  recs/pages  pages/io
 L-2   0          16       1        1         1         1.0         1.0

address   number   flags     uniqid    begin       size       used     %used
832bb110  1        U---      1         100233      250        250      100.00
832bb12c  2        U---      2         10032d      250        250      100.00
```

```
832bb148  3        U---      3         100427      250         250      100.00
832bb164  4        U---      4         100521      250         250      100.00
832bb180  5        U---      5         10061b      250         250      100.00
832bb19c  6        U---      6         100715      250         250      100.00
832bb1b8  7        U---      7         10080f      250         250      100.00
832bb1d4  8        U--C-L    8         100909      250         120       48.00
832bb1f0  9        U---      0         100a03      250           0        0.00
832bb20c  10       U---      0         100afd      250           0        0.00
```

출력의 논리 로깅 섹션에는 각 논리 로그 파일의 주소, 로그 파일 번호 및 로그 파일에 대한 상태 플래그가 나열되어 있습니다. 로그 파일에 대한 상태 플래그는 표 6-7에 나열되어 있습니다.

표 6-7 논리 로그 상태 플래그

로그 상태	설명
A	새로 추가되었음
B	백업되었음
C	현재의 논리 로그 파일
F	사용 가능
L	가장 최근의 체크포인트 레코드를 포함
U	사용되었음

논리 로그가 백업을 나타내는 B 또는 사용 가능을 나타내는 F 중 하나인 플래그와 함께 나열되면, 로그 파일은 사용 가능합니다. 다른 플래그가 있는 로그는 백업을 기다리거나 사용할 수 없습니다.

사용 가능한 로그 파일이 필요하면, 사용한 로그 파일을 테이프에 백업해야 할 수도 있습니다. 이렇게 하면 해당 로그 파일을 사용할 수 있게 됩니다. 다음 절에 설명되어 있는 대로 ontape 명령을 사용하면 로그 파일을 백업할 수 있습니다.

로그 파일 백업

논리 로그가 가득 차게 되면 논리 로그 파일의 연속 백업을 사용하여 로그 파일을 테이프에 백업할 수 있습니다. 이 접근 방식은 높은

수준의 오류 복구율을 제공합니다. 치명적인 디스크 오류 발생 시 상실되는 가장 큰 부분은 부분 로그 파일 내에서 처리 중이던 트랜잭션입니다.

연속 로깅의 사용에서는 논리 로그 테이프만을 저장하는 테이프 드라이브가 필요합니다. 데이터베이스 보관을 위하여 테이프를 사용하는 시스템에서, IDS 데이터베이스 지원에 필요한 테이프 드라이브에는 보관용과 연속 논리 로그 백업용의 두 가지가 있습니다.

연속 논리 로그 백업을 시작하기 위한 ontape 명령은 다음과 같습니다.

```
ontape -c
```

ontape가 사용하는 테이프 장치는 LTAPEDEV 구성 매개변수에 의해 정의됩니다. 논리 로깅에 영향을 주는 다른 명령에서와 마찬가지로 사용자는 사용자 informix 또는 root로 로그인해야 합니다.

연속 논리 로그 파일 백업이 가능성이 없는 경우, 완료된 논리 로그 파일을 디스크에 누적한 후 다음 명령을 사용하면 한 번에 모두 백업할 수 있습니다.

```
ontape -a
```

이 명령은 automatic 논리 로그 백업이라는 완료된 모든 논리 로그의 백업을 시작합니다. 연속 논리 로깅에서와 마찬가지로 LTAPEDEV 매개변수로 지정된 장치에 로그 파일을 보관합니다.

물리 로그의 크기 및 위치 변경

물리 로그는 IDS 데이터베이스에 의해 일정하게 사용됩니다. 공유 메모리 버퍼의 페이지가 세션에서 처음으로 변경될 때마다, 페이지의 사전 이미지(업데이트 이전)가 물리 로그에 쓰여집니다. 물리 로그에 대한 일정한 쓰기는 사용 중인 시스템에서 대량의 디스크 입출력을 야기하게 합니다. 이러한 이유로 디스크 입출력이 다른 데이터베이스 디스크 활동에 간섭하지 않는 위치로 물리 로그를 이동하는 것이 바람직합니다.

데이터베이스 인스턴스를 작성하면 물리 로그는 기본적으로 root DBSPACE에 위치합니다. 이 데이터베이스 인스턴스를 작성하면 다른 DBSPACE가 없기 때문에 데이터베이스 인스턴스의 초기화

중에 어쩔 수 없이 물리 로그를 root DBSPACE에 배치해야 합니다. 그러나 일단 추가 DBSPACE를 데이터베이스 인스턴스에 추가하면 다음 방법 중 하나를 사용하여 물리 로그의 위치 및 크기를 변경할 수 있습니다.

- onmonitor 유틸리티
- PHYDBS 또는 PHYSFILE 구성 매개변수의 변경
- onparams 명령

위의 방법 중 한 가지 접근 방법을 사용하여 데이터베이스 인스턴스의 공유 메모리를 다시 초기화해야 합니다. 또한, 데이터베이스의 레벨 0 백업을 작성하는 것도 좋습니다. 이 레벨 0 백업은 데이터베이스가 손상되었을 때 데이터베이스(새로 변경된 물리 로그를 포함하여)를 복구할 수 있습니다. 다음 절에서는 물리 로그의 크기 지정 및 이동 프로세스를 자세하게 설명합니다.

tip *LTAPEDEV가 /dev/null로 설정되어 있는 경우, 논리 로그 파일이 가득 차면(로그를 백업하려는 시도가 없을 때) 논리 로그 파일을 비우게 됩니다. 이러한 작업은 복구 중 모든 논리 로그 롤포워드를 방해합니다. 즉, 복구는 최종 완료 백업의 시점까지만 유효합니다.*

때때로 LTAPEDEV를 /dev/null로 설정하는 것이 유용할 수도 있습니다. 예를 들면, insert 또는 update문을 사용하여(예: 로드되고 있는 상태에서 데이터를 필터링하거나 변환하는 로드 프로그램을 사용하여) 데이터베이스의 많은 부분을 로드하는 경우나 Informix-SQL load문을 사용하고 있는 경우입니다. 이 경우, 데이터베이스를 복구하는 데 로그 롤포워드를 사용할 필요가 없을 수도 있습니다. 대신에 오류가 발생한 로드를 단순히 다시 시작할 수 있습니다. /dev/null의 LTAPEDEV 설정을 사용하면 필요 없는 다수의 논리 로그를 유지보수하는 오버헤드를 간단하게 방지할 수 있습니다. 로드가 완료되면, 엔진을 오프라인 상태로 가져올 수 있고, LTAPEDEV 매개변수를 논리 로그의 테이프 장치로 변경할 수 있으며, 엔진을 재시작할 수 있습니다.

onconfig 파일의 변경

onconfig 파일의 두 매개변수 PHYDBS와 PHYSFILE은 물리 로그를 직접 제어하는 데 사용됩니다. PHYDBS 구성 매개변수를 사용하면 물리 로그가 위치할 DBSPACE를 식별할 수 있습니다. 물리 로그의 기본 위치는 root DBSPACE입니다. 이전에 설명한 대로 물리 로그는, 로그의 중요한 디스크 입출력이 될 수 있는 항목이 시스템의 성능에 영향을 주지 않는 위치에 지정하는 것이 가장 좋습니다. 일반적으로 물리 로그를 단독으로 DBSPACE에 배치하거나, 매우 정적이며 제한된 디스크 입출력을 필요로 하는 테이블이 있는 DBSPACE에 배치하는 것입니다. 이 매개변수에 제공된 값은 유효한 DBSPACE여야 하며 물리 로그에 대하여 적절한 인접 공간이 있어야 합니다. (5장에 IDS 물리 로그의 크기 설정을 위한 팁이 나와 있습니다.)

PHYSFILE 구성 매개변수는 킬로바이트로 물리 로그의 크기를 설정하는 데 사용됩니다. 기본적으로 물리 로그 파일의 크기는 200KB입니다. 이 매개변수의 입력값은 항상 200KB 이상이어야 합니다.

onparams 명령 사용

다음과 같이 onparams 명령을 사용하면 물리 로그의 크기와 위치를 이동시킬 수 있습니다.

```
onparams -P -s <size> -d <dbspace>
```

-P 매개변수는 물리 로그가 변경되고 있다는 것을 나타냅니다. -s 매개변수는 물리 로그의 크기를 표시하고 물리 로그의 크기를 나타내는 부호가 없는 정수가 다음에 옵니다. 이 크기는 200 이상이어야 합니다. -d 매개변수 다음에는 물리 로그 파일을 저장할 DBSPACE의 이름이 옵니다. 이 매개변수를 제공하는 값은 유효한 DBSPACE의 문자열 이름이어야 합니다. 이 DBSPACE 내에는 물리 로그에 적절한 인접 공간이 있어야 합니다.

논리 로그의 추가

논리 로그 파일이 필요한 경우, onparams 명령을 사용하여 추가

할 수 있습니다. 논리 로그 파일을 추가하면 관리자는 논리 로그 파일의 DBSPACE와 크기를 지정해야 합니다. 논리 로그는 사용 시 많은 양의 디스크 입출력을 수행하기 때문에, 추가 입출력 문제가 발생하지 않는 단일 DBSPACE 또는 DBSPACE 세트에 논리 로그 파일을 지정해야 합니다. 지정한 DBSPACE에는 논리 로그에 지정된 인접 공간이 있어야 합니다. 그렇지 않은 경우, 명령이 실패합니다. 논리를 추가하는 데 사용하는 onparams 명령은 다음과 같습니다.

```
onparams -a -d dbspace [-s <size>]
```

논리 로그의 크기가 지정되지 않았을 경우, LOGSIZE 구성 매개변수가 지정한 로그 크기가 사용됩니다. 이 명령은 지정한 DBSPACE에 단일 로그 파일을 추가합니다. 즉, 몇 개의 논리 로그 파일을 추가하려면 명령을 반복적으로 실행해야 합니다. 추가된 로그 파일은 레벨 0 보관을 작성할 때까지 사용할 수 없습니다. onparams 명령을 실행하는 사용자는 informix 또는 root이어야 하며 데이터베이스는 대기(단일 사용자) 모드여야 합니다.

논리 로그 삭제

경우에 따라 논리 로그가 사용한 공간을 비우고 다른 목적으로 할당해야 하는 경우가 있을 수 있습니다. 논리 로그는 onmonitor 유틸리티 또는 onparams 명령 중 한 방법을 사용하여 데이터베이스 인스턴스에서 삭제할 수 있습니다.

onparams 명령에는 일반적인 모든 관리 경고가 적용됩니다. 데이터베이스는 대기 모드여야 하고, 명령을 실행하는 사용자는 informix 또는 root이어야 합니다. 또한 해제 되고 있는 논리 로그는 사용 가능을 의미하는 “F” 상태이거나 새로 추가를 의미하는 “A”여야 합니다. 이 명령에 대한 구문은 다음과 같습니다.

```
onparams -d -l <logid>
```

logid는 onstat -l 명령이 표시한 대로 사용 가능 로그 또는 새로 추가된 논리 로그의 유효한 로그 ID여야 합니다. 논리 로그는 한 번에 하나씩 삭제되어야 합니다. 10개의 논리 로그를 삭제하려면 논리 로그마다 한 번씩 onparams 명령을 10번 실행해야 합니다.

보관 및 복구

Informix에서는 데이터베이스에 데이터를 보관하기 위해 ontape, onarchive 및 onbar의 세 가지 유틸리티를 제공합니다. 이 책에서는 데이터 보관과 복구를 위한 간단하고 직접적인 방법인 ontape 유틸리티에 대해 다룹니다. onbar 유틸리티는 스토리지 관리 장치(테이프 저장소 또는 많은 매체 관리 작업을 자동화하는 기타 백업 장치)와 함께 사용되며 Informix 기술 문서로 작성됩니다. onarchive 유틸리티는 ontape 유틸리티에 몇 가지 추가 기능을 제공하지만 대부분 onbar 유틸리티의 기능이 이를 대신합니다.

보관은 온라인으로 수행할 수 있습니다. 이 경우는 사용자가 레코드를 읽고 쓰며 데이터베이스 데이터에 액세스하고 있을 때, 데이터베이스 백업이 진행 중일 수 있다는 의미입니다. IDS 백업 스레드는 버퍼 페이지의 타임스탬프를 검토함으로써 페이지가 보관에 속하는지 여부를 결정할 수 있습니다.

복구는 전체 복구(cold restore) 또는 선택 복구(warm restore) 중 하나로 수행할 수 있습니다. 전체 복구는 데이터베이스가 완전히 오프라인일 때 수행됩니다. 이 유형은 중요한 DBSPACE의 복구에 필수적입니다. 선택 복구는 데이터베이스가 온라인일 때(일부 기능에 제한 사항이 있음) 수행됩니다.

보관 레벨

IDS 백업 유틸리티는 레벨 0, 레벨 1, 레벨 2의 점증적 보관 레벨을 제공합니다. 다중 보관 레벨을 사용함으로써 점증적으로 데이터를 보관하여 테이프에 써야 하는 데이터의 양을 줄일 수 있습니다.

레벨 0 보관은 모든 데이터를 데이터베이스에 보관할 때 사용합니다. 페이지의 타임스탬프와는 상관없이 모든 데이터 페이지를 보관합니다.

레벨 1 보관은 최종 레벨 0 보관 이후 수정된 모든 데이터 페이지를 보관할 때 사용합니다. ontape 보관 스레드는 데이터베이스 인스턴스의 각 페이지의 datetime 스탬프를 검토하고 최종 레벨 0 보관(시스템 예약 페이지에 기록된 대로) 이후 수정된 페이지만 선택합니다.

레벨 2 보관은 최종 레벨 1 보관 이후에 수정된 모든 데이터 페이지를 보관할 때 사용합니다. ontape 보관 스레드는 데이터베이스 인스턴스의 모든 데이터 페이지에서 datetime 스탬프를 검토하고 최종 레벨 1 보관 이후 수정된 백업 페이지를 선택합니다.

보관 수행

ontape 유틸리티는 IDS 보관에 필수적인 기본 기능을 제공하는 것으로, Informix 엔진에서 절대적으로 가장 많은 작업을 하는 유틸리티입니다. ontape를 사용함으로써 관리자는 전체적인 데이터베이스에 대해 레벨 0, 레벨 1, 레벨 2를 작성할 수 있습니다. (복구용으로 DBSPACE 세트를 식별할 수 있는 경우에도 ontape는 단일 DBSPACE 또는 DBSPACE의 세트 보관을 지원하지 않습니다.)

보관을 위한 ontape 구문은 다음과 같습니다.

```
ontape -s -L <archive level>
```

-s 매개변수는 보관이 수행 중임을 나타내기 위해 ontape로 전달됩니다. -L 매개변수 다음에는 수행 중인 보관 레벨을 표시하는 값이 옵니다. 유효한 값은 물론 0, 1, 2입니다.

ontape 프로그램은 어떤 예측을 기준으로 설계되었습니다. 우선, 보관이 여러 테이프를 처리할 수 있기 때문에 운영자는 테이프를 변경해야 한다는 것입니다. 현재 백업 매체의 테이프 스토리지 용량을 기준으로 항상 성립할 수 있는 것은 아니지만, 이러한 예측의 결과는 ontape 유틸리티가 배치 프로세스에서 실행되도록 설계되지 않았다는 것입니다. ontape 프로그램은 프롬프트와 메시지를 Unix stdin 및 stdout 장치에 읽고 씁니다. 이는 ontape가 프로그램을 시작한 터미널 키보드로부터 입력(stdin)을 읽고 터미널 화면으로 출력(stdout)한다는 것을 의미합니다.

여러 관리자가 시도하는 작업은 의도하지 않은 보관, 즉 시간 외에 실행되고 사용자가 개입할 필요가 없는 보관을 생성하는 것입니다. 안타깝게도 ontape는 이러한 목적을 염두에 두고 만들어지지 않았습니다. Unix 배치 파일 내에서 stdin과 stdout을 재지정하고 ontape를 실행할 수는 있지만, 작업을 다루기 어려울 수 있습니다. 보관과 그 의미의 중요성을 고려할 때 이러한 작업은 시도하지 않는 것이 좋습니다.

onarchive와 onbar 유틸리티를 사용하여 의도하지 않은 보관을 생성할 수 있습니다. onbar 유틸리티는 추가 보관 기능을 제공하는 스토리지 관리 장치와의 상호 작용을 허용합니다. 의도하지 않은 보관을 필요로 하는 사용자는 이 유틸리티에 대한 문서를 미리 살펴보아야 합니다.

데이터 복구

불행히도 하드웨어 또는 소프트웨어 오류로 인해 보관에서 데이터베이스를 복구해야만 할 때가 올 수 있습니다. 전체 복구는 적어도 레벨 0 보관이 필요합니다. 레벨 0 보관 이후 레벨 1 또는 레벨 2 보관을 수행했다면 복구에도 사용되어야 합니다. 그리고 마지막으로, 최종 보관(레벨 0, 1, 2) 이후 테이프에 작성한 논리 로그를 사용할 수 있어야 합니다. 모든 테이프가 준비되면 복구 프로세스를 시작할 수 있습니다.

이상적으로는 레벨 0, 레벨 1, 레벨 2 보관을 조합하면 데이터베이스를 전날 밤의 보관 위치로 가져옵니다. 그런 후 논리 로그를 사용하여 데이터베이스를 최종 체크포인트 위치까지 롤포워드합니다. 이 복구 프로세스를 완료하면 최종 체크포인트 이후에 데이터베이스에 작성된 데이터만이 상실되고, 다른 모든 데이터는 복구됩니다. 복구를 수행하는 ontape 옵션은 다음과 같습니다.

```
ontape -r [-D <dbspace list for restore>]
```

전체 데이터베이스 복구를 수행하기 위해 첫 번째 테이프(레벨 0 보관의 시작)를 보관 테이프 장치(TAPEDEV 구성 매개변수)로 지정된 테이프 드라이브에 배치한 후, 오프라인의 IDS 엔진에서 다음 ontape 명령을 실행합니다.

```
ontape -r
```

단일 DBSPACE를 복구하려면, 다음 명령을 실행하십시오.

```
ontape -r -D dbspace1
```

이 명령을 실행하려면 테이프 드라이브에서 레벨 0 보관이 사용 가능해야 합니다.

ontape 유틸리티는 추가 테이프가 필요하다고 감지한 경우, 관리자에게 보관 레벨마다 추가 테이프에 대한 프롬프트를 표시합니다. 일단 레벨 0 보관을 복구했으면, 모든 추가 보관 레벨을 요청하고 사용 가능한 경우 테이프에 대한 프롬프트를 표시합니다. 일단 보관 테이프를 복구했으면 ontape 유틸리티는 로그 롤포워드 작업에 사용할 수 있는 모든 논리 테이프에 대한 프롬프트를 표시합니다. 논리 로그 포워드는 DBSPACE 레벨 복구를 수행하는 것이나, 데이터베이스의 더 완전한 복구를 분명히 제공하더라도 다른 복구에는 필수가 아니라는 점에 유의하십시오.

전체 복구 및 선택 복구

전체 데이터베이스를 복구해야 하거나, root DBSPACE와 같은 일부 중요한 데이터베이스 부분을 복구해야 하는 경우, 전체 복구(cold restore)를 수행해야 합니다. 전체 복구는 복구 작업 중 데이터베이스가 오프라인임을 의미합니다. (물론 root DBSPACE가 손상된 경우 데이터베이스는 어떤 경우에도 시작되지 않습니다.)

전체 복구는 물리적 복구(레벨 0 및 다른 보관의 복구)와 논리적 복구(로그 테이프를 사용한 논리 로그 롤포워드의 수행)로 구성됩니다.

복구가 시작되면 IDS 엔진은 오프라인이 됩니다. 하지만 시스템 예약 페이지를 복구하면 복구 모드로 들어가게 됩니다. 그런 후 논리 복구가 완료되거나 onmode를 사용하여 다른 모드로 들어가게 할 때까지 복구 모드로 남게 됩니다.

선택 복구(warm restore)를 사용하여 중요하지 않은 DBSPACE를 복구할 수 있습니다. 선택 복구를 사용하면 데이터베이스는 온라인 또는 대기 모드상태에 있습니다. 데이터베이스가 온라인 모드에 있는 경우, 데이터베이스 쿼리는 복구 중인 DBSPACE에 있는 데이터에 액세스할 필요가 없는 한 성공합니다.

데이터베이스 로깅 상태의 변경

Informix IDS 엔진은 다른 관계형 데이터베이스와는 달리 데이터베이스 로깅을 필요로 하지 않습니다. 즉, 데이터베이스 내에서의 트랜잭션 사용을 방해하지는 않으면서, 로깅하지 않고 IDS 데이터베이스에 대해 작업할 수 있습니다.

데이터베이스 로깅은 “with log”절을 사용하여 데이터베이스 작성 명령문에서 지정됩니다. 이 “with log”절은 논리 로깅을 사용하여

데이터베이스를 작성하도록 지정할 때 사용됩니다. 데이터베이스가 일단 실행되면, ontape 명령을 사용하여 데이터베이스의 로깅 상태를 변경할 수 있습니다. 데이터베이스에 로깅을 추가하려면 레벨 0 보관을 수행해야 합니다. 이는 데이터베이스에서 복구를 수행해야 할 때 필요합니다. 표 6-8의 나열된 ontape 명령을 사용하여 데이터베이스의 로깅 상태를 변경할 수 있습니다.

표 6-8 ontape를 사용하여 로깅 상태 변경

매개변수	설명
-B	버퍼가 지정된 로깅을 데이터베이스에 추가
-N	지정한 데이터베이스에 대해 로깅 종료
-U	로깅 상태를 버퍼가 지정되지 않은 로깅으로 변경
-A	로깅 상태를 ANSI 준수 로깅으로 변경

위의 명령문을 다음과 같이 실행할 수 있습니다.

```
ontape -s [-A | -B | -N | -U | -A <database name>]
```

sales_db의 상태를 버퍼가 지정된 로깅에서 버퍼가 지정되지 않은 로깅으로 변경하려면 다음과 같이 명령을 실행할 수 있습니다.

```
ontape -s -U sales_db
```

sales_db의 상태를 버퍼가 지정된 로깅으로 변경하려면 다음과 같이 명령을 실행할 수 있습니다.

```
ontape -s -B sales_db
```

요약

이 장에서는 엔진의 시작과 중지와 같은 기본적인 관리 옵션에서부터 IDS 데이터베이스의 보관 및 복구 프로세스에 이르기까지 IDS 엔진의 기본적인 관리 기능에 대해 다루었습니다. 사용자 역할과 테

이블 레벨의 사용 권한에 대한 IDS 구현을 상세히 설명하며 데이터베이스 보안의 측면에 대해서도 다루었습니다.

IDS onmode 명령을 사용하여, 엔진이 온라인 상태일 때 엔진이 환경의 요구에 맞게 동적으로 변경할 수 있도록 하는 강력한 기능을 특정 구성 매개변수에 설정하도록 할 수 있습니다. 이 장에서는 onmode 명령과 효과에 대하여 다루어 보았습니다.

논리 로깅 및 물리 로깅은 IDS 엔진의 데이터 무결성 기능에 대한 중요한 구성 요소입니다. 논리 로깅은 데이터베이스의 물리적 복구를 수행한 후에 데이터베이스에 대한 중요한 오류 복구 메커니즘을 제공합니다. 따라서 논리 로그는 연속 논리 로깅을 사용하여 테이프로 작성하는 것이 중요합니다. 이 장에서는 논리 로그 백업 및 다른 로깅 관리를 설정하는 프로세스에 대해 다루었습니다.

다음 장 미리보기

다음 장에서는 구조화된 쿼리 언어(SQL)의 Informix 버전을 다룹니다. Informix-SQL은 관리자와 개발자가 IDS 데이터베이스와 상호 작용하는 방법입니다. SQL을 적절하게 사용하는 것은 관리자에게 필요한 관리 작업을 수행할 기능을 제공하면서 데이터베이스를 유지보수하는 프로세스의 중요한 부분입니다. 응용프로그램 개발자는 IDS 엔진에 대해 효율적이고 생산적인 응용프로그램을 적절하게 프로그래밍하기 위해 Informix의 SQL을 완전하게 이해해야 합니다. 다음 장에서는 개발자와 관리자 모두에게 유용한 정보를 제공하며 중요한 Informix-SQL문을 자세하게 다루고 이 사용을 위한 실질적인 팁과 제안사항을 제시합니다.

7장

관계형 데이터베이스와 Informix-SQL

관계형 데이터베이스는 성능이 낮은 Unix 및 NT 플랫폼에서부터 고성능 Unix 서버에 이르기까지 데이터베이스 서버 시장을 지배하는 데이터베이스 중에서 최상의 데이터베이스로 여겨지고 있습니다. 이렇게 인기 있는 데이터 스토리지 매체의 역사를 이해하면 관계형 데이터베이스 이론의 특성에 대해 보다 명확한 시각을 가질 수 있습니다.

관계형 데이터베이스의 성공에 기여한 것으로는 관계형 데이터베이스 데이터의 액세스 언어로 널리 보급된 구조화된 쿼리 언어(SQL)의 사용을 꼽을 수 있습니다. SQL은 사실상 모든 데이터베이스 공급자가 지지하는 주요한 표준이 되어 왔기 때문에 관계형 데이터베이스는 데이터 스토리지 플랫폼에 많은 사용자들에게 익숙한 인터페이스를 제공할 수 있었습니다. SQL을 사용하여 Sybase 데이터베이스에 액세스하는 프로그래머는 동일한 기본 기술을 사용하여 Informix 데이터베이스에 액세스할 수 있도록 프로그램을 작성할 수 있습니다.

이 장에서는 관계형 데이터베이스의 기본사항을 설명하며 기본 용어와 개념을 소개합니다. 각 데이터베이스 공급자는 SQL 버전에 몇 가지 확장 기능을 구현했으며 Informix도 예외는 아닙니다. SQL은 비프로시저 언어이므로 SQL만을 통해 표현할 수 있는 논리에는 제한사항이 있습니다. 이러한 이유로 인해 데이터베이스 공급자들은 데이터베이스 입출력을 수행할 때 프로시저 논리를 표현하기 위해 내장 프로시저 언어(SPL)를 작성했습니다. 이 언어도 이 장에서 다룰 것입니다.

관계형 데이터베이스의 역사

관계형 데이터베이스 모델은 이론적으로 1960년대 후반에 시작되었습니다. 1968년, IBM 연구 기관의 E. F. Codd 박사가 데이터베이스 관리 시스템의 세계에 수학적인 엄밀성을 적용하는 개념을 연구하기 시작했습니다. 이후에 Codd의 개념은 획기적인 논문이라 할 수 있는 "A Relational Model of Data for Large Shared Data Banks"(Communications of the ACM, 13권, 6번, 1970년 6월)로 발표되었습니다. 이 논문에서 펼친 개념은 그 뒤로 데이터베이스 시스템의 성격에 대단한 영향을 미쳤으며, 오늘날에는 모든 관계형 데이터베이스 시스템의 이론적인 표준이 되었습니다.

관계형 데이터베이스는 증명할 수 있는 수학적 기초를 근거로 하기 때문에 학회 단체에서 선호되고 있습니다. 이러한 점으로 인해 관계형 데이터베이스가 다른 데이터베이스 형식과 구별되고 쿼리 언어를 개발하는 과정이 보다 쉽게 바뀌었습니다.

관계형 데이터베이스 개념

관계형 데이터베이스는 개념적으로 테이블의 컬렉션입니다. 데이터베이스의 각 테이블은 데이터 엔티티를 나타내며 각 엔티티는 데이터 속성의 컬렉션입니다. 엔티티는 관련 정보가 기록될, 구별이 가능한 객체입니다. 이러한 객체의 예로 자동차를 들 수 있습니다. 속성은 구별할 수 있는 객체와 연관된 특성 또는 정보입니다. 자동차 엔티티의 속성에 대한 예로는 자동차의 색상, 자동차의 구조, 엔진 크기 및 자동차의 수명을 들 수 있습니다.

엔티티, 속성 및 관계

관계형 데이터베이스를 설계하는 과정에서 먼저 모델로 만들 엔티티(객체)를 식별해야 합니다. 예를 들어, 카탈로그 주문을 받는 시스템 설계에서 엔티티는 카탈로그 품목, 한 품목 또는 여러 품목에 대한 주문, 품목의 제조업체 및 품목을 구입한 고객과 같은 객체가 됩니다.

다음 단계는 속성을 식별하는 것입니다. 엔티티의 속성은 엔티티나 객체의 특성 또는 특징입니다. 카탈로그 품목의 경우 특성은 품목의 설명, 품목의 가격, 품목의 무게, 품목의 크기 및 품목의 제조업체가 됩니다. 고객과 같이 기타 엔티티의 특성은 고객의 이름 및 우편 번호를 포함한 고객의 주소가 됩니다. 그리고 주문에는 주문을 작성한 고객, 주문하는 품목 및 주문 비용에 대한 특성이나 속성이 포함됩니다.

엔티티는 관계를 가질 수 있습니다. 관계는 모델로 만드는 객체들 간의 연결입니다. 주문 자체만 존재할 수는 없습니다. 주문을 작성한 고객이 있으며 고객이 구입한 품목을 나타내는 주문 상의 품목이 있습니다. 따라서 고객 및 품목 엔티티는 주문 엔티티와 관련되어 있습니다.

관계에는 여러 가지 형식이 있습니다. 일대일, 일대다 및 다대다 관계가 있습니다. 일대일 관계는 특정 엔티티에 대해 관련된 엔티티가 하나만 있음을 나타냅니다. 일대일 관계의 예로는 제조업체와 품목의 관계가 있습니다. 특정 품목 레코드에 대해 하나의 제조업체 레코드만 들어 있습니다. (한 품목에 대해 여러 제조업체를 허용하는 설계도 있을 수 있지만 여기에서는 다른 제조업체가 다른 품목을 수반하는 것으로 간주합니다.)

일대다 관계는 주어진 엔티티 레코드에 대해 하나 이상의 관련된 엔티티 레코드가 있음을 나타냅니다. 이 경우는 주문에 주문 엔티티와 라인 품목이 있는 경우입니다. 각각의 모든 주문 레코드에 대해 구입한 품목이 여러 개이고 각 품목을 라인 품목 레코드로 나타냅니다.

다대다 관계는 주어진 엔티티의 여러 레코드에 대해 하나 이상의 레코드가 관련된 엔티티에 있음을 나타냅니다. 다대다 관계의 예로는 자동차와 가족 구성원 간의 관계를 들 수 있습니다. 한 가족 구성원이 둘 이상의 자동차를 운행할 수 있고 하나의 자동차를 둘 이상의 가족 구성원이 운행할 수 있습니다.

관계형 데이터베이스는 종종 엔티티 관계 다이어그램(ERD)을 사용하여 다이어그램으로 표시됩니다. 이러한 다이어그램은 일련의 특수화된 선, 상자 및 기호를 제공하여 엔티티들 간의 관계를 나타냅니다.

정규화

데이터베이스의 속성과 엔티티가 식별되었으면 데이터베이스 개발자가 완료해야 하는 정규화 프로세스가 있습니다. 일반적으로 정규화 프로세스에서는 엔티티에서 반복되는 속성을 제거하고 특정

엔티티에 속한 속성과 다른 엔티티에 속한 속성을 식별해야 합니다. 데이터 분석가는 적절한 정규화 수준에 도달할 때까지 보통 여러 번 반복되는 검토 및 수정 과정을 거치게 됩니다.

여러 정규화 수준은 Codd 및 다른 학자들에 의해 정의되었습니다. 각 수준은 엔티티와 해당 속성을 조사 및 검토하고 해당 엔티티에 속하는 것과 속하지 않는 것을 판별합니다. 다섯 가지 정규형이 정의되었지만, 가장 자주 사용되는 정규화 수준은 제 3 정규형입니다.

관계형 데이터베이스의 정규화는 데이터베이스 설계 프로세스에서 중요한 부분입니다. 정규화는 데이터 중복을 줄이므로 데이터에 필요한 데이터 스토리지 양이 줄어듭니다. 정규화된 설계는 대부분의 데이터베이스에 대해 필요한 불가피한 수정 프로세스도 간단하게 만듭니다.

IDS의 구조화된 쿼리 언어

Informix는 ANSI 표준 SQL과 일부 확장된 버전을 모두 지원하는 SQL 버전을 제공하여 분할화와 같은 IDS 엔진의 고유 기능을 지원합니다. 이 언어는 IDS 데이터베이스와 상호 작용하는 데 필요한 모든 기능을 제공합니다.

이 책의 이 SQL 부분에서는 Informix-SQL문 전체 세트를 다루지 않습니다. 많은 명령문이 비교적 분명하지 않으며 거의 사용되지도 않습니다. 이 책은 데이터베이스 관리자와 개발자에게 가장 유용한 명령문에 초점을 맞추어 작성되었습니다.

SQL문 유형

Informix-SQL문은 데이터 정의 언어(DDL)와 데이터 처리 언어(DML) 명령문의 두 가지 유형으로 나뉩니다. 이름이 나타내듯이, DML문은 데이터베이스에 있는 데이터의 처리 및 유지보수와 관련되고 DDL문은 데이터에 대한 스토리지의 정의와 관련됩니다. 이들 명령문은 표 7-1과 7-2에 요약되어 있습니다.

표 7-1 SQL DDL문

DDL문	설명
create database	비어 있는 IDS 데이터베이스를 작성합니다.
create table	비어 있는 테이블을 작성합니다. 컬럼, 자료형을 지정하고 선택적으로 제한 조건과 기본값을 지정합니다.
create view	기존 테이블에 기초하여 뷰를 작성합니다.
drop table	기존 테이블을 삭제합니다.
drop view	기존 뷰를 삭제합니다.
drop database	기존의 비어 있는 데이터베이스를 삭제합니다.
alter table	기존 테이블을 변경합니다. 컬럼 변경, 컬럼 추가, 컬럼 삭제 및 기타 기능 수행에 사용할 수 있습니다.

표 7-2 SQL DML문

명령문	설명
select	지정된 데이터베이스 테이블에서 행을 검색합니다. 둘 이상의 관련된 테이블을 조인하는 데 사용합니다.
update	데이터베이스에서 기존 행을 업데이트합니다.
insert	데이터베이스에 새 행을 삽입합니다.
delete	데이터베이스에서 기존 행을 삭제합니다.

확장 SQL문

Informix는 많은 확장 SQL문을 제공합니다. 이들 명령문은 데이터베이스에 대해 특수한 기능을 수행하며 표 7-3에 요약되어 있습니다.

표 7-3 확장 SQL문

명령문	설명
load	ASCII 파일에서 행을 로드하며 그 내용을 데이터베이스 테이블에 삽입합니다. ASCII 파일은 구분되어야 합니다.
unload	데이터베이스 테이블에서 구분된 ASCII 파일로 행을 언로드합니다.
update statistics	데이터베이스에서 옵티마이저 통계를 업데이트합니다.

데이터 스토리지 작성 - DDL문

IDS 엔진은 데이터베이스 및 이를 구성하는 테이블에 대한 데이터 스토리지 요구사항을 정의하는 DDL문을 지원합니다. 이들 명령문은 다음 절에서 설명합니다.

데이터베이스 작성

create database문은 데이터베이스에 대한 데이터 스토리지를 지정합니다. 이 명령문의 구문은 다음과 같습니다.

```
create database [in <dbspace>] [with log];
```

관리자는 데이터베이스에 대한 DBSPACE를 지정하여 데이터베이스에 대한 시스템 카탈로그의 위치를 지정할 수 있습니다. 추가로, 이 DBSPACE는 데이터베이스에 작성된 테이블의 기본 위치가 됩니다. 관리자가 이 기능을 사용하면 데이터베이스와 테이블의 위치를 어느 정도 구성하고 제어할 수 있습니다.

"with log"절은 데이터베이스 로깅을 지정할 수 있게 합니다. Informix를 사용하면 로깅 없이 데이터베이스를 작성할 수 있습니다. 이러한 데이터베이스는 로깅이 제공하는 복구 메커니즘을 제공하지 않으므로, 로깅 없이 작성된 데이터베이스에서는 트랜잭션 제어를 제공하는 SQL문(시작 작업, 완료 작업)을 실행할 수 없습니다.

"no log" 데이터베이스는 로깅을 완전히 사용하지 않는 것은 아닙니다. 논리 및 물리 로그가 데이터베이스에 대해 여전히 작성됩니다. 논리 로깅은 DDL문에 계속 사용되며 각 SQL문은 "단일 트랜잭션"으로 취급됩니다. 이는 둘 이상의 명령문 세트를 롤백할 수 없지만, 단일 명령문의 실패는 데이터 무결성을 보존하기 위해 엔진에 의해 롤백될 수 있다는 것을 의미합니다. "no log" 데이터베이스에서 실행되는 특정 명령문에 대해 상당수의 논리 로그가 사용될 수 있음을 나타냅니다. 예를 들어, ASCII 파일에서 데이터베이스로 수많은 레코드를 로드하기 위해 다음에 설명될 load문을 사용할 수 있습니다. 이 전체 명령문은 로그되어지며 잠재적으로 많은 논리 로그 레코드 수를 완료하도록 요구할 수 있습니다.

일반적으로 권장되는 것은 아니지만 “no log” 데이터베이스는 몇 가지 경우에 유용할 수 있습니다. 예를 들어, 정보의 하루 스냅샷이나 주간 스냅샷을 나타내며 정기적으로만 업데이트되는 데이터가 포함된 데이터베이스는, 로깅을 사용하지 않고 효과적으로 실행될 수 있습니다. 이 데이터베이스 유형을 사용하면 매일 또는 매주 실행되는 정기적인 로드를 실패한 경우에 다시 실행하도록 설계할 수 있으므로 데이터베이스 내에서 트랜잭션을 실행하는 기능이 필요하지 않습니다. 그리고 복잡한 온라인 업데이트가 발생하지 않기 때문에 온라인 트랜잭션이 일련의 업데이트에 대한 논리적인 롤백을 제공할 필요가 없습니다. 데이터 웨어하우스는 이러한 데이터베이스 유형의 확실한 예제라 할 수 있습니다.

create table 명령문

create table문은 데이터베이스 테이블에 대한 데이터 스토리지를 정의하는 데 사용됩니다. 이 명령문은 테이블 이름과 컬럼의 기본 정의, 컬럼의 자료형과 길이를 제공합니다. 또한 테이블에 대한 참조 및 무결성 제한 조건과 분할화 스키마의 정의를 가능하게 합니다. 다음은 이 명령문의 구문입니다.

```
create table <table name> ( <column name> <column data
type>| <data length> [ constraint definition ] [default
values], … )
[in <dbspace name> ] |
[fragmentation definition]
```

테이블 이름, 컬럼 이름 및 자료형

create table문의 일부분에 구문 오류가 있으면 전체 테이블 빌드 작업이 실패합니다. 테이블 작성 명령문은 테이블의 테이블 이름을 지정하는 데 사용됩니다. IDS를 사용하는 경우 테이블 이름은 18자로 제한됩니다. 테이블 이름은 테이블이 작성된 데이터베이스 내에서 고유해야 하며 데이터베이스 인스턴스 내에서는 고유할 필요가 없습니다.

테이블의 컬럼 이름은 테이블 이름과 동일한 이름을 지정해야 하는 제한사항을 따릅니다. 길이는 18자 이하가 되어야 합니다. 컬럼

이름은 테이블에 대해 고유 컬럼 이름이어야 하지만 데이터베이스에 대해 고유한 컬럼 이름일 필요는 없습니다(즉, 다른 테이블에 동일한 컬럼 이름이 있을 수 있지만 동일한 테이블 내에서는 동일한 이름을 가진 다른 컬럼이 존재할 수 없음).

컬럼의 자료형은 표 7-4에 나열된 자료형 중 하나이어야 하며 식별된 크기 제한사항을 따릅니다.

표 7-4 유효한 자료형

자료형	길이	최소값	최대값
smallint	2	-32,767	+ 32,767
integer, int	4	-2,147,483,647	+ 2,147,483,647
smallfloat, real	4	플랫폼에 따름	플랫폼에 따름
float, double precision	8	플랫폼에 따름	플랫폼에 따름
decimal, dec, numeric	정밀도/2 + 1	-32 유효 숫자	+ 32 유효 숫자
money	정밀도/2 + 1	-32 유효 숫자	+ 32 유효 숫자
date	4	1/1/0001	12/31/9999
char, character	프로그래머가 정의함	1byte	32,767byte

Informix의 `insert` 및 `update` 명령문은 보통 적합한 경우에 자료형 변환을 수행한다는 점을 유의하십시오. 따라서 컬럼을 문자 컬럼으로 작성한 다음 이 컬럼에 정수를 삽입할 수 있습니다. 마찬가지로 변환된 정수 값이 Informix 정수 제한을 초과하지 않은 경우 정수 컬럼은 정수의 문자열 입력을 승인합니다. 컬럼 내용을 엄격히 제어해야 하는 경우 컬럼에 제한 조건을 지정해야 하며, 이에 대해서는 다음 절에서 자세히 설명합니다.

`create table`문의 DBSPACE 이름절은 특정 DBSPACE에서 테이블을 작성할 수 있게 합니다. 기본적으로 DBSPACE절이나 분할화 절에 DBSPACE가 지정되지 않은 경우 테이블은 이를 포함하고 있는 데이터베이스가 작성된 DBSPACE에 배치됩니다.

IDS 분할화 정의

테이블에 대한 분할화 정의는 해당 테이블의 데이터에 대한 분할화 스키마를 정의합니다. 분할화는 표현식을 사용하거나 라운드 로빈

스키마를 사용하여 이루어질 수 있습니다.

라운드 로빈 분할화 스키마는 DBSPACE 목록에 지정된 DBSPACE들에 데이터가 분산되도록 지정합니다. 데이터는 먼저 목록의 첫 번째 DBSPACE에 기록된 다음 목록의 두 번째 DBSPACE에 기록되는 것과 같이 계속 순차적인 형식으로 목록의 DBSPACE에 분산됩니다. 다음은 create table문의 "fragment by"절에 대한 구문입니다.

```
… fragment by round robin in <dbspace list> |
            expression <expression1> in <dbspace name>,
            <expression2> in <dbspace name> …;
```

라운드 로빈 분할화 스키마는 여러 디스크에 데이터를 고르게 분산하는 것이 바람직하거나 표현식 기준 분할화 스키마를 간단하게 개발할 수 없는 경우에 유용합니다. 다음은 라운드 로빈 분할화의 예입니다.

```
create table sales (
            sales_id integer,
            region char(10),
            product char(10),
            sales_amt decimal(12,2))
        fragment by round robin in dbspace1, dbspace2,
dbspace3;
```

이 예에서 sales 테이블의 데이터는 dbspace1, dbspace2 및 dbspace3의 세 DBSPACE에 분할됩니다. 이러한 DBSPACE를 분할이라고 합니다. 이 분할화 스키마는 먼저 dbspace1에 레코드를 기록한 다음 dbspace2와 dbspace3에 레코드를 기록하여 세 개의 DBSPACE에 고르게 데이터를 분산시킵니다. dbspace3에 기록한 후에는 DBSPACE 목록에서 다시 시작하여 dbspace1에 다음 레코드를 기록합니다.

이 테이블에 대한 액세스가 분리된 인덱스(테이블과 동일한 DBSPACE에 있지 않음)를 통해 이루어지는 경우, 데이터 액세스 작업을 통하여 인덱스를 읽고 데이터 레코드가 있는 페이지의 주소를 가져오므로 세 개의 분할에 대해 데이터 분배를 고르게 하면 데이터 액세스에 거의 영향을 미치지 않습니다. 인덱스의 이러한 데이터 레코드 정보는 데이터 레코드가 있는 분할, 페이지 및 페이지 오프셋을

나타내므로 레코드를 찾기 위해 다른 분할의 데이터를 모두 읽을 필요가 없습니다.

그러나 테이블에 대해 테이블을 전부 스캔해야 하는 쿼리를 실행하는 경우 일부 테이블 분할의 스캔을 제거하는 것이 유용할 수 있습니다. 이러한 작업은 표현식 기준 분할화를 사용하면 가능합니다. 표현식 기준 분할화는 표현식을 사용하여 데이터 행이 기록될 위치를 결정합니다. 이 분할화 스키마의 구문은 다음과 같습니다.

```
… fragment by expression
        <expression1> in <dbspace name>,
        <expression2> in <dbspace name> …;
```

이 구문과 함께 일련의 표현식을 사용하여 특정 DBSPACE에 데이터를 기록할 수 있습니다. 이 작업을 수행하고 update statistics 명령을 실행하여 테이블의 옵티마이저 정보를 업데이트하면 IDS 옵티마이저가 분할에 있는 데이터의 특성을 파악합니다. 이 정보가 제공되면 옵티마이저는 테이블에서 데이터 행을 검색하기 위해 스캔할 분할에 대해 지능적인 결정을 할 수 있습니다. 다음은 표현식을 사용한 분할화의 예입니다.

```
create table sales (
        sales_id integer,
        region char(10),
        product char(10),
        sales_amt decimal(12,2))
fragment by expression
    region in ("NW", "NE") in dbspace1,
    region in ("SW", "SE") in dbspace2,
    region in ("MW", "HW") in dbspace3;
```

이 create table문은 판매 레코드의 region 컬럼을 기준으로 하여 여러 DBSPACE에 판매 데이터 레코드를 분할합니다. 이 전체 테이블을 스캔해야 하는 쿼리는 데이터를 지역별로 필터하는 필터 기준을 사용함으로써 옵티마이저가 검색 경로에서 분할을 지능적으로 제거할 수 있게 하여 옵티마이저에 의한 분할 제거의 이점을 가질 수 있습니다.

이전 절에서 언급한 바와 같이 Informix create table문을 사용하면 데이터 스토리지에 대한 여러 특성을 정의할 수 있습니다. 데이

터 테이블, 테이블 이름, 컬럼 이름 및 자료형에 대한 기본 정보와 함께 데이터의 제한 조건 및 분할화를 통한 데이터 분배 방법에 대한 정보도 정의합니다. 전체적으로 create table문은 데이터베이스의 데이터에 효율적으로 액세스하기 위한 기초 작업을 담당한다고 말할 수 있습니다.

데이터의 분할화는 효율적인 데이터 액세스 제공의 결정적인 부분이 될 수 있습니다. 이것은 IDS 데이터베이스를 다른 관계형 데이터베이스 공급자들이 제공하는 것과 구별되게 하는 중요한 기능 중 하나입니다. 성능 튜닝과 관련하여 이 중요한 기능에 관한 자세한 정보는 5장을 참고하십시오.

데이터베이스 뷰 작성

데이터베이스 뷰는 여러 가지 목적으로 사용할 수 있습니다. 테이블과 이들 테이블에 대한 필터 기준 간의 관계에 대한 논리적인 뷰를 표현하는 데 사용할 수 있습니다. 뷰는 하나의 데이터베이스 테이블이나 복잡하게 조인된 여러 테이블을 참조할 수 있습니다. 작성된 뷰는 데이터베이스 테이블이 약간의 제한사항과 함께 액세스되는 것처럼 액세스됩니다. 뷰에 다음 중 하나가 포함되어 있지 않으면 이를 SQL update문에 사용할 수 있습니다.

- 업데이트 가능한 뷰에 대한 뷰 제한사항
- 집계 값이 있는 select 목록
- unique 또는 distinct 키워드 사용
- Group by절
- 산술 표현식을 사용하여 작성된 컬럼에 대해 파생된 값

데이터베이스 뷰는 관계형 데이터베이스의 복잡성을 숨기고 캡슐에 넣는 뛰어난 도구입니다. 복잡한 조인과 필터를 뷰에 표현한 다음, 보다 간단한 방식으로 데이터베이스 테이블에 액세스할 수 있습니다. 테이블 컬럼의 실제 이름은 as절을 사용하여 이름을 바꿔 개발자나 사용자에게 보다 친숙한 이름으로 단순화시키거나 숨길 수 있습니다. create view문의 구문은 다음과 같습니다.

```
create view <view name> [ ( <column name list> )
    as < select statement >
```

뷰를 사용하여 업데이트를 수행할 때 선택적인 with check option절을 사용하여 일련의 규칙을 시행할 수 있습니다. 예를 들어, 다음의 뷰는 지역 "NW" 내의 판매로 업데이트를 제한합니다.

```
create view sales_nw as
    select *
    from sales
    where region = "NW"
    with check option;
```

SQL update문은 뷰 이름을 테이블 이름으로 사용할 수 있지만 지역 컬럼의 값이 "NW"인 행만 업데이트하도록 제한됩니다. 마찬가지로 insert문은 테이블에 행을 삽입할 수 있지만 지역 컬럼의 값이 "NW"인 행만 삽입할 수 있습니다.

뷰는 데이터베이스에 일반적이지만 복잡한 쿼리를 저장하는 데 사용할 수 있습니다. 그러면 프로그래머와 데이터베이스 관리자가 이 쿼리에 "공유 코드"로 액세스할 수 있습니다. 이러한 방법은 개발자들 사이에서 쿼리를 표준화하고 공유할 수 있게 하므로 조직의 비즈니스 규칙을 중앙 위치에 저장하는 수단이 됩니다.

뷰를 사용하여 보안 시행

종종 데이터베이스 보안은 특정 사용자를 테이블에 있는 일련의 행에 대한 뷰로 제한하면서 행 수준에서 표현되어야 합니다. 예를 들어, 특정 판매 지역의 사용자들은 해당 지역의 데이터 행만 볼 수 있도록 허용되어야 합니다. Informix 데이터베이스 보안은 테이블 및 컬럼 보안을 제공하나 행 수준의 보안은 제공하지 않습니다. 그러나 Informix 키워드와 결합된 뷰는 행 수준의 보안을 제공할 수 있습니다. 다음의 create view문이 이러한 예입니다.

```
create table user_perms (
    login_name char(10),
    region char(10) );

create table sales_recs (
    sales_id integer,
    sales_date date,
```

```
    sales_amt decimal(12,2),
    prod_id char(5),
    region char(10));

create view sales_view as
  select sales_recs.*
  from sales_recs, user_perms
  where sales_recs.region = user_perms.region and
        user_perms.login_name = user;
```

이 예에서는 쿼리의 user 키워드에서 해석된 것과 같이 테이블에 액세스하는 사용자의 로그인 ID에 기초하여 sales_recs 테이블에서 레코드를 선택합니다. 이 쿼리는 사용자에게 유효한 지역이 저장된 user_perms 테이블과 sales_recs 테이블 사이의 조인을 통해 보안을 시행합니다. 다음과 같이 뷰를 사용하여 판매 레코드를 간단하게 선택함으로써 이 쿼리에 대한 자세한 내용을 숨길 수 있습니다.

```
select *
from sales_view
where sales_date between "1/1/1998" and "3/30/1998"
```

이 쿼리를 실행하면 1998년 1월 1일과 1998년 3월 30일 사이의 sales 레코드를 검색합니다. 그러나 sales_view가 사용되고 있기 때문에 사용자가 볼 수 있는 판매 레코드(user_perms 테이블에 정의됨)만 쿼리에 의해 검색됩니다.

뷰에 키워드 사용

오늘의 판매량만 리턴하려는 경우에도 today 키워드를 이 쿼리와 결합하여 사용할 수 있습니다. 이 쿼리를 다음과 같이 뷰로 저장할 수도 있습니다.

```
create view todays_sales as
    select * from sales_view
    where sales_date = today;
```

이 뷰를 사용하면 다음과 같은 select문은 사용자가 "from"절에

참조된 sales_view 뷰에 의해 리턴된 대로 레코드를 볼 수 있는 지역에서 오늘 날짜로 입력된 판매량만 검색합니다.

```
select * from todays_sales;
```

전체 쿼리 참조

완전하게 정규화되었거나 부분적으로 정규화된 데이터베이스는 때때로 데이터를 검색할 때 복잡한 쿼리를 필요로 할 수 있습니다. 데이터베이스 뷰를 사용하면 이러한 복잡성을 숨길 수 있습니다. 판매 데이터를 검색하는 데 사용된 복잡한 세 개 테이블의 조인을 숨기려면 다음 쿼리를 사용하면 됩니다.

```
create view sales_startup as

    select sales_rec.*
        from sales_rec, region, product, sales_code, periods
        where
              sales_rec.region = regions.region and
region.region_type = "S" and
product.prod_code = sales_rec.prod_code and
product.prod_scale = "X" and
product.sales_period = period.period_code and
period.flag = "CURRENT";
```

이 뷰를 사용하여 다음의 쿼리를 사용하여 특정 판매원에 대해 해당 기준에 맞는 판매 레코드를 검색할 수 있습니다.

```
select * from sales_startup
where sales_startup.sales_id = 123;
```

이 쿼리는 뷰의 조인 및 필터 기준과 일치하고 sales_id가 123인 판매원에 대한 레코드를 검색합니다.

데이터베이스 테이블 삭제

drop table문을 사용하면 기존의 데이터베이스 테이블을 삭제할 수 있습니다. 테이블 작성자, 사용자 Informix 또는 데이터베이스 관리자만이 데이터베이스 테이블을 삭제할 수 있습니다. 삭제할 테이블은 반드시 비어 있지 않아도 됩니다. 테이블이 삭제될 때 테이블과 직접 관련되고 테이블에 종속된 모든 데이터베이스 객체(뷰, 트리거, 테이블 컬럼에 대한 제한 조건)도 데이터베이스에서 삭제됩니다.

dbaccess 및 SQL Editor와 같은 도구를 사용하면 drop table문을 실행 중인 사용자에게 프롬프트가 표시됩니다. 그러나 ESQL/C와 같은 일부 언어 도구를 사용할 경우에는 사용자에게 테이블을 삭제한다는 프롬프트가 표시되지 않는다는 점을 유의하십시오. drop table문의 구문은 다음과 같습니다.

```
drop table <table name>;
```

sales_recs 테이블을 삭제하려면 다음 명령문을 실행해야 합니다.

```
drop table sales_recs;
```

데이터베이스 뷰 삭제

데이터베이스 뷰는 drop view문을 사용하여 삭제할 수 있습니다. 뷰가 작성될 때 뷰를 정의하기 위해 일련의 엔트리가 시스템 카탈로그 테이블에 작성됩니다. 이 명령문은 시스템 카탈로그 테이블에서 뷰와 관련된 행들을 삭제하여 데이터베이스에서 뷰를 효과적으로 제거합니다. drop view문의 구문은 다음과 같습니다.

```
drop view <view name>
```

이름이 sales_view인 뷰를 삭제하려면 다음 명령문을 실행해야 합니다.

```
drop view sales_view;
drop database
```

데이터베이스 삭제

drop database문을 사용하여 특정 데이터베이스를 삭제할 수 있습니다. 비어 있는 데이터베이스, 시스템 카탈로그 테이블만 들어 있는 데이터베이스, 활성 사용자 세션이 없는 데이터베이스만 삭제할 수 있습니다. 따라서 관리자가 현재 사용 중인 데이터베이스를 실수로 삭제할 수는 없습니다. 사용자 informix나 root 또는 데이터베이스 관리자만이 데이터베이스를 삭제할 수 있습니다.

데이터베이스를 제거하면 이 데이터베이스가 있는 DBSPACE에서 해당 데이터베이스에 대한 시스템 카탈로그 테이블도 제거됩니다. drop database문의 구문은 다음과 같습니다.

```
drop database <database name>
```

이름이 sales_dw인 데이터베이스를 삭제하려면 다음 명령문을 실행해야 합니다.

```
drop database sales_dw;
```

alter table 명령문

alter table문을 사용하여 기존 테이블을 변경할 수 있습니다. alter table문으로 수행되는 작업은 여러 SQL DDL 및 DML 명령문을 실행하여 수행할 수 있으므로 관리자의 작업을 줄이고 이 작업을 수행하는 데 편리한 방법을 제공합니다. alter table문의 구문은 다음과 같습니다.

```
alter table <table name> <action>
```

수행되는 작업은 표 7-5에 나온 작업 중 하나입니다.

표 7-5 alter table문 작업

기능	설명
add	테이블에 컬럼을 추가합니다.
modify	테이블의 컬럼을 수정합니다.
drop	컬럼을 삭제합니다.
add constraint	테이블에 제한 조건을 추가합니다.
drop constraint	테이블에서 제한 조건을 삭제합니다.
modify next size	다음 익스텐트 크기를 수정합니다.
lock mode	테이블의 로크 모드를 변경합니다.

alter table문을 가장 자주 사용하는 경우는 테이블의 컬럼을 추가하거나 수정하는 경우입니다. 이 작업의 구문은 다음과 같습니다.

```
alter table <table name> add ( <column name> [<default clause>]
[not null] [<constraint definition>] [before <column name>] )
```

또는 alter table문을 사용하여 다음과 같이 기존의 컬럼을 삭제할 수 있습니다.

```
alter table <table name> drop ( <column name> )
```

alter table문의 또 다른 용도는 다음의 구문을 사용하여 기존의 컬럼을 수정하는 것입니다.

```
alter table <table name> modify ( <column name> [<default clause>]
[not null] [<constraint definition>] [before <column name>] )
```

다음의 SQL문은 alter table문을 사용하여 sales 테이블에서 기존의 컬럼을 삭제합니다.

```
alter table sales drop ( region );
```

둘 이상의 컬럼을 삭제할 경우 다음과 같이 명령문에 콤마로 구분된 컬럼 목록을 포함시킬 수 있습니다.

```
alter table sales drop ( region, sales_code );
```

sales 테이블에 컬럼을 추가하는 경우 다음의 명령문을 실행할 수 있습니다.

```
alter table sales add ( region char(5), sales_code char(2) );
```

region 컬럼을 수정하고 길이가 5인 원래의 문자 필드 지정대신 길이가 10인 문자 필드로 변경하려면 다음의 alter table문을 사용할 수 있습니다.

```
alter table sales modify ( region char(10) )
```

또는, alter table문을 사용하여 다음과 같은 구문으로 기존의 테이블에 제한 조건을 추가할 수 있습니다.

```
alter table sales add constraint check ( region in ( "NW", "SW", "SE",
"MW", "W" ) );
```

alter table문은 컬럼 이름을 변경하는 데 사용할 수 없다는 점을 유의하십시오. rename column문을 사용하면 이 작업을 수행할 수 있습니다.

alter table문은 원래의 컬럼과 변경된 추가 컬럼으로 새 테이블을 작성한 다음 이전 테이블에서 새 테이블로 작업을 복사하고 이전 테이블을 삭제함으로써 작업을 수행합니다. 이는 이전 테이블이 삭제될 때까지 원래의 테이블 크기가 두 배가 된다는 것을 의미합니다. 관리자는 alter table문을 실행할 때 이 점을 고려해야 합니다. (버전 7.3 이상에는 이러한 공간 문제를 없애주는 적절한 alter table이 있습니다.)

rename column 명령문

rename column문을 사용하여 기존 컬럼의 이름을 변경할 수 있습니다. 컬럼 이름은 해당 테이블 내에서 고유해야 하지만 전체 데이터베이스에서 고유한 이름일 필요는 없습니다. 이 명령문의 구문은 다음과 같습니다.

```
rename column < table name >.< column name >
              to < new column name >
```

이름이 region인 sales 테이블 컬럼의 이름을 sales_region으로 변경하는 경우 다음의 명령문을 실행할 수 있습니다.

```
rename column sales.region to sales_region
```

rename table 명령문

rename table문을 사용하여 기존 테이블의 이름을 변경할 수 있습니다. 이 명령문의 구문은 다음과 같습니다.

```
rename table < table name > to < new table name >
```

이 명령문은 데이터베이스 변경사항을 작성할 때 SQL 스크립트에서 유용하게 사용할 수 있습니다. 다음의 스크립트는 새 sales 테이블을 작성하고 이전 sales 테이블의 데이터를 새 sales 테이블로 복사합니다.

```
create table tmp_table ( sales_id integer,
                         region char(5),
                         sales_amt decimal(12,2) );

insert into tmp_table select * from sales where region in
( "NW", "NE" );
rename table sales to sales_bak;
rename table tmp_table to sales;
```

이 스크립트는 처음에 tmp_table로 이름 지정된 새 테이블을 작성합니다. sales 테이블에서 선택된 행은 이 테이블로 복사됩니다. 복사 작업이 완료되면 sales 테이블의 이름은 sales_bak로 변경되고 데이터가 복사된 테이블은 sales로 이름이 변경됩니다.

컬럼 무결성 및 참조 제한 조건

IDS 엔진은 참조 및 무결성 제한 조건을 모두 지원합니다. 참조 제한 조건은 데이터베이스에 있는 다른 테이블과의 관계가 자동으로 데이터베이스 레벨에서 시행될 수 있게 합니다. 무결성 제한 조건은 속성 도메인(유효 값 세트)이 데이터베이스 레벨에서 설정되고 시행될 수 있도록 합니다. 이들 기능은 모두 함께 데이터베이스에 저장된 데이터의 무결성과 품질을 상당히 향상시킬 수 있습니다. 이들 기능을 사용하면 데이터베이스 관리자는 데이터 품질을 보장하기 위해 데이터베이스를 사용하는 응용프로그램에 의존할 필요가 없으며 시행은 데이터베이스의 일부분으로 될 수 있습니다.

제한 조건은 데이터베이스 테이블이 작성될 때 간단한 구문을 사용하여 설정할 수 있습니다. create table문을 사용한 constraint 작성 구문은 다음과 같습니다.

```
create table <table name > (
        <column name> <data type> check <check constraint> |
            references <table reference>
...
```

컬럼 정의의 일부분으로 작성된 제한 조건은 단일 컬럼(제한 조건이 작성된 컬럼)만 참조할 수 있습니다. 다음의 명령문은 판매 데이터를 포함하는 테이블의 컬럼에 대해 몇 가지 제한 조건을 지정합니다.

```
create table sales (
sales_id serial primary key,
region char(5) check ( region in ("NW", "SW", "NE", "SE" ) ),
sales_code char(2) references sales_codes( sales_code ),
...
```

여기에서 create table문은 sales 테이블 컬럼에 대해 세 가지 제한 조건을 지정합니다. 첫 번째 제한 조건은 sales_id 컬럼을 기본 키 즉, 행을 고유하게 식별하는 값을 포함하는 컬럼으로 식별합니다. 정의에 따라 기본 키는 테이블의 행을 고유하게 식별하기 때문에 이 키는 고유한 식별자이어야 합니다. 따라서 기본 키 제한 조건을 구현하기 위해 IDS 엔진이 이 컬럼에 대해 고유한 인덱스를 지

정합니다.

테이블에 대해 지정된 다음 제한 조건은 region 컬럼을 "in"절 표현식에 지정된 값으로 제한합니다. 이 검사 제한 조건을 충족시키지 않는 region 값이 있는 레코드를 삽입하려고 시도하면 제한 조건을 위반했음을 나타내는 오류가 표시되고 삽입에 실패합니다.

sales 테이블에 대해 지정된 세 번째 제한 조건은 sales_code 컬럼에 대해 규정된 참조 제한 조건입니다. 이 참조 제한 조건은 sales_code 컬럼이 sales_code 테이블을 참조하고 sales_code 컬럼에 삽입된 모든 컬럼 값을 sales_code 컬럼의 값으로 sales_codes 테이블에 포함시켜야 함을 나타냅니다.

제한 조건 정의에서 둘 이상의 테이블 컬럼을 식별하려면 테이블 레벨 제한 조건을 정의해야 합니다. 이러한 기능은 다음 절에서 설명됩니다.

기본 키와 참조 키는 기본 및 참조된 테이블에서 인덱스를 작성하여 구현된다는 점을 유의하십시오. 기본 키가 작성될 때 기본 키가 정의된 테이블의 기본 키 컬럼에서 고유 인덱스가 작성됩니다. 컬럼에 대한 참조 제한 조건이 규정될 때에는 해당 컬럼의 참조된 테이블에서 인덱스가 작성됩니다(따라서 값에 대해 전체 테이블을 스캔할 필요가 없음).

인덱스는 이 키를 사용하여 읽기 작업의 성능을 향상시킬 수 있지만 쓰기(업데이트 및 삽입) 작업에서는 불리합니다. 따라서 테이블에 대해 많은 참조 제한 조건을 작성하면 많은 인덱스가 작성되어 업데이트 작업 중 성능에 영향을 미칠 수 있습니다. 그러나 성능에 미치는 영향은 참조 및 무결성 제한조건을 사용하면서 상당한 데이터 무결성 이점이 파생함으로써 균형을 이루게 됩니다.

테이블 레벨의 기본 키와 외부 키 정의

기본 키 및 외부 키 정의는 테이블 레벨에서 이루어질 수 있습니다. 사실, 여러 컬럼의 기본 또는 외부 키 제한 조건이 지정되는 경우, 컬럼 레벨이 아닌 테이블 레벨에서 입력해야 합니다. 다음의 예에서는 테이블 레벨 기본 및 외부 키 정의를 작성합니다.

```
create table region_link (
            sales_id integer,
            region char(5),
```

```
            reg_flags char(3),
      primary key ( sales_id, region ),
      foreign key (sales_id) references salesmen (sales_id) )
```

이 create table문은 여러 컬럼의 기본 키 정의를 정의합니다. salesmen 테이블을 참조하는 외부 키 정의도 작성됩니다. (외부 키 참조 제한 조건에서는 기본 키로 식별되고, 참조된 테이블의 컬럼을 참조해야 한다는 점을 유의하십시오.)

둘 이상의 컬럼을 참조하는 제한 조건은 테이블 레벨에서 정의해야 합니다. 예를 들어, 둘 이상의 컬럼과 관련된 검사 제한 조건은 다음과 같이 정의해야 합니다.

```
create table sales_stats (
            sales_id integer,
            trans_id integer, gross_sales decimal ( 12,2 ),
            cost_of_sales decimal(12,2),
            gross_profit decimal(12,2),
                check ( cost_of_sales <= gross_sales ) ;
```

이 테이블에서 cost_of_sales를 gross_sales의 값보다 작거나 같은 값으로 제한하는 제한 조건이 작성됩니다. 따라서 이 검사 제한 조건은 판매 발생 비용이 판매의 총 소득보다 클 수 없음을 나타내는 비즈니스 규칙을 시행합니다.

고유 컬럼 작성

고유 컬럼은 unique 제한 조건을 사용하여 작성할 수 있습니다. unique 제한 조건은 컬럼에 대한 고유 인덱스를 작성하는 것과 동일한 영향을 미칩니다(이는 실제로 구현되는 방법임). 다음의 SQL create table문은 이러한 제한 조건의 사용을 보여줍니다.

```
create table region (
    region char(5) primary key,
    region_type char(3),
    region_id integer unique );
```

이 create table문은 이름이 region인 테이블을 작성하고 region_id 컬럼을 고유 식별자로 식별합니다.

제한 조건 이름 지정

기본적으로 IDS 엔진은 제한 조건에 이름을 지정합니다. 그러나 제한 조건은 명시적으로 이름이 지정될 수 있습니다. 이는 제한 조건을 삭제해야 하는 경우(alter table문을 통해)에 유용할 수 있습니다. 다음은 제한 조건에 이름을 지정하는 예입니다.

```
create table region (
    region char(5) primary key constraint pk_region,
    region_type char(3) check ( region_type in ("SK", "ZE"))
        constraint region_type_check );
```

다음의 예에서는 기본 키 제한 조건이 region 컬럼에 작성되어 pk_region으로 이름이 지정됩니다. alter table문을 사용하여 이 제한 조건을 삭제해야 하는 경우 제한 조건 이름은 다음과 같이 참조될 수 있습니다.

```
alter table sales drop constraint ( pk_region);
```

모든 제한 조건에는 alter table문에서 참조할 수 있는 이름이 있다는 점을 유의하십시오. 제한 조건 이름이 명시적으로 지정되지 않은 경우 IDS 엔진은 기본 이름을 지정합니다. 이 기본 이름은 다음의 SQL문을 사용하여 식별할 수 있습니다.

```
select sc.constrname, sc.constrtype
from   systables st, sysconstraints sc
where  sc.tabid=st.tabid and
       st.tabname = "sales";
```

이 SQL문에는 systables와 sysconstraints 테이블의 시스템 카탈로그 테이블 쌍이 있습니다. 이 테이블들은 IDS 데이터베이스에 있는 테이블의 고유 식별자인 tabid 컬럼에 조인되어 있습니다. 쿼리는 필터 기준을 사용하여 sales 테이블만을 선택하지만(tabna

me = "sales") 유효한 테이블 이름으로 대체하여 해당 테이블에 대한 제한 조건을 검색할 수 있습니다.

기본적으로 제한 조건은 테이블이 업데이트된 직후에 테스트됩니다. insert문의 세트가 데이터베이스에 위임될 때까지 검사를 지연시키도록 이러한 동작을 변경할 수 있습니다. set constraints문은 다음의 구문을 사용하여 이러한 동작을 변경하는 데 사용됩니다.

```
set constraints [all]| <constraint name> immediate|deferred
```

이 명령문을 사용하여 제한 조건 검사는 트랜잭션이 완료될 때까지 지연시킬 수 있습니다. 이 명령문을 트랜잭션 내에서 실행해야 합니다.

데이터 액세스 – DML문

데이터베이스 내에서 데이터를 처리하고 업데이트하기 위해 수많은 명령문들을 사용할 수 있습니다. 이러한 명령문들을 데이터 처리 언어라고 하며 select, update 및 delete 명령문들이 이에 해당합니다. 이 명령문들은 SQL을 사용하여 수행되는 대량의 데이터베이스 상호 작용을 구성합니다. 다음 절에서 이 명령문들을 다루게 됩니다.

select 명령문

select문은 SQL문의 일꾼이라 할 수 있습니다. 이 명령문을 사용하여 데이터베이스를 쿼리하고 원하는 행 세트를 검색합니다. select문의 "where"절을 사용하여 행을 필터하고 2개 이상의 여러 테이블 간 조인을 표현할 수 있습니다. select문의 형식은 다음과 같습니다.

```
select [ first n ] <column list>
from <table list>
[where <filter criteria> ]
[group by <column list> ]
```

```
[having <filter criteria>]
[order by <column list>]
```

select 목록

select절에는 쿼리의 일부로 검색되는 컬럼의 목록이 들어 있습니다. 이러한 컬럼 목록은 수학적 계산이나 문자열 처리를 수행하기 위한 표현식을 포함할 수 있습니다. 별표(*)는 식별된 테이블의 모든 컬럼이 검색될 것임을 나타냅니다. 선택적인 first n절은 쿼리에 의해 지정된 수의 행이 리턴될 수 있도록 합니다. 이 수를 초과하여 쿼리된 레코드는 무시됩니다(이 기능은 IDS 버전 7.3 이상에서만 사용할 수 있음). select절은 select문의 필수적인 부분입니다. 다음의 쿼리는 쿼리 결과의 각 행에 대해 리턴할 두 개의 컬럼을 판매 테이블에서 선택합니다.

```
select region, sales_amount
from sales
```

다음 쿼리와 같이 표현식이 select 목록의 일부분이 될 수도 있습니다.

```
select region, sales_amount * 1.1
from sales
```

from 절

select문의 from절은 이 쿼리를 충족시키기 위해 쿼리될 테이블을 나열합니다. 이 절을 사용하여 나열된 테이블들 간의 조인 유형을 단순 조인(기본값) 또는 outer 키워드로 지정된 외부 조인으로 지정할 수 있습니다. 이 절은 select문의 필수적인 부분입니다. 다음 예에서는 from절에 지정된 대로 sales 테이블에서 레코드를 선택합니다.

```
select *
from sales
```

where 절

선택적인 where절은 실행되는 SQL문의 필터 및 조인 기준을 표현하는 데 사용합니다. 조인 기준은 from절에 나열된 하나 이상의 테이블을 조인하는 데 사용되는 컬럼을 표시합니다. 그리고 필터 기준은 식별된 테이블에 있는 전체 행 세트에서 포함시키거나 제외시킬 행을 표시합니다.

sales 테이블에서 모든 행을 검색하는 select문은 다음과 같이 작성됩니다.

```
select *
from sales
```

where절이 포함되지 않았으므로 테이블의 모든 행을 검색합니다. 이 쿼리의 결과 세트를 필터하려면 다음과 같이 where절이 포함되어야 합니다.

```
select *
from sales
where region = "NW"
```

이 select문은 sales 테이블에서 행을 검색하지만 결과 세트를 region 컬럼에 "NW" 값이 있는 행으로만 제한합니다. 결과 세트를 계속 필터하려면 다음과 같이 괄호를 사용하여 쿼리 결과를 묶어 쿼리를 작성해야 합니다.

```
select *
from sales
where region = "NW" and
            sales_code = 23 and
            (sales_date = "1/1/1998" or
            sales_date = "2/1/1998)
```

이 select문은 where절에 추가 필터 기준을 추가하여 sales_code가 23이고 sales_date가 1998년 1월 1일이거나 sales_date가 1998년 2월 1일인 행으로 결과 세트를 제한합니다. 쿼리에서 두 테이블 간의 조인 조건을 표현하려면 다음의 select문을 작성해야 합니다.

```
select sales.*, region.region_name
from sales, region
where sales.region = region.region
```

이 select문은 sales 테이블을 region 테이블과 조인합니다. 조인 조건은 region 컬럼을 사용하여 sales 테이블을 region 테이블에 조인합니다. where절이 제거된 경우(허용됨), 두 테이블 간의 조인은 결과적으로 첫 번째 테이블의 모든 행을 두 번째 테이블의 모든 행에 조인하여 두 테이블 간의 외부 카테시안 곱(Cartesian Product)을 검색하게 됩니다.

표시 레이블 지정

select 목록의 컬럼에는 표시 레이블을 지정할 수 있습니다. 표시 레이블은 표현식 컬럼이나 컬럼 이름이 컬럼의 내용을 명확하게 나타내지 않는 컬럼인 경우에 유용합니다. 이러한 컬럼 레이블은 작성된 쿼리 내에서만 의미가 있습니다. 다음 쿼리는 select 목록 컬럼의 표시 레이블을 정의하는 예를 보여줍니다.

```
select     region, sales_amount base_sale,
          (sales_amount * sales_discount) discounted_amt
from sales
```

이 쿼리에는 할인이 포함되어 있지 않으므로 base_sale로 이름 지정된 sales_amount에 대한 컬럼 레이블과 Sales_amount 컬럼에 판매 할인을 적용하는 표현식에 대한 discount_amt가 들어 있습니다.

정렬 순서 식별

order by절은 쿼리에 대한 정렬 기준을 지정할 수 있게 합니다. order by 컬럼 목록에 식별된 모든 컬럼이 select 목록에 나열됩니다. 표현식이 정렬 컬럼 목록에 식별되어 있는 경우 select 목록에서의 위치 순서나 다음 쿼리에서와 같이 컬럼에 대해 작성된 표시 레이블로 식별할 수 있습니다.

```
select region, sales_amount base_sale,
        (sales_amount * (1-discount_amount)) discounted_sale
from sales
order by base_sale, discounted_sales, 1
```

이 쿼리는 base_sale 및 discounted sale의 표시 레이블을 select문의 첫 번째 두 개의 정렬 기준으로 사용합니다. 정렬 목록에 식별된 마지막 컬럼은 select 목록의 위치 순서(이 경우에는 위치 1 즉, select 목록에 나타나는 첫 번째 컬럼인 region 컬럼)로 참조됩니다. 기본적으로 컬럼의 정렬 순서는 오름차순입니다. 선택적으로 다음 쿼리에 표시된 대로 desc 키워드를 사용하여 내림차순으로 지정할 수 있습니다.

```
select region, sales_amount base_sale,
        (sales_amount * (1-discount_amount)) discounted_sale
from sales
order by base_sale desc, discounted_sale desc, 1
```

이 쿼리는 앞의 쿼리와 동일한 결과를 리턴하지만 이 결과는 base_sale 금액, discounted_sale 금액 및 지역에 따라 내림차순으로 정렬됩니다. 따라서 결과 세트는 가장 큰 base_sale에서 가장 작은 base sale의 순서로 정렬된 다음, 가장 큰 discounted_sale에서 가장 작은 discounted sale의 순서로 정렬된 후 마지막으로 region 컬럼별로 정렬됩니다.

리턴되는 최대 행 수 지정

first n절을 사용하여 쿼리에 대해 리턴될 최대 레코드 수를 지정할 수 있습니다. 이 수는 first n절에 대해 하나의 숫자 매개변수로 지정됩니다. 지정된 수를 초과하는 모든 레코드는 무시됩니다. 다음 쿼리는 쿼리를 충족시키는 최대 50개의 레코드를 검색합니다. 쿼리 결과는 지역으로 필터되고 sales_amount별로 정렬되기 때문에 지정된 지역에 대해 상위 50개까지의 판매를 리턴합니다.

```
select first 50 *
from sales
where region = "NW"
order by sales_amount
```

레코드 그룹화

select문으로 레코드 그룹을 지정할 수 있습니다. 이러한 레코드 그룹을 집계 함수와 함께 사용하면 리턴 세트의 데이터 그룹에 대한 부분합계와 같은 일련의 집계 표현식을 제공할 수 있습니다. 집계 그룹으로 사용할 컬럼 목록 요소는 where절 뒤와 order by절 앞에 나타나는 group by절에 의해 지정됩니다. select 목록에 표시되는 컬럼만 group by절에 표시될 수 있습니다. 표현식을 그룹화에 사용하는 경우 select 목록의 위치 순서에 의해 지정됩니다. 집계된 컬럼은 group by 목록에 지정할 수 없습니다. 다음 쿼리는 sales 테이블에 대한 쿼리의 결과를 region별로 그룹화합니다.

```
select region, sum( sales_amount ) base_sale,
               sum( (sales_amount * (1-discount_amount)) )
discounted_sale
from sales
group by region
order by base_sale desc, discounted_sale desc, region
```

이 쿼리는 sales 테이블에서 레코드를 검색하여 sales_amount 컬럼에 대한 집계 함수와 sales_amount에 할인을 적용하는 집계 함수를 적용합니다. 지역 컬럼은 리턴된 데이터를 그룹화하는 데 사용할 컬럼으로 select 목록에 지정됩니다. 이 쿼리는 각 지역에 대한 sales_amount와 각 지역에 대한 할인된 sales_amount의 합계를 리턴하여 region별로 집계된 행을 리턴합니다. 정렬 순서는 집계와 그룹화가 수행된 후에 적용되며 지역의 내림차순(order by절의 desc 키워드로 지정됨) 기본 판매량 합계와 지역의 내림차순 할인 판매량 합계 및 지역별로 결과를 정렬합니다. 다음은 이 쿼리가 리턴하는 샘플 결과 세트입니다.

```
region   base_sale     discounted_sale
NW       115466.65      77362.65550000
SE       105559.98      70725.18660000
SW        92373.32      61890.12440000
MW         4773.32       3341.32400000
```

select문의 having절은 그룹화된 결과에 대한 필터 기준을 지정합니다(따라서 반드시 'group by'절과 함께 사용해야 함). 지역에 대해 집계된 기본 판매량이 10,000보다 큰 경우에 대해 앞에서 표시된 쿼리를 필터하려면, 다음 쿼리를 사용할 수 있습니다.

```
select region, sum( sales_amount ) base_sale,
sum( (sales_amount * (1-discount_amount)) )
discounted_sale
from sales
group by region
having sum( sales_amount) > 10000
order by base_sale desc, discounted_sale desc, region
```

having절은 group by절의 뒤와 order by절의 앞에 나타난다는 점을 유의하십시오. 이 쿼리는 다음과 같은 결과를 리턴합니다.

```
region    base_sale    discounted_sale
NW        115466.65     77362.65550000
SE        105559.98     70725.18660000
SW         92373.32     61890.12440000
```

테이블 별칭 지정

Informix에서는 테이블에 대해 짧은 이름을 사용할 수 있는데 이를 테이블 이름 별칭이라고도 합니다. 이 기능을 사용하여 다음과 같이 from절 목록에서 테이블에 짧은 이름을 지정할 수 있습니다.

```
select
from parts p
where p.initial_date < today and
p.parts_code in ("E", "F", "C");
```

이 예에서 테이블의 필터 기준은 테이블 별칭을 사용하여 지정합니다. 테이블 별칭은 다음 구문을 사용하여 from 목록에 지정됩니다.

```
...
from parts p
```

테이블의 별칭으로 문자 “p”가 지정되었으므로 select문 전체에서 테이블 이름 대신 이 별칭을 사용해야 합니다.

서브쿼리 사용

서브쿼리는 쿼리 내에 쿼리를 삽입하는 방법을 제공합니다. 서브쿼리는 where절에서 쿼리의 필터 기준에 사용되는 표현식의 오른쪽 부분으로 사용할 수 있습니다. 다음은 서브쿼리의 예를 보여줍니다.

```
select *
from sales
where sales.region in
(select unique region
from sales_tmp
where region_type = “E”);
```

이 서브쿼리는 쿼리에 대한 필터 기준으로 사용될 region 세트를 제공합니다. (중복 값을 제거하기 위해 unique 키워드가 서브쿼리에 사용되는 점을 유의하십시오.) 지역 세트가 알려졌으므로 이 명령문은 다음과 유사할 것입니다.

```
select *
from sales
where sales.region in ( “NE”, “SE”, “NW” );
```

서브쿼리에서 복잡한 쿼리를 구성하는 기초를 형성할 수 있습니다. 쿼리의 논리를 명확하게 나타내는 방법을 제공합니다. 다음은 여러 서브쿼리가 있는 쿼리의 예입니다.

```
select *
from sales
where region in (
    select unique region
    from sales_tmp
    where sales_amt > 10000) and
sales_disct <= (
    select max(sales_disct)
```

```
    from sales_tmp ) and
sales_code not in (
        select unique sales_code
        from sales_reject )
```

이 쿼리는 보다 복잡한 서브쿼리 세트와 서브쿼리를 사용할 수 있는 여러 가지 표현식의 예를 제공합니다. 첫 번째 서브쿼리는 등식 표현식에 의해 조사된 지역 세트를 리턴합니다. 이것은 앞의 예와 동일합니다. 그러나 다음 서브쿼리는 다음과 같이 "<=" 표현식 평가를 수행합니다.

```
… sales_disct <= ( select max(sales_disct) from sales_tmp )
```

sales_tmp 테이블에서 sales_disct 컬럼의 최대값을 선택하기 위해 집계 표현식 max(sales_disct)가 사용되었습니다. 이 쿼리가 sales_tmp 테이블의 값을 sales 테이블의 값과 비교하는 비교 보고서를 작성하는 데 사용되었다고 가정할 경우, 이 쿼리는 sales_disct 컬럼(판매에 대한 할인이 들어 있다고 가정할 수 있음)을 필터하여 같거나 낮은 할인을 사용한 행을 검색합니다.

다음과 같은 마지막 서브쿼리는 "not" 연산자를 사용하여 sales_reject 테이블에 포함된 sales_code를 기준으로 하여 sales 테이블에서 검색되는 레코드를 필터합니다.

```
… sales_code not in (
        select unique sales_code
        from sales_reject )
```

"not" 연산자는 강제로 SQL 처리가 sales_reject 테이블에 있는 판매 코드를 포함하는 레코드를 거부하도록 합니다. 이 경우 서브쿼리는 최종 결과 세트에서 제거할 판매 코드(sales_code) 세트를 생성합니다. 그리고 서브쿼리는 다음과 같이 포함 목록을 리턴할 수 있습니다.

```
… sales_code in (
       select unique sales_code
       from sales_reject )
```

이 서브쿼리는 SQL문이 쿼리에 대한 포함 목록으로 사용될 판매 코드 목록을 검색하도록 지시합니다. 이 쿼리는 서브쿼리에 의해 리턴되는 판매 코드 세트에 판매 코드가 있는 레코드를 검색합니다.

서브쿼리가 기본 쿼리의 테이블을 참조하는 경우, 이를 관련된 서브쿼리라고 하며 다음과 같습니다.

```
select *
from sales s
where s.region in (
      select unique region
      from sales_tmp st
      where st.sales_amt > 10000 and
         st.scode = s.scode );
```

이 쿼리에서는 sales_tmp 테이블을 스캔하여 발견된 지역 세트를 찾기 위해 판매 테이블을 스캔합니다. sales_tmp 테이블에서는 판매 금액(sales_amt)이 10,000보다 크고 sales_tmp 및 sales 테이블의 scode가 일치하는 판매를 스캔합니다. 이 쿼리는 다음과 같이 서브쿼리 대신 조인으로 다시 작성할 수 있습니다.

```
select s.*
from sales s, sales_tmp st
where s.region = st.region and
      st.sales_amt > 10000 and
      st.scode = s.scode;
```

이 쿼리에서는 sales 테이블을 스캔하고 sales 테이블과 sales_tmp 테이블 간의 조인에 기초하여 행을 리턴합니다. sales_tmp 테이블에서는 sales_amt가 10,000보다 크고 sales_tmp 테이블의 scode가 sales 테이블의 scode와 일치하는 행만 스캔합니다.

반복적 쿼리

테이블에 저장된 데이터의 관계에 반복적인 데이터 검색 작업이 필요한 경우가 있습니다. 다른 부품을 번갈아 사용하여 만들어지는

부품이 있는 행이 들어 있는 부품 테이블이 이와 같은 경우입니다. 이 테이블에서 행을 검색하려면 행을 검색한 후 이 행과 관련된 행이 같은 테이블에 들어 있는지 여부를 판별해야 합니다. 다음은 이러한 데이터 관계 유형의 예입니다.

```
create table parts
       (part_id integer,
        subpart_id integer,
        part_name char(20)
```

이 간단한 테이블에는 부품 번호와 부속부품 ID를 포함하는 약간의 부품 정보가 들어 있습니다. 부속 부품 ID는 동일한 부품 테이블에 있는 부품을 참조합니다. 이들 각 구성 부품에는 자체 구성 부품과 관련된 다른 부품이 있을 수 있습니다. 표 7-6은 이에 대한 예입니다.

표 7-6 샘플 부품 테이블 출력

part_id	subpart_id	part_name
10	0	도어
11	15	도어 핸들
15	18	잠금 장치
18	0	잠금 장치 회전판

이 테이블은 자동차 도어 부품의 부분적인 목록을 제공합니다. 도어에는 도어 핸들이 있고 도어 핸들에는 잠금 장치가 있습니다. 이들 부품은 네 개의 개별 부품으로 나열되지만 모두 단일 부품인 자동차 도어의 구성 요소입니다. 자동차 도어와 해당 구성 부품들을 모두 검색하기 위한 반복적 쿼리를 부분 분해라고 하며 다음과 같이 부분적으로 표현할 수 있습니다.

```
select pa.*, pb.*
from parts pa, parts pb
where pa.part_id = pb.subpart_id
```

이 쿼리는 부품 테이블에서 레코드들을 반복적으로 검색합니다. 레코드를 검색할 때 subpart_id 컬럼을 조사하고 테이블에서 part_id로 subpart_id의 값을 가진 레코드를 검색하므로 제공된 부품의

모든 부속 부품을 효과적으로 검색할 수 있습니다. 따라서 테이블에 있는 모든 부품의 부분 분해가 가능해지지만 실질적으로는 두 레벨까지만 가능합니다. 보다 완전한 부분 분해를 얻으려면 반복을 보다 완전하게 지원하는 언어가 필요합니다.

외부 조인 사용

대부분의 조인은 테이블 간의 단순 조인을 나타냅니다. 단순 조인을 사용하는 경우에는 지정된 조인 기준에 따라 두 테이블 간에 일치하는 행만 리턴됩니다. 한 조인 테이블에는 행이 들어 있지만 이와 일치하는 행이 다른 테이블에 들어 있지 않은 경우에는 행이 리턴되지 않습니다.

외부 조인을 사용하면 쿼리에 지정된 조인 조건이 충족되지 않더라도 행을 검색할 수 있습니다. 다음은 외부 조인의 예를 나타냅니다.

```
select sales.*, items.*
from   sales, outer items
where  sales.items_id = items.items_id and
       sales.sale_date <= today;
```

이 쿼리는 sales_id 컬럼, sales 테이블의 기본 키 및 items 테이블의 외부 키를 사용하여 sales 테이블을 items 테이블에 조인합니다. 쿼리는 sales_date 컬럼의 결과를 필터하여 items 테이블에 대한 조인이 외부 조인임을 나타냅니다. 따라서 items 테이블에 해당(조인된) 행이 들어 있는지 여부와 관계없이 sales 테이블에서 행을 검색할 수 있습니다. 이 쿼리의 결과는 다음과 같을 수 있습니다.

sales_id	sales_date	items_id	item_name
10	1/28/58	222	coffee mug
11	2/1/56	NULL	NULL
13	6/6/91	123	dino poster
15	8/17/88	NULL	NULL

sales 테이블에 대해서는 필터 기준을 충족시키지만 items 테이블에 대해서는 필터 기준을 충족시키지 않는 결과 행이 items 테이블

컬럼에 대해 NULL 값을 대신 사용한다는 점을 유의하십시오. 외부 조인을 사용하여 프로그래밍된 응용프로그램은 이 동작을 알아야 하며 이들 컬럼에 대한 NULL 값을 관리하기 위한 준비를 해야 합니다.

SQL 합집합

SQL의 union절은 SQL select문에 지정된 두 집합의 합집합을 나타냅니다. 기본적으로 union절은 중복 행을 제거합니다. 이 기본 동작은 all 키워드를 union절의 일부분으로 사용함으로써 무시할 수 있습니다. 다음과 같이 하나의 select문에 여러 union 작업을 지정할 수 있습니다.

```
    select *
    from sales
    where sales_id = 10
union
    select *
    from sales_tmp
    where sales_id = 20
union
    select *
    from db1@svr1:sales
    where sales_id = 30;
```

이 SQL문은 세 개의 select문의 합집합을 나타냅니다. union절을 사용하려면 select문의 컬럼 수가 일치하고 컬럼의 순서 위치와 관련된 자료형이 일치해야 합니다. 기본적으로 결과 세트에서 중복 행이 제거되므로 이 동작은 무시할 수 있습니다.

종종 union절을 조인 조건과 같이 다르게 표현할 수 있습니다. 즉, 앞의 명령문을 다음과 같이 표현할 수 있습니다.

```
select s1.*, s2.*, s3.*
from sales s1, sales s2, db1@svr1:sales s3
where s1.sales_id = s2.sales_id and
      s3.sales_id = s1.sales_id and
```

```
s1.sales_id = 10 and

s2.sales_id = 20 and

s3.sales_id = 30;
```

이 명령문은 중복 행이 있기는 하지만 이전 쿼리와 동일한 행을 검색합니다.

union절을 사용하여 값 세그먼트화

union절은 쿼리에 의해 리턴된 각 select 컬럼 목록에 상수 값을 삽입하여 리턴 값을 세그먼트화하는 유용한 수단이 될 수 있습니다. 그런 다음, 이 값은 다음 select문에 표시된 대로 마지막 쿼리 다음에 오는 order by절에 의해 사용됩니다.

```
select "C",* from sales
where region = "NW"

union

select "B",* from sales
where region = "SW"

union

select "A",* from sales

where region = "MW"

order by 1
```

이 쿼리는 sales 테이블에서 행을 검색하고 select 목록의 첫 번째 컬럼을 기준으로 하여 결과를 정렬합니다. 이 컬럼은 결과에 대해 인위적인 정렬 순서를 강제 실행하는 데 사용되는 문자 상수입니다. 이로 인해 첫 번째 select문에 의해 리턴된 값이 리턴 결과에서 먼저 정렬되고(컬럼 목록의 문자열 상수 "B"에 대해 정렬됨), 두

번째 select 및 세 번째 select 순으로 정렬됩니다(컬럼 목록의 “B” 및 “C” 값을 기준으로 하여). 이에 대한 결과는 다음과 같습니다.

```
(constant)  sales_id  sales_amount  sales_discount  region
     A          1        1193.33       0.300000       MW
     B          1       23093.33       0.330000       SW
     C          1       23093.33       0.330000       NW
```

기본적으로 union절은 중복 값을 제거합니다. 판매 결과를 리턴하는 쿼리에 나타나는 중복 값은 반드시 제거할 필요가 없으므로(판매가 간혹 동일한 지역의 동일한 판매원에 의한 동일한 금액인 경우가 있음), 다음 쿼리와 같이 union절과 함께 all 키워드를 사용해야 합니다.

```
select “C”,* from sales
where region = “NW”

union all
select “B”,* from sales
where region = “SW”

union all

select “A”,* from sales

where region = “MW”

order by 1
```

이 쿼리에는 중복 값이 있으며 다음 결과에서와 같이 지정된 필터 기준을 충족시키는 모든 판매가 들어 있는 리턴 세트를 생성합니다.

```
(constant)   sales_id   sales_amount  sales_discount  region
     A          1         1193.33       0.300000       MW
     A          1         1291.33       0.300000       MW
     A          1         2193.33       0.300000       MW
     A          1         1193.33       0.300000       MW
     B          1        23093.33       0.330000       SW
```

```
B          1        22011.23       0.330000       SW
B          1        13093.33       0.330000       SW
B          1        23093.33       0.330000       SW
C          1        11192.22       0.330000       NW
C          1        23093.33       0.330000       NW
C          1          183.32       0.330000       NW
C          1         1309.11       0.330000       NW
C          1        23093.33       0.330000       NW
```

insert 명령문

insert문은 지정된 데이터베이스 테이블에 행을 추가합니다. 행은 다음의 구문을 사용하여 삽입됩니다.

```
insert into <table name> [(column list)] values (<value
list>)
```

컬럼 목록은 선택적입니다. 컬럼 목록이 지정되지 않은 경우 테이블의 모든 컬럼이 사용됩니다(암시적 컬럼 목록). 컬럼 목록의 컬럼 수는 값 목록의 컬럼 수와 일치해야 하며 그렇지 않을 경우 오류가 리턴됩니다. 컬럼은 서수 자리에 기초하여 값 목록에 일치됩니다. 자료형은 호환 가능해야 하며 엔진은 값 목록의 값을 컬럼의 자료형으로 변환할 수 있어야 합니다. 그렇지 않을 경우 오류가 리턴됩니다. 컬럼 목록을 사용하지만 테이블의 모든 컬럼을 참조하지 않는 경우, 목록에서 생략된 컬럼에 대해 NULL 값을 대신 사용합니다.

다음은 insert문의 예입니다.

```
insert into sales values ( 10, "1/28/58", 233.33, 10);
```

이 명령문은 sales 테이블에 지정된 값을 삽입합니다. 또는 다음과 같이 컬럼 목록을 사용할 수 있습니다.

```
insert into sales (sales_id, sales_date, sales_amt)
values (10, "1/28/58", 233.33);
```

이 명령문은 세 개의 컬럼이 있는 컬럼 목록을 지정하며, 값 목록을 사용하여 컬럼 목록의 컬럼과 직접 관련된 세 개의 값을 지정합니다.

insert문은 select문을 사용하여 insert문에 대한 값을 검색할 수 있습니다. 따라서 다른 테이블의 데이터를 이 테이블에 복사할 수 있습니다. 이들 테이블은 동일한 데이터베이스 인스턴스나 다른 머신에 있는 다른 데이터베이스 인스턴스에 상주할 수 있습니다. 다음은 이 insert문을 사용하는 예를 보여줍니다.

```
insert into sales select * from db1@new_jersey:sales;
```

이 명령문은 원격 데이터베이스 인스턴스에 있는 판매 테이블에서 데이터를 검색합니다. 해당 인스턴스에서 로컬 인스턴스로 데이터를 복사합니다. (다른 데이터베이스 인스턴스에 액세스하기 때문에 두 데이터베이스 인스턴스가 동일한 로깅 모드에서 작동해야 한다는 점을 유의하십시오.)

update 명령문

update문은 테이블의 행을 업데이트하는 데 사용됩니다. 이 명령문은 업데이트할 테이블 이름과 테이블 내의 컬럼을 지정합니다. 선택적으로 where절을 사용하면 업데이트할 행을 지정할 수 있습니다. where절이 생략된 경우 테이블의 모든 행이 업데이트됩니다(오류 없이). 이 명령문의 구문은 다음과 같습니다.

```
update <table name> set <column list = column value list>
[where clause]
```

items 테이블을 업데이트하려면 다음의 명령문을 사용할 수 있습니다.

```
update items
set price = price * 1.1
where item_code in ("XX", "Z1", "BA" );
```

이 명령문은 items 테이블의 모든 행을 업데이트하며 가격 컬럼의 값을 10%(가격 * 1.1) 올립니다. 이 명령문은 item_code가 upda

te문의 where절에 참조된 세 개의 코드 중 하나에 해당하는 행만 업데이트합니다. 이 where절을 update문에서 제거하면 다음과 같습니다.

```
update items
       set cost = cost * 1.1;
```

이 명령문은 품목 테이블의 모든 행을 업데이트하므로 모든 품목에 대해 10% 가격 상승을 효과적으로 실행합니다. 가격의 10% 상승이 요구되는 경우, 이 명령문은 논리적인 명령문이 됩니다. 그러나 보다 제한적인 업데이트가 요구되는 경우, 이 명령문은 데이터베이스 오류가 아닌 논리적 오류를 나타냅니다. 이 명령문은 재고가 있는 모든 품목에 대해 10% 가격 상승을 실행해야 하는 경우에 실행하는 논리적 명령문이 됩니다.

한 명령문을 사용하여 테이블의 모든 행을 업데이트한다고 데이터베이스 오류가 발생하지는 않습니다. Informix 언어 도구(Informix-4GL, Informix-ESQL/C)로 실행할 경우, 테이블의 모든 행에 대한 업데이트 실행은 오류 메시지를 리턴하지 않지만 경고(sqlca.sqlwarn4)를 생성합니다.

업데이트가 트랜잭션에서 수행되는 경우 이를 롤백할 수 있습니다. 그러나 그렇지 않은 경우에는 데이터를 보관에서 복구해야 합니다. 따라서 응용프로그램 프로그래머들은 이 동작과 프로그램에 대해 잘 알아야 합니다.

delete 명령문

delete문은 기존 테이블에서 행을 삭제하는 데 사용합니다. 이 명령문은 다음의 구문을 사용하여 삭제 작업에 대한 테이블 및 필터 기준을 지정합니다.

```
delete from <table name>
[ where <filter criteria> ]
```

where절을 사용하지 않을 경우, 이 명령문은 기본적으로 테이블의 모든 행을 삭제합니다. 이 삭제 작업은 오류 없이 수행됩니다.

delete문은 SQL문 중에서 가장 위험한 명령문 중 하나입니다. where절이 없으면 delete문은 테이블의 모든 행을 삭제합니다. Informix 언어 도구(Informix-4GL, Informix-ESQL/C)에서 실행하는 경우, 테이블의 모든 행에 대해 delete문을 실행해도 오류 메시지가 리턴되지 않지만 경고(sqlca.sqlwarn4)가 생성됩니다.

삭제 작업이 트랜잭션에서 수행되는 경우 롤백할 수 있습니다. 그러나 그렇지 않은 경우에는 데이터를 보관에서 복구해야 합니다. 따라서 응용프로그램 프로그래머들은 이 동작과 프로그램을 잘 알아야 합니다.

기타 SQL

다음 절에서는 이 절에 앞서 다루었던 SQL DDL 또는 DML 명령문에 적절하지 않은 여러 주제에 대해 설명합니다. 또한 한 테이블에서 다른 테이블로 레코드를 복사하고 임시 테이블을 사용하는 프로세스에 대해 설명합니다. 그 밖에 Informix-SQL 확장 명령문인 load와 unload에 대해서도 설명합니다.

SQL 복사 작업

Informix(및 일반적으로 SQL)는 명시적인 복사 명령문을 제공하지 않습니다. 데이터는 insert문과 select문을 함께 사용하여 한 테이블에서 다른 테이블로 복사됩니다. 다음은 이 작업의 구문입니다.

```
insert into <target table name>
   select from <source table name>
```

이러한 방식으로 사용된 select문의 제한사항은 select문에 있는 컬럼 수와 자료형이 대상 테이블에 대한 insert문에 있는 컬럼 수 및 자료형과 호환 가능하다는 점입니다. 컬럼 수는 정확하게 일치해야 하며 소스 컬럼의 자료형은 대상 컬럼의 자료형과 같거나 이 자료형으로 변환할 수 있어야 합니다. 다음은 이 작업의 예를 나타냅니다.

```
insert into sales_holding( sales_id, region, sales_amt)
    select (sales_id, region, sales_amt)
    from sales
    where region in ("NW", "SW", "NE" );
```

이 명령문은 sales 테이블에서 특정 컬럼을 선택하여 이를 sales_holding 테이블로 복사합니다. select문에 대한 필터 기준은 region이 "NW", "SW" 또는 "NE" 중 하나인 행만 선택하는 것입니다.

load 및 unload 명령문

load문은 ASCII 구분 파일에서 데이터베이스로 데이터를 로드할 수 있게 하는 Informix SQL 확장 명령문입니다. 파이프 문자(|) 구분 문자를 사용하거나 선택적 delimiter절을 사용한 load문에서 참조된 구분 문자를 사용하여 파일을 구분해야 합니다. load문의 구문은 다음과 같습니다.

```
load from <file name> [ delimiter 'c']
insert into <table name> | values (<column list> )
```

load문의 insert문 구성 요소는 Informix-SQL insert문처럼 컬럼 목록을 지원합니다. 다음 명령문은 load문의 사용을 보여줍니다.

```
load from sales_data.dat
insert into sales values (sales_id, region, sales_amt);
```

이 명령문에서는 데이터가 sales.dat 파일에서 로드되어 sales 테이블의 특정 컬럼에 삽입됩니다. sales 레코드의 다른 모든 컬럼은 삽입된 각 행에 대해 NULL 값으로 설정됩니다. sales.dat 파일의 내용은 ASCII 형식이며 다음을 포함합니다.

```
101|NE|1222.22|
```

구분 문자가 첫 번째 컬럼을 시작할 필요는 없다는 점을 유의하십시오. 문자열도 따옴표로 묶을 필요가 없습니다. load문은 데이터를 읽은 다음 변환이 필요하다고 판단되면 변환을 수행합니다. 로드 파

일에 특수 문자를 포함시키려면 이 문자 앞에 "\" 문자가 표시되어야 합니다. 예를 들어, 테이블에 대한 입력에 구분 문자를 포함시키려면 로드 파일에 다음과 같은 라인을 표시할 수 있습니다.

```
101|\|N\|E|1222.22|
```

테이블에서 ASCII 파일로 데이터를 언로드하려면 unload문을 사용하십시오. unload문의 구문은 다음과 같습니다.

```
unload to <table name> [ delimiter 'c' ] <select statement>
```

unload문은 ASCII로 변환하고 언로드 파일에 기록할 레코드를 식별하는 데 select문을 사용합니다. 언로드 파일에는 언로드 프로세스를 위해 컬럼들이 구분 문자로 구분된 ASCII 데이터 행이 들어갑니다. 구분 문자는 기본적으로 파이프 기호(|)를 사용하지만 unload문에 delimiter절을 사용하여 선택적으로 지정할 수 있습니다. 다음 명령문은 select문에 의해 지정된 데이터 행을 sales 테이블에서 ASCII 파일로 언로드합니다. "NW" 지역의 레코드만 파일로 언로드됩니다.

```
unload to sales.dat select * from sales where region = "NW"
```

임시 테이블 사용

때로는 임시 테이블을 사용하는 것이 유용할 수 있습니다. 이로 인해 수행해야 하는 디스크 입출력의 양은 늘어날 수 있지만 처리 주기에서 필수적인 부분으로서, 이를 통해 데이터를 저장한 후 SQL 문을 사용하여 처리할 수 있습니다.

임시 테이블은 create temp table문(create table문의 변형)을 사용하여 명시적으로 작성하거나 select문의 into temp절을 사용하여 작성할 수 있습니다.

임시 테이블은 테이블을 작성한 세션이 활성 상태인 동안 존재합니다. 따라서 임시 테이블을 사용하여 수행해야 하는 처리는 해당 세션 동안에 수행해야 합니다. 이는 배치 프로세스의 개발자가 처리 중에 작성된 임시 테이블을 제거하는 하우스키핑 작업을 염려할 필요가 없으므로 일반적으로 유용한 메커니즘으로 여겨집니다. 임시 테이블 제거는 엔진 세션이 종료하면 자동으로 이루어집니다. 다음은

create temp table문을 사용하여 임시 테이블을 작성하는 명령문의 예입니다.

```
create temp table sales_tmp
   ( sales_id integer,
     region char(5),
     sales_amt decimal(12,2));
```

이 명령문은 세 개의 컬럼이 있는 임시 테이블을 작성합니다. 이 테이블은 DBSPACETEMP 구성 매개변수에서 임시 DBSPACE로 지정된 DBSPACE 중 하나에 작성되거나 임시 DBSPACE가 식별되지 않은 경우, root DBSPACE에 작성됩니다.

명시적 임시 테이블을 작성하려면 테이블에 저장할 데이터의 각 컬럼의 크기와 자료형을 명령문에 명시적으로 지정해야 합니다. 이 구문 대신 select문과 함께 into temp절을 사용할 수 있습니다. 이 절을 사용하면 select문에 지정된 컬럼의 자료형과 길이를 사용하여 임시 테이블을 암시적으로 작성할 수 있습니다. 이 명령문의 예는 다음과 같습니다.

```
select sales_id, region, sales_amt
from sales
where region in ("NW", "NE", "SW" )
into temp sales_tmp;
```

이 명령문은 이름이 sales_tmp인 임시 테이블을 작성합니다. 이 임시 테이블에는 세 개의 컬럼 즉, select문의 컬럼 목록에 지정된 대로 sales_id, region 및 sales_amt 컬럼이 있습니다. 필터 기준을 충족시키는 판매 테이블의 데이터 행이 임시 테이블로 복사되며 테이블을 작성한 프로세스로 임시 테이블에 액세스할 수 있습니다.

Informix 내장 프로시저 언어(SPL)

SQL 언어는 결코 프로그래밍 언어로서 만들어진 것이 아닙니다. 변수를 제공하지 않고, 흐름 제어 명령문이 없으며, 특별한 오류 보고 기능을 제공하지 않고, 함수를 작성하여 저장하는 능력을 제공하지 않는 비프로시저 언어입니다. 그러나 Informix 내장 프로시저 언

어(SPL)는 이러한 기능을 제공합니다.

내장 프로시저는 SQL과 연결하여 사용할 수 있는 제한된 프로그래밍 언어를 제공합니다. Informix-SPL은 SQL의 기능을 확장하여 어느 정도의 비즈니스 논리를 데이터베이스에 저장하여 액세스할 수 있습니다.

Informix SPL을 통해 바이트, 일련 번호 및 텍스트와 같은 경우만을 제외하고는 유효한 SQL 자료형으로 변수를 작성하고 처리할 수 있습니다. 변수는 편리한 "like" 구문을 사용하여 선언할 수 있습니다. 이 구문은 변수에 대한 데이터베이스 바인딩 유형을 제공하며, SPL에 데이터베이스 테이블 선언을 사용하여 자료형과 길이 정보를 검색하도록 지시합니다.

내장 프로시저는 변수를 매개변수로 받을 수 있으며 하나 이상의 값을 리턴할 수 있습니다. 매개변수는 유효한 자료형이면 모두 사용 가능합니다.

Informix-SPL은 표 7-7과 같이 여러 가지 프로그래밍 흐름 제어 명령문을 제공합니다.

IDS 내장 프로시저 언어는 내장 프로시저 함수가 자체적으로 호출할 때 반복을 사용할 수 있도록 합니다. 이 기능은 여러 논리적 문제를 해결하는 데 유용하며, 종종 해당되는 반복적인 솔루션보다 문제에 대해 보다 명쾌한 솔루션을 제공하기도 합니다.

표 7-7 SPL 흐름 제어 명령문

명령문	용도
foreach	select 결과 전체 반복
for	지정된 반복 수만큼 루핑
while	부울 조건이 충족되는 동안 루핑
if	조건문
let	변수 지정

다음 표기와 같이 Informix-SQL문과 동일한 구문을 사용하여 주석을 삽입할 수 있습니다.

```
- this is a comment
{this is a comment}
```

내장 프로시저는 다음과 같은 구문을 사용하여 작성됩니다.

```
create procedure <procedure name> ( [<parameter list>] )
[returning <return value list>;]
<SPL statements>
end procedure;
```

다음 명령문을 실행하여 sales_update라는 이름의 프로시저를 작성할 수 있습니다.

```
create procedure sales_update( sales_id integer )
...
end procedure;
```

내장 프로시저의 본문에는 유효한 SPL 또는 SQL문이 여러 개 있습니다. SQL과 결합된 SQL 흐름 제어 명령문은 데이터베이스 객체 처리에 상당히 유용합니다. 다음 절에서는 Informix-SPL을 구성하는 다양한 언어 명령문을 자세하게 설명하며 사용 예를 제공합니다.

if 명령문

SPL if문은 조건 분기에 사용됩니다. 조건식을 평가하여 표현식이 TRUE로 평가되면 then절 다음의 명령문 블록을 실행합니다. 조건식이 FALSE로 평가되면 then절의 명령문은 무시되며 선택적 else절이 코드되어 있으면 여기에서 명령문이 실행됩니다. 그렇지 않으면, if문을 종료하는 end if절이 끝난 다음에 명령문 실행이 시작됩니다. 조건식이 TRUE나 FALSE에 대한 수학적 값으로 평가될 수 있게 하는 C 프로그래밍 언어와는 달리, SPL 조건식은 두 개의 값과 관계(표 7-8 참조) 또는 SQL 부울 표현식을 포함해야 하며 TRUE 또는 FALSE 값으로 평가되어야 합니다.

표 7-8 Informix 관계 연산자

연산자	설명
<	보다 작음
>	보다 큼
<=	보다 작거나 같음
>=	보다 크거나 같음
=	같음
!=	같지 않음

Informix SQL 연산자 IN, NOT IN 및 EXISTS도 SQL 서브쿼리와 함께 SPL 조건식에서 사용할 수 있습니다. 다음은 if문의 구문입니다.

```
if [ NOT ] <conditional expression(s)> then
   <SPL statements>
[else <SPL statement> ]

end if;
```

다음의 예와 같이 조건식 내에 서브쿼리를 사용하면 표현식이 간결해집니다.

```
...
if ( "NW" in (select unique region from sales) )
   then
         return 1;
else
         return 0;
end if;

if ( var1 in ( 1, 2, 3 ) ) then
         return 1;
else
         return 0;
end if;
...
```

SPL은 case 또는 switch문을 지원하지 않으므로 여러 번의 평가가 필요한 프로시저는 다음과 같이 if/then/else 스타일의 조건 평가를 사용해야 합니다.

```
...
if ( psales_code = "A" ) and
   (s1.region = "NW" ) then

    select region_code
    into pregion_code
```

```
    from region
    where region_type = 5;

let region_count = region_count + 1;

elif ( psales_code = "B" ) and
     (s1.region = "NE" ) then
      select region_code
      into pregion_code
      from region
      where region_type = 6;

      let region_count = region_count + 1;

elif ( psales_code = "E" ) and
     (s1.region = "SW" ) then
      select region_code
      into pregion_code
      from region
      where region_type = 4;

      let region_count = region_count + 1;

end if;
 ...
```

이 예에서 현재 레코드의 지역은 이 레코드에 지정된 판매 코드에 대해 평가됩니다. 이 if/then/elif 명령문 절은 지역과 판매 코드의 조합을 지역 유형 코드(if/then 블록에서 실행되는 select문에 있는)로 매핑될 수 있도록 합니다. 이것은 SQL로만 코드할 수 있는 경우 어려워질 수 있는 복잡한 논리입니다. (버전 7.3에 도입된 SQL case절은 SQL select문에서 조건 매핑을 수행하는 기능을 제공하지 않습니다. 이 절에 대해서는 이 장의 뒷부분에서 설명합니다.)

return 명령문과 returning 절

Informix-SPL 프로시저는 하나 이상의 값을 리턴할 수 있습니다.

리턴 값은 내장 프로시저 선언의 선택적 returning절에서 선언되어야 합니다. returning절의 구문은 다음과 같습니다.

```
create procedure <procedure name> ( <procedure parameters> )
       [ returning <data type declaration>… ]
```

returning절은 유효한 SPL 자료형의 여러 리턴 값을 식별할 수 있습니다. return절을 사용하여 값이 리턴될 때, 자료형은 returning절에 선언된 자료형의 수 및 위치와 일치해야 합니다. 다음은 SPL 선언에 사용하는 returning절의 예를 나타냅니다.

```
create procedure proc1()
       returning integer, integer, char(20);
…
end procedure;
```

프로시저에서 하나 이상의 값을 리턴할 때 return문이 사용됩니다. return문의 구문은 다음과 같습니다.

```
…
return <variable list> [ with resume ]
…
```

return문의 목적은 return문의 범위(프로시저에서 선언되었거나 전역 범위임) 내에 있는 하나 이상의 값이 포함된 변수 목록입니다. 기본적으로 return문은 값을 리턴하고 내장 프로시저를 종료합니다. 내장 프로시저를 결과 세트에서 반복하여 여러 데이터 행을 리턴해야 하는 경우 이 동작을 무시해야 합니다. with resume절을 이용하면 내장 프로시저에서 값이 리턴되고 다음과 같이 return문 다음의 명령문을 사용하여 프로시저 실행을 재개할 수 있습니다.

foreach 명령문

foreach문을 사용하면 SPL로 커서를 효과적으로 작성할 수 있습니다. SQL문을 실행한 다음 결과를 반복하여 각 결과 세트에 대해 SPL문 세트를 실행합니다. 다음은 foreach문의 구문입니다.

```
foreach <controlling SQL statement>

   <SPL statements to execute>

end foreach;
```

foreach문은 SQL문의 실행에 의해 리턴된 각 행에 대한 루프 또는 실행을 계속합니다. SQL문이 더 이상 리턴할 행이 없음을 나타내면 foreach 루프는 종료되고 end foreach문 뒤의 명령문을 사용하여 SPL이 실행을 시작합니다. 다음의 SPL 샘플은 foreach 루프를 실행하고 return문을 resume절과 함께 사용하여 결과를 리턴합니다.

```
...
foreach select sales_rec_code
        into psales_rec_code
        from sales

    if psales_rec_code = 8 and
       psales_rec_code
           then
             return ( psales_rec_code ) with resume;
end if;
end foreach;
...
```

이 SPL 조각은 select문을 반복하고, 리턴된 각 값에 대해 값을 평가합니다. 그리고 값은 특정 조건이 충족될 때 프로시저로부터 리턴됩니다.

call 명령문

SPL call문은 SPL문 내에서 내장 프로시저를 실행합니다. call문의 구문은 다음과 같습니다.

```
call <procedure name>( <parameter1>, <parameter2> )
     [returning <value1>, <value2> ]
```

내장 프로시저는 call문을 사용하여 다른 내장 프로시저를 호출함으로 추가 정보를 검색할 수 있습니다. 따라서 내장 프로시저 개발에 대해 모듈식 접근법을 사용할 수 있습니다. 다음은 call문의 예입니다.

```
...
foreach select sales_id into psales_id from sales

    call getAddLData( psales_id ) returning pdata1, pdata2;
  end foreach;
```

이 call문은 foreach문 내에서 실행됩니다. foreach 루프가 반복될 때마다 sales_id를 사용하여 내장 프로시저 getAddLData가 호출되며 pdata1과 pdata2의 두 값을 변수로 리턴합니다.

while 루프

while 루프는 특정 루프 조건이 충족되는 동안 명령문 세트를 실행하는 데 사용됩니다. 루프에서 테스트되는 조건은 유효한 SPL 조건입니다. 다음은 while 루프의 구문입니다.

```
while <condition>
    <statement block>
end while;
```

변수가 특정 값으로 설정될 때까지 루프를 실행하려면 다음 명령문을 실행해야 합니다.

```
...
let cflag = 1;
while ( cflag = 1 )

     select sales_id, sales_amt, region
     into psales_id, psales_amt, pregion
     where scode = pscode;

     if psales_amt/10000 < 10000 then
```

```
        let cflag = 0;    - force exit from loop
        continue while;
    end if;

    if ( cflag ) then

    insert into sales_tmp (sales_id, sales_amt, region)
                    values ( psales_id, psales_amt,
                             pregion );
    end if;

    insert into tracking_table values ( sales_id, sales_amt );

end while;
...
```

이 코드는 cflag가 1로 설정되어 있는 동안 루프를 수행합니다. cflag 변수의 값은 sales_amt에 따라 좌우됩니다. 이것은 c 변수의 값을 검사하는 조건문이며 특정 조건을 충족하면 cflag의 값이 영(0)으로 설정되어 다음 번 통과 시에 루프를 강제로 종료시킵니다.

for 명령문

for문은 설정된 반복 수에 따라 하나 이상의 조치를 수행하는 데 사용되는 루프 명령문입니다. 이 명령문의 구문은 다음과 같습니다.

```
for <control variable> ( <starting value> to <ending value>
[STEP] <step value>)
<statements to execute>

end for;
```

다음과 같은 형식으로 for 루프를 실행할 수도 있습니다.

```
for <control variable> = <starting value> to <ending value>
[STEP <step value>]
```

for 루프 표현식은 루프가 시작되기 전에 계산됩니다. 제어 변수가 종료 값으로 지정된 값에 도달하면 루프가 종료됩니다. step절을 사용하지 않는 한 증분 값은 기본적으로 1이며, step절을 사용할 경우 증분 값은 step절에 지정된 값입니다.

제어 변수는 이전에 지정된 변수이어야 하지만 루프가 시작되면 초기화되기 때문에 루프 전에 초기화할 필요가 없습니다. 루프 내의 명령문은 루프가 반복될 때마다 실행됩니다.

제어 변수 값의 범위에 대한 반복은 두 가지 방법으로 지정할 수 있습니다. 한 가지 방법은 "=" 기호를 사용하여 제어 변수에 시작 값을 지정하고 "to"절을 사용하여 종료 값을 지정하는 것입니다. 다음의 명령문은 이러한 방법을 보여줍니다.

```
...
    for n = 2 to 20 step 2

      insert into table test_table values (n, 1, 2, 3, 4);

    end for;
...
```

이 명령문은 10회의 반복을 수행하며 매번 insert문을 실행합니다. 제어 변수는 변수 n입니다. 변수는 값 2에서 시작하여 제어 변수가 값 20에 도달할 때까지 2씩 증가합니다.

tip *for 루프는 테스트 데이터를 생성하는 유용한 방법이 될 수 있습니다. 다음의 내장 프로시저는 for 루프를 작성하고 함수 매개변수로 지정된 반복 횟수만큼 반복하며 반복될 때마다 sales 테이블로 삽입을 실행하여 테스트 판매 레코드를 sales 테이블에 삽입합니다.*

```
create procedure sales_test_data( iteration_count integer )

define loop_counter, n integer;
define region_code like sales.region;
define sales_code like sales.sales_code;
define sales_amt like sales.sales_amt;

for n = 1 to iteration_count
```

```
let sales_amt = n * 239.22;

- make some attempt to disperse these values among the data set
if ( mod(n,2) = 1 ) then
   let region_code = "NW";
else
   let region_code = "NE";
end if;

if ( mod(n,3) = 1) then
   let sales_code = 6;
else
   let sales_code = 10;
end if;

insert into sales values (n, region_code, sales_code, sales_amt );

end for;
end procedure;
```

이 프로시저는 작성할 테스트 레코드의 수를 나타내는 숫자를 받습니다. 이 숫자는 루프를 실행하는 반복 횟수를 설정하는 데 사용됩니다. 증분 값은 기본값인 1을 그대로 사용합니다. 루프 내에서는 실제적인 값 세트를 작성하려고 시도합니다. 달러로 표시된 판매 금액은 루프 카운터의 함수이며 판매 코드 및 지역 코드의 값은 루프 카운터가 홀수 또는 짝수인지에 따라 결정됩니다.

begin/end 명령문 블록

begin/end문 블록은 명령문의 세트를 함께 그룹화하는 데 사용됩니다. 변수 선언은 begin/end 블록 내에 작성될 수 있으며, 이들 선언은 내장 프로시저의 다른 부분에 있는 선언을 마스크할 수 있으므로 블록 내의 코드에 대해 특정 레벨의 캡슐화를 제공합니다. 변수를 코드에서 사용되는 위치에 가까운 곳에 선언하여 코드를 보다 읽기 쉽게 만들 수도 있습니다. 다음은 이 명령문의 사용 예입니다.

```
create procedure sales_proc()

define n integer;

for n = 1 to 20

    begin
      define n integer;

        for n = 1 to 30

          inset into sales(n, "NW", 3, 3334.33);

        end for;
    end; - ends the block

end for;
end procedure;
```

이 명령문은 두 개의 for 루프를 실행하여 begin/end 블록을 사용하는 예를 보여줍니다. 외부 for 루프는 n이라고 하는 제어 변수를 사용합니다. 내부 for 루프는 begin/end 블록 내에 있으므로 이 블록 내에 작성된 모든 변수 선언을 마스킹합니다. 따라서 이 블록에서 선언된 변수 n은 프로시저의 시작 부분에 선언된 변수와 다른 변수입니다. 변수 n의 내부 선언은 begin/end 블록의 외부 범위에 해당하지 않으므로, 외부 for 루프는 내부 for 루프가 실행되는 동안 계속해서 올바로 실행될 수 있습니다.

begin/end 블록을 사용하면 보다 간결한 코드를 제공할 수 있으며, 변수가 사용되는 위치와 가까운 곳에 변수를 선언하고 이전 선언을 모두 마스킹할 수 있습니다.

continue 명령문

continue문은 포함된 for/while/foreach 루프의 처음으로 프로그램 실행을 분기하는 데 사용됩니다. 이 명령문이 발견되면 루프 내의 나머지 명령문은 무시됩니다. 이 명령문의 구문은 다음과 같습니다.

```
continue while | for | foreach;
```

다음 SPL 조각은 continue문의 사용을 보여줍니다.

```
...

foreach select sales_id, sales_name
    into psales_id, psales_name
    from salesman

    if psales_name = "FRED" then
    continue foreach;
    end if;

    insert into sales_bonus values ( psales_id, psales_name);
);

end foreach;

...
```

이 프로시저에서 foreach 루프를 실행하여 판매원 테이블에서 레코드를 검색합니다. 루프를 반복할 때마다 psales_name 변수를 검사하여 sales_name이 "FRED"인지 여부를 판별합니다. sales_name이 "FRED"이면 continue foreach문을 실행하고, 루프의 시작으로 실행을 분기하여 if/then문 다음에 오는 insert문을 생략합니다. if/then문의 일부인 조건 세트에 기초하여 삽입 작업(또는 그 뒤의 다른 모든 명령문)을 제거하는 결과가 됩니다.

raise exception 명령문

raise exception문은 특정 SQL 오류를 생성하거나 발생시키는데 사용됩니다. 이 명령문을 통해 프로시저는 반드시 SQL 오류를 생성하지 않을 수 있는 조건 세트에 기초하여 오류를 생성할 수 있습니다. 다음은 raise exception문의 구문입니다.

```
raise exception <SQL error>, <ISAM error>, <error text>
```

"SQL error"는 지정된 Informix-SQL 오류 번호 중 하나를 식별하는 정수이고, "ISAM error"는 지정된 Informix ISAM 오류 번호(일반적으로 LOW 레벨의 데이터 액세스 오류) 중 하나를 식별하는 정수입니다. error text는 발견된 오류에 대한 정보를 제공하는 문자열입니다.

발생된 예외는 기존의 SQL 오류일 수 있습니다(SQL문에 프로시저가 사용되었으며 개발자가 다른 SQL문에 대해 트랩되는 오류 코드로 SQL문을 실패하게 하려는 경우에 유용함). 또는 다음과 같이 내장 프로시저에 고유한 오류 메시지를 사용하여 특수 오류 번호 -746을 사용하는 오류를 생성할 수 있습니다.

```
...
 if psales_amt < 0 then raise exception -746, 0, "Bad data in
procedure sales_update"
 end if
...
```

이 SPL 조각에서 조건문이 TRUE이면 오류의 성격을 나타내는 오류 텍스트와 함께 예외 -746이 발생합니다. 이 내장 프로시저를 실행하는 코드는 오류를 트랩하고 오류 번호와 오류 텍스트를 둘 다 받아 이에 따라 적절히 조치합니다.

let 명령문

let문은 변수 지정을 수행하는 데 사용됩니다. 몇몇 언어와 달리 SPL은 등식 연산자(즉, n = 1) 사용을 기초로 하여 지정을 수행하지 않습니다. let문은 SQL처럼 유형 변환을 시도하여 정수를 10진수로 변환하고, 문자열이 숫자 값을 포함하는 경우 이 문자열을 숫자로 변환합니다.

let문은 유연성이 있어 하나의 명령문에서 여러 값 지정을 수행하는 데 사용할 수 있으며, SQL문을 오른쪽 표현식으로 사용할 수 있습니다. 변수 지정의 구문은 다음과 같습니다.

```
let <variable1>, <variable2> ··· = <value1>, <value2> ···;
```

오른쪽에 있는 값은 왼쪽에 있는 변수 수와 일치해야 하며 그렇지 않으면 오류가 리턴됩니다. 한 명령문에서 세 개의 변수에 대해 값 지정을 수행하려면 다음의 구문을 사용할 수 있습니다.

```
let var1,var2,var3 = val1, val2, val3;
```

값과 대상 변수의 자료형은 일치해야 하며 그렇지 않으면 적당한 변환이 수행되어야 합니다. 표현식의 오른쪽에 있는 값은 다음과 같이 표현식의 왼쪽에 있는 변수의 자료형과 일치하는 하나 이상의 값으로 해석되는 표현식이 될 수 있습니다.

```
let var1,var2 = spl_func1(), spl_func2();
```

또는 다음과 같이 SQL문을 사용하여 표현식의 오른쪽에 있는 값을 리턴할 수 있습니다.

```
let var1 = (select col1, col2 from qtab where scode=10);
```

값 지정에 사용되는 SQL문을 괄호로 묶어야 한다는 점에 유의하십시오. 이러한 형식의 변수 지정은 select into문과 동일하지만 어떤 면에서는 보다 명확하고 간결합니다. 다음의 명령문은 SPL을 사용한 변수 지정을 보여줍니다.

```
let x,y,z = 2 * 3, 3 + 3, 2 + 2;
let avg_sales = (select avg( sale_amt) from sales where region
                 = "NW" );
let full_name = first_name || middle_initial || last_name;
```

첫 번째 지정 명령문은 표현식 2 * 3, 3 + 3 및 2 + 2의 값을 각각 변수 x, y 및 z에 지정합니다. 다음의 지정 명령문은 오른쪽 표현식(이 경우는 SQL문)에 의해 리턴된 값을 변수 avg_sales에 지정합니다. 그리고 마지막 SPL문은 문자열 연결을 수행하여 이름, 중간 이니셜 및 성을 연결한 전체 이름으로 문자열 변수를 작성합니다.

system 명령문

system문을 사용하여 외부 프로그램을 실행할 수 있습니다. 그러면 여기에서 실행될 수 있는 응용프로그램 및 운영 체제에 대한 제한된 액세스가 SPL에 제공됩니다. 외부 응용프로그램은 내장 프로시저를 호출한 사용자의 사용 권한 내에서 실행됩니다. 다음은 system문의 구문입니다.

```
system <external program and parameters>;
```

프로그램과 매개변수는 문자열(character/varchar) 변수나 문자열로 평가되는 표현식에서 큰따옴표로 묶은 문자열 상수로 표시될 수 있습니다. 다음의 system문은 로그 파일에 오류 메시지를 보내는 데 사용됩니다.

```
system "echo Error in SPL processing >> spl_error.log";
```

이 명령문은 Unix echo문의 재지정된 출력을 사용하여 외부 로그 파일로 오류 메시지를 기록합니다. 또는 Unix mail 프로그램을 호출하여 특정 사용자에게 전자 우편을 보낼 수 있습니다.

SPL system 명령문 리턴 값

system문을 사용하여 실행되는 프로그램에서 리턴되는 값은 적합한 SQL 오류와 함께 ISAM 오류로 발생된 영(0)이 아닌 종료 상태뿐이므로, 이는 외부 프로그램에서 값을 호출하고 리턴하는 데 유용한 메커니즘이라고 할 수 없습니다. 외부에서 실행되는 프로그램에서 내장 프로시저로 값을 리턴하려면 외부 프로그램은 데이터베이스 테이블에 데이터를 기록해야 하며 내장 프로시저는 이 테이블에서 데이터를 읽게 됩니다. SPL system문은 사용자에게 전자 우편 통지를 보내거나 외부 로그 파일에 기록하기 위해 외부 프로그램을 호출할 때 가장 적절하게 사용됩니다.

on exception 명령문

on exception문은 SPL 처리 중 발생하는 런타임 오류를 트랩하는 데 사용됩니다. 이 명령문을 사용하면 오류가 발생할 때 프로그램을 특정 프로시저로 분기할 수 있습니다. 다음은 on exception문의 구문입니다.

```
on exception
  | in (<error number list>)
  | < statement block >
  | set <sql error variable>, <isam error variable>, <error string>
end exception | with resume;
```

on exception문은 트랩되는 오류에 기초하여 명령문 블록을 실행하거나 SQL 및 ISAM 오류의 값과 발생된 오류를 설명하는 문자열을 사용하여 다른 변수를 설정할 수 있습니다.

on exception문은 내장 프로시저에서 변수 선언문 다음과 언어 명령문 앞에서 사용해야 합니다. on exception문은 명시적으로 실행되지 않으며, 단지 특정 오류가 발생한 경우 취해질 조치 과정만 정의합니다.

in절은 트랩할 특정 오류의 목록을 지정하는 데 사용됩니다. "in" 절이 생략되면 on exception문은 발생된 모든 오류에 대해 취해질 조치 과정을 지정합니다.

with resume절을 사용하면 on exception 블록의 명령문을 실행한 후 오류를 발생시킨 SPL 라인 다음에서 프로그램이 계속 실행되도록 지정합니다. 다음 프로시저는 on exception문의 사용을 보여줍니다.

```
create procedure sales_update()
define psales_id like sales_id;
define psaels_amt like sales_amt;
define error_flag integer;

let error_flag = 0;

on exception in ( -323 )
```

```
    let error_flag = error_flag +1;

end exception with resume;

foreach select sales_id, sales_amt
        into psales_id, psales_amt

        insert into sales_upd ( sales_id, sales_amt)
          values ( psales_id, psales_amt );
end foreach;
insert into control_file( "sales_upd", "errors", error_count );

end procedure;
```

이 SPL 분할에서 오류 -323에 대해 예외 트랩이 작성됩니다. 이 오류가 SQL 또는 프로시저에 의해 발생되면(raise exception문 사용) error_count 변수가 증분되고 이 오류 다음의 명령문에서 실행이 재개됩니다. 프로시저가 종료되기 전에 오류 수를 제어 파일에 작성하여 발생된 오류 수를 기록합니다.

define 명령문

define문은 내장 프로시저 내에서 변수를 선언하는 데 사용됩니다. 변수는 사실상 유효한 SQL 자료형이면 모두 사용 가능합니다. 변수는 테이블 컬럼처럼 선언할 수도 있으므로 변수를 테이블 컬럼에 직접 바인드할 수 있습니다. define문의 구문은 다음과 같습니다.

```
define [ global ] <variable name> <data type> | like
<table_name.column_name> [ default <default value> ]
```

변수는 기본적으로 로컬 범위로 정의됩니다. 선택적 global 키워드를 사용하여 전역 범위를 갖도록 변수를 선언할 수 있습니다. 기본값은 전역 범위의 변수에 대해서만 선언할 수 있습니다. 내장 프로시저에서 사용될 일련의 변수를 정의하려면 다음 명령문을 실행하십시오.

```
...
define counter_variable integer;
define psales_id like sales.sales_id;
define pregion like sales.region;

define x,y,z smallint;
...
```

이 SPL 조각에서 변수는 먼저 카운터 변수로 사용될 정수로 선언됩니다. 그 다음의 두 선언은 테이블 컬럼에 자료형으로 바인드될 변수를 선언합니다. 이들 변수는 like절을 사용하여 선언합니다. 프로시저가 작성되었을 때(프로시저가 p-코드로 컴파일되었을 때) 변수는 컬럼의 자료형으로 선언됩니다.

기본적으로 변수는 로컬 범위를 가진다는 점에 유의하십시오. 따라서, 프로시저 또는 이들이 선언되어 있는 begin/end 블록에서만 볼 수 있습니다. 변수가 전역 범위를 가지려면 다음 절에 설명된 대로 전역 변수로 선언되어야 합니다.

SPL에서 전역 범위 변수 사용

define문은 전역 변수 즉, 현재 세션 중 실행 중인 모든 프로시저에 표시될 수 있는 변수를 선언하는 데 사용할 수 있습니다. 전역 변수는 현재 데이터베이스 세션 중 현재 데이터베이스 인스턴스 내에서만 볼 수 있습니다. 다음의 내장 프로시저는 전역 SPL 변수의 사용을 보여줍니다.

```
create procedure demo_global1()
define global gvar1 char(20) default "this is global";
system "sleep 200";

end procedure;

create procedure demo_global2()
    returning char(20);

let gvar1 = "this is a new value";
```

```
return gvar1;

end procedure;
```

내장 프로시저 demo_global1을 실행하면 먼저 gvar1의 값을 "this is global"로 설정한 다음 system문을 실행하여 200초 동안 휴면하게 하십시오. 그러면 두 번째 프로시저가 실행되기에 충분한 정도로 프로시저를 일시정지할 수 있습니다. 그런 다음, 내장 프로시저 demo_global2를 실행하면 전역 변수에 액세스하여 gvar1의 값을 "this is a new value"로 설정하고 이 값을 프로시저를 호출한 프로세스로 리턴합니다.

exit 명령문

exit문은 제어 루프 다음의 명령문으로 무조건 분기하는 데 사용할 수 있습니다. exit문은 다음의 구문을 사용합니다.

```
exit while | for | foreach;
```

다음의 내장 프로시저는 분기 명령문을 사용하여 foreach 루프를 종료하는 것을 보여줍니다.

```
create procedure get_vals()

define psales_id like sales.sales_id;
define psales_code like sales.sales_code;

foreach select sales_id, sales_code
        into psales_id, psales_code
        from sales

        if psales_code not in ( "NW", "NE", "SW", "SE" )
          then
                exit foreach; - bad value

            else
                  insert into sales_holding values (
```

```
psales_id, psales_code );
            end if;

end foreach;


end procedure;
```

trace 명령문을 사용하여 디버그 결과 생성

Informix-SPL은 SPL문이 지정하는 파일에 써진 디버그 결과를 생성할 수 있는 기능을 제공합니다. 디버그 결과는 추적이라고 하며 SPL문을 사용하여 이를 설정하고 해제할 수 있습니다. 이들 명령문의 구문은 다음과 같습니다.

```
set debug file to <debug file name> | with append;
trace on | off | procedure | <trace output string> ;
```

set debug file문은 디버그 파일을 열거나 파일이 없는 경우 파일을 작성하는 데 사용됩니다. 이 명령문은 디버그 결과를 생성할 trace문을 실행하기 전에 실행해야 합니다. 기본적으로 디버그 파일이 삭제되고 새 결과가 쓰여집니다. with append절은 이 기본 동작을 무시하고 디버그 파일에 추가합니다.

프로시저에서 dbinfo 함수 사용

Informix 도구 세트는 내부 구조를 사용하여 오류와 경고를 추적하고 마지막 명령문이 실행될 때 정보를 저장합니다. 이 내부 구조를 sqlca 구조라고 하며, 이 내부 구조에는 현재 세션에 의해 삽입된 마지막 시리얼 값 및 마지막으로 실행된 update 또는 delete 명령문에 의해 업데이트된 행 수와 같은 중요한 정보가 포함되고 있습니다. 원래의 SPL 버전은 sqlca 구조를 묻는 기능을 제공하지 않았습니다. 다행히도 그 후에 Informix-SQL에서 사용할 수 있는 함수 목록에 dbinfo 함수가 추가되었습니다. 이 함수는 sqlca 구조에 대한 정보와 함께 내장 프로시저에 매우 유용한 기타 중요한 정보를

제공합니다.

특히 dbinfo 함수는 다음과 같이 몇 가지 유용한 정보를 리턴합니다.

- serial 컬럼에 대한 insert에 의해 생성된 마지막 값
- 특정 SQL문에 의해 처리된 행 수
- 현재 엔진 세션의 세션 ID
- DBSPACE 또는 TABLESPACE 번호나 표현식에 대한 DBSPACE 또는 TABLESPACE의 이름

dbinfo 함수의 구문은 다음과 같습니다.

```
dbinfo ( <parameter 1> [, <parameter 2>] )
```

dbinfo 함수는 단일 값을 리턴합니다. 함수로 전달된 첫 번째 매개변수에 따라 하나 또는 두 개의 매개변수를 승인합니다. 전달된 매개변수는 표 7-9에 나와 있습니다.

*표 7-9 **dbinfo** 함수 매개변수*

parameter1	parameter2	제공되는 정보
dbspace	tblspace number	TABLESPACE 번호
sqlca.sqlerrd1		삽입된 마지막 시리얼 값
sqlca.sqlerrd2		업데이트된 행 수
sessionid		현재 세션 ID

dbspace 옵션은 TABLESPACE가 있는 DBSPACE의 이름을 나타냅니다. TABLESPACE 번호는 정수이거나, 기존의 TABLESPACE에 해당하는 정수 표현식입니다.

sqlca 옵션은 Informix 개발 도구에서 사용되는 구조인 sqlca 정보 구조의 구성 요소를 참조하여 SQL 작업과 관련된 정보를 제공하고 오류를 추적합니다. sqlca.sqlerrd1 옵션을 사용하여 dbinfo 함수를 실행하면 삽입 작업으로 생성된 마지막 시리얼 값에 해당하는 정수가 시리얼 컬럼으로 리턴됩니다. sqlca.sqlerrd2 옵션을 사용하여 dbinfo 함수를 실행하면 select, insert, update, delete 또는 execute 프로시저 명령문으로 처리된 행 수를 나타내는

정수가 리턴됩니다. 다음 SPL 조각은 dbinfo 함수의 사용을 보여줍니다.

```
...
    insert into sales_man( 0, psale_name, psales_address );
  let psales_id = dbinfo( 'sqlca.sqlerrd1' );
  insert into sales_region_assgnmt values( psales_id,
psales_region1);
  insert into sales_region_assgnmt values ( psales_id,
psales_region2);
...
```

이 코드 조각은 정수 값으로 저장된 테이블에 대해 자동으로 유지보수되는 고유 식별자인 serial 필드를 포함하는 sales_man 레코드를 작성합니다. 이 시리얼 ID는 판매원을 고유하게 식별하는 데 사용됩니다. 이 ID가 바로 판매원 ID(sales_id)입니다. 이 sales_id는 판매원에 대한 지역 지정을 정의하는 sales_region_assgnmt 테이블에서 외부 키로 사용됩니다. dbinfo 함수를 사용하여 새로 삽입된 판매원 레코드의 시리얼 ID(sales_id)가 검색되고 프로그램 변수에 저장된 다음, 이 ID가 sales_man 테이블에 대한 외부 키의 기능을 수행할 sales_region_assgnmt 테이블에 기록됩니다.

IDS 데이터베이스 트리거

IDS 엔진은 트리거, 이벤트 및 데이터베이스 업데이트로 시작된 해당 조치를 지원합니다. 데이터베이스 업데이트는 SQL insert, update 및 delete 명령문입니다.

데이터베이스 트리거는 데이터베이스 무결성을 향상시키는 훌륭한 수단입니다. 비즈니스 규칙을 시행하고, 데이터 유효성 검증을 수행하며, 데이터를 복제하고, 시스템 활동을 로그하는 데 사용할 수 있습니다. IDS 트리거는 실행된 트리거가 순서에 따라 조치 이벤트를 수행하는 연속 트리거가 될 수 있으며, 트리거 조치가 발생하기 전에 데이터의 스냅샷을 사용할 수 있습니다.

create trigger 명령문

데이터베이스 트리거는 create trigger문을 사용하여 작성됩니다. IDS 트리거의 구문은 다음과 같습니다.

```
create trigger <trigger name> insert on <table name> |
                                delete on <table name> |
                                update on <table name>
                   [referencing <referencing clause> ]
                   <action statements>
```

referencing절을 사용하면 트리거 조치를 수행하기 전에 트리거링 테이블의 데이터 레코드를 참조할 수 있습니다. 이 절은 테이블의 성격이 트리거 유형에 따라 다르기 때문에 필요합니다(insert 작업 이전의 레코드는 없음).

action문은 트리거 본문 내에서 수행될 조치 목록을 제공합니다. 조건절(when 키워드 사용) 이외에는 특정 SQL문만 이 명령문 블록에 사용할 수 있습니다. 다행히도 트리거가 action문 블록의 내장 프로시저 명령문을 실행할 수 있도록 함으로써 전체 SPL 구문을 트리거에 대해 사용할 수 있습니다.

insert 또는 delete문마다 하나의 트리거만 사용할 수 있습니다. 특정 조건이 충족되면 하나의 update문에 대해 여러 트리거를 정의할 수 있습니다.

트리거되는 이벤트에 대해 더 이상 단일 SQL문이 실행되지 않고 대신 여러 명령문이 실행되고 있으므로, 트리거는 성능에 영향을 미칠 수 있습니다. SQL문 트리거링 성능은 단일 SQL문이 아닌 트리거에 의해 실행되는 전체 SQL문 세트와 관련하여 평가되어야 합니다.

트리거의 구문과 제한사항은 실행되고 있는 트리거의 유형에 따라 다릅니다. 다음 절에서는 특정 트리거 유형과 해당 구문을 자세하게 설명합니다.

삽입 트리거

삽입 트리거는 삽입 이벤트(SQL insert문)를 트랩하며 해당 이벤트에 대해 조치를 지정할 수 있도록 합니다. 삽입 트리거의 구문은 다음과 같습니다.

```
create trigger insert on <trigger name>
[referencing <referencing clause> ]
[before] | [for each row] | [after]
                          <action statements>
```

이 트리거 이름은 트리거가 작성되는 데이터베이스에 대해 고유한 이름이어야 합니다. 선택적 referencing절은 새로 삽입된 행과 관련시킬 이름(테이블 이름과 구별된 이름)을 지정하는 데 사용됩니다. 이 절의 구문은 다음과 같습니다.

```
...
        referencing new as <correlation name>
...
```

지정된 상호 관련 이름은 조치 명령문의 for each row절 내에서 사용할 수 있습니다(트리거 이벤트 이전에는 이러한 행이 존재하지 않으므로 논리적으로 before절에는 사용할 수 없음). 이 절은 다음과 같이 사용할 수 있습니다.

```
...
    referencing new as new_one
        for each row ( insert into sales_log values
(new_one.sales_id, new_one.region, new_one.sales_date) );
```

트리거에 대한 조치 명령문은 트리거링 이벤트에 대해 실행할 조치를 지정하는 데 사용됩니다. 조치는 insert, update 또는 delete 명령문이 될 수 있습니다. 삽입 트리거는 트리거링 테이블을 참조하는 insert 또는 delete 명령문이 있는 action절을 가질 수 없습니다. 트리거 조치 명령문의 구문은 다음과 같습니다.

```
...
before (<action statements>)
  for each row ( <action statements> )
    after ( <action statements> );
```

before 조치 명령문은 행이 삽입되기 전에 한 번 실행됩니다. for each row 조치 명령문은 삽입된 각 행에 대해 한 번 실행됩니다. after 조치 명령문은 행이 삽입된 후에 한 번 실행됩니다. 다음은 삽입

트리거를 사용하는 조치의 예를 보여줍니다.

```
create trigger ins_sales
    insert on sales
    referencing new as post
    for each row ( insert into sales_log values (
post.sales_id, post.sales_date, post.region, post.sales_code ) );
```

표시된 트리거는 sales 테이블에 대한 삽입을 트랩하고 for each row문을 사용하여 sales_log 테이블에 sales 레코드를 삽입합니다. sales 테이블에 행이 삽입될 때 삽입 트리거가 실행하여 sales_id, sales_date, region 및 sales_code 컬럼들을 모두 하나의 행으로 sales_log 테이블에 삽입합니다. 그러면 특정 시간에 sales 테이블에 작성된 엔트리의 실행 로그가 작성됩니다.

삭제 트리거

삭제 트리거를 사용하면 테이블에서의 레코드 삭제 이벤트를 트랩할 수 있습니다. 따라서 일정한 용도로 필요하거나 삭제 활동 기록(레코드를 삭제한 사람과 시기)을 위해 필요한 대상 테이블에서 행이 삭제되기 전에 그 행을 저장하는 유용한 수단을 제공합니다. delete 트리거 명령문의 구문은 다음과 같습니다.

```
create trigger delete on <trigger name>
[referencing <referencing clause> ]
[before] | [for each row] | [after]
                            <action statements>
```

논리적으로 삭제 트리거는 새로운 행을 참조할 수 없습니다. 테이블에서 행을 제거하는 것이 delete문의 목적이기 때문입니다. 따라서 referencing절은 다음과 같이 이전 행만 참조할 수 있습니다.

```
create trigger
  ...
    referencing old as <correlation name>
  ...
```

삭제 트리거에 referencing절이 포함된 경우 트리거 본문에 for each row절이 포함되어야 합니다. 삭제 트리거에 대한 유효한 조치 절은 다음과 같습니다.

```
create trigger
  ...
    before <action statements> |
    for each row <action statements> |
    after <action statements>
```

이러한 조치 명령문 블록은 다른 트리거의 경우와 동일하며, 삭제되는 각 행에 대해 삭제 전과 삭제 후에 조치를 실행할 수 있도록 합니다. 삭제 트리거에 referencing절이 포함된 경우 for each row 조치 블록이 포함되어야 합니다. 다음은 삭제 트리거의 예를 보여줍니다.

```
create trigger del_sales
    delete on sales
    referencing old as pre
    for each row ( insert into sales_delete_log
            values ( pre.sales_id, pre.sales_date,
                     pre.region,
                     pre.sales_code ) );
...
```

표시된 삭제 트리거에는 이전 행(삭제할 행)의 상호 관련 이름을 pre로 지정하는 referencing문이 있습니다. 이 트리거에 대한 for each row문은 삭제되는 각 행에 대해 실행됩니다. 행이 삭제될 때 이전 레코드의 일부가 sales_delete_log에 기록됩니다.

이는 복구 테이블에 복구 레코드가 될 수 있는 것을 기록하는 것 외에도, 판매 레코드를 삭제하고 있는 사람 및 레코드가 삭제되는 시간에 대한 레코드를 작성하기 위해, 로그 레코드를 테이블에 기록하는 데 유용할 수 있습니다. 이 로그 레코드는 삭제되는 모든 행에 대해서가 아니고 삭제 작업마다 한 번만 기록되므로 다음의 트리거에서와 같이 트리거 이벤트에 대해 한 번만 실행되는 before문 블록에 있어야 합니다.

```
create trigger del_sales
    delete on sales
    referencing old as pre
    before ( insert into sales_log values
           ( sales_id, "DELETE", user, today ) )
    for each row ( insert into sales_delete_log
            values ( pre.sales_id, pre.sales_date,
                     pre.region,
                     pre.sales_code ) );
```

insert문은 before문 블록 내에서 실행되어 판매 로그 테이블에 레코드를 삽입합니다. 이 레코드에는 삭제 작업을 실행한 사람에 대한 정보 즉, 판매 ID와 삭제 작업이 발생했음을 나타내는 문자열, 키워드 "user" 및 "today"에 의해 제공된 사용자 이름 및 현재 날짜가 있습니다.

업데이트 트리거

업데이트 트리거는 테이블에 대해 update문이 실행될 때 실행됩니다. update문은 delete문처럼 여러 행에 영향을 미칠 수 있습니다. update문의 영향을 받은 각 행에 대해 for each row 조치 명령문을 실행할 수 있습니다. update문은 유효한 before 레코드(업데이트가 발생하기 전)와 after 레코드(업데이트가 발생한 후)도 포함할 수 있습니다. referencing절은 이들 두 레코드의 상호 관련 이름을 설정하는 데 사용할 수 있으며, 이에 대한 구문은 다음과 같습니다.

```
referencing old as <correlation name> | new as <correlation name>
```

이 referencing절을 사용하면 트리거의 조치 명령문에서 이전(업데이트 전) 레코드와 새(업데이트 후) 레코드를 모두 참조할 수 있습니다.

삽입 및 업데이트 트리거와 마찬가지로 다음의 구문을 사용하여 before, for each row 및 after 트리거 이벤트에 대해 조치 명령문을 실행할 수 있습니다.

```
create trigger
    ...
      before <action statements> |
      for each row <action statements> |
      after <action statements>
```

before 및 after 명령문 블록은 트리거 이벤트 중 즉, 업데이트가 발생하기 전과 이 업데이트 작업이 명령문에 지정된 행을 업데이트한 후에 한 번만 실행됩니다. for each row문은 트리거에 의해 업데이트된 모든 행에 대해 실행됩니다. 업데이트 트리거에 referencing절이 포함된 경우 for each row절도 포함되어야 합니다.

삽입 및 삭제 트리거와 마찬가지로 업데이트 트리거는 테이블에 대한 변경사항을 추적하는 유용한 방법입니다. 사용자가 레코드를 변경하면 이벤트가 로그될 수 있으며 다음의 트리거에 표시된 것과 같이 레코드의 사전 이미지가 복구 테이블에 기록될 수 있습니다.

```
create trigger upd_sales
    update on sales
    referencing old as pre new as post before ( insert into
sales_log values
            ( sales_id, "UPDATE", user, today ) )
    for each row ( insert into sales_upd_log
            values ( pre.sales_id, pre.sales_date,
                     pre.region,
                     pre.sales_amt),
            insert into sales_upd_recs
              values ( post.sales_id,
                       pre.sales_date,
                       pre.sales_amt) )
```

여기에 표시된 트리거는 sales 테이블에 대해 update가 발생할 때 실행됩니다. referencing문은 sales 테이블이 업데이트되기 전의 사전 레코드와 업데이트 후에 존재하는 레코드인 사후 레코드를 모두 참조합니다. before절은 sales 레코드가 업데이트되고 있음을 나타내는 로그 레코드와 업데이트를 작성한 사용자 및 업데이트의 날짜를 저장하는 데 사용됩니다.

for each row문 블록에서는 두 개의 insert문이 실행됩니다. 첫 번째 insert문은 이전(업데이트 전) 버전의 판매 레코드를 삽입하여 sales_upd_log 테이블에 레코드를 삽입함으로써 업데이트 작업에 대해 보다 완전한 로그 레코드를 작성합니다. 다음 insert문은 post(업데이트 후) 레코드의 일부분을 sales_upd_recs 테이블에 삽입합니다. 레코드의 사전 및 사후 이미지를 함께 사용하면 sales 테이블에 대해 작성되는 업데이트의 성격에 유용한 점을 제공할 수 있습니다.

트리거의 유용성

앞에 요약된 기능 외에도 트리거는 내장 프로시저를 호출하고, 값을 리턴하며, 조건절을 사용하여 표현식을 평가하고, 표현식의 평가에 기초하여 조치를 수행하는 능력도 제공합니다. 이러한 기능을 통해 로깅, 복제, 사용자 통지 및 기타 기능에 트리거를 사용할 수 있습니다. 다음 절에서는 트리거를 사용하여 이러한 기능을 수행하는 예를 제공합니다.

트리거 내에서 내장 프로시저 실행

트리거는 프로그래밍 목적으로 제한된 구문만 제공합니다. IDS 내장 프로시저 언어는 훨씬 더 확장된 프로그래밍 명령문과 기능들을 제공합니다. 다행히 이들 두 기능은 다음 명령문에서와 같이 트리거 내에서 내장 프로시저를 호출하는 능력을 제공함으로써 결합되었습니다.

```
create trigger del_sales
    delete on sales
    referencing old as pre
for each row ( execute procedure log_sales( pre.sales_id ) );
```

이 명령문은 sales 레코드가 삭제될 때 내장 프로시저를 실행하는 트리거를 작성합니다. 삭제되는 각 행에 대해 내장 프로시저 log_sales가 호출되며 삭제되는 레코드의 sales_id가 전달됩니다. log_sales 프로시저는 삭제를 실행하는 사용자의 이름 및 삭제 날짜와 같은 삭제에 관한 정보를 추가하여 판매 로그 테이블에 레코드를 기

록하는 코드를 포함할 수 있습니다. 이 프로시저에는 일련의 비즈니스 규칙이 충족되지 않은 경우 삭제를 중단할 수 있는 무결성 검사 기능도 추가할 수 있습니다. 내장 프로시저를 사용하면 삭제가 발생하기 전에 보다 복잡한 논리를 실행할 수 있습니다.

업데이트 트리거는 into절을 사용하여 값을 리턴하는 내장 프로시저를 호출할 수 있습니다. 이러한 값은 트리거링 테이블에 대한 컬럼으로만 리턴될 수 있습니다. 그리고 into절이 있는 내장 프로시저는 트리거의 for each row 블록 내에서만 실행할 수 있습니다. 다음은 이러한 트리거의 예입니다.

```
create trigger upd_sales
    update of sales_amount on sales
    referencing old as pre new as post for each row (
execute procedure count_item( pre.item_id ) into
curr_in_stock);
```

이 프로시저는 sales 테이블을 업데이트하는 동안 실행됩니다. 판매 레코드가 업데이트될 때 이 트리거는 내장 프로시저를 실행하여 업데이트되는 품목에 대한 재고 품목의 계수를 가져옵니다. 이 계수는 업데이트되고 있는 레코드의 curr_in_stock 컬럼에 저장됩니다. 이는 결과적으로 계산된 필드의 값을 데이터베이스에 삽입되는 데이터로 계산합니다.

트리거에서 when절 사용

when절은 트리거에서 조건 테스트를 수행하는 데 사용할 수 있습니다. when절은 트리거 작성 시 존재하는 테이블, 컬럼 및 내장 프로시저를 사용하여 다른 SQL 조건 표현식과 마찬가지로 조건 테스트를 수행할 수 있습니다. 조건 표현식이 TRUE로 해석되면 조치 명령문이 실행됩니다. 조건문이 FALSE 또는 알 수 없음으로 해석되면 조치 명령문은 실행되지 않습니다. 다음은 트리거 명령문 when절의 예를 나타냅니다.

```
create trigger upd_sales
      update on sales
      referencing old as pre new as post
    for each row when ( pre.sales_amt > 10000 )
```

```
          ( insert into sales_log values ( pre.sales_id,
 pre.sales_amt, today ) );
```

이 트리거는 sales_amt 컬럼에 대해 조건 테스트를 수행하며, sales_amt가 10,000 달러를 초과하면 sales_log 테이블에 레코드를 기록합니다. 필요하면 둘 이상의 when절을 사용하여 트리거에서 데이터를 평가할 수 있습니다.

사용자 정의 복제에 트리거 및 내장 프로시저 사용

트리거의 유용한 기능은 정교하게 사용자 정의된 복제를 수행하는 것입니다. 삽입 트리거를 사용하여 전체 레코드, 특정 컬럼 또는 컬럼에 대해 실행된 표현식 값을 현재 데이터베이스의 테이블이나 원격 데이터베이스의 테이블로 복제할 수 있습니다. 또한 트리거 명령문의 조건문을 사용하여 데이터를 복제할 위치를 결정할 수 있습니다. 다음 예는 트리거를 사용하는 데이터 복제의 예를 보여줍니다.

```
create trigger ins_sales
      insert on sales
      referencing new as new
      for each row (
        insert into region11@corporate_hq:sales_replicate
                              values ( new.sales_date,
                                       new.region,
                                       new.sales_code,
                                       new.sales_amt );
```

이 삽입 트리거에서는 행이 로컬 데이터베이스의 테이블로 삽입되고, 삽입되는 각 행에 대해 이 행의 데이터가 원격 머신(region11)과 sales_replicate라고 하는 테이블로 이동됩니다. 이 트리거는 현재 삽입되고 있는 행을 복사한 다음 이 행을 다른 머신으로 복제하는 효과를 가집니다.

표시된 복제 스키마의 경우, 트리거를 성공적으로 완료하려면 로컬 및 원격 머신이 모두 온라인 상태가 되어야 합니다. 어떠한 이유로 인해 원격 머신이 오프라인되면, 원격 테이블에 대한 쓰기 및 전체 삽입 작업을 실패하게 됩니다. 이는 로컬 판매 테이블에 대해 삽

입을 성공하려면 로컬 및 원격 머신을 모두 온라인 상태로 만들어야 함을 의미합니다. 이러한 형태의 사용자 정의 복제 대신, 복제할 행 또는 행에 대한 키를 로컬 테이블에 저장한 후 정기적으로 이 로컬 테이블을 원격 테이블로 플러시할 수 있습니다. 이러한 복제 방법은 다음의 예에 나와 있습니다.

먼저 삽입될 레코드를 기록하도록 테이블을 작성합니다. 복제를 수행하는 데 필요한 시스템의 노력을 줄이기 위해 레코드의 기본 키만 이 테이블에 씁니다. 이 테이블에는 다음과 같이 키 필드를 저장한 다음 소스 테이블 및 대상 테이블의 이름을 저장할 수 있는 몇 개의 컬럼이 있는 레코드가 들어 있습니다.

```
create table replicate (
      key1 char(10),
      key2 char(10),
      key3 char(10),
      key4 char(10),
      stable_name char(18)
      rflag char(1)
);
```

삽입 트리거가 복제될 테이블에 대해 작성됩니다. 이 트리거는 다음과 같이 삽입되고 있는 레코드에 대해 replicate 테이블에 레코드를 기록합니다.

```
create trigger ins_sales
      insert on sales
      referencing new as new
      for each row (
        insert into replicate (key1, rflag, stable )
                    values ( new.sales_id, "N", "sales" ) );
```

이 트리거는 sales 테이블의 기본 키를 복제 테이블에 삽입하고 레코드가 "새(new)" 레코드이며 복제되지 않았음을 표시하는 값 "N"의 플래그를 삽입합니다. 그러면 내장 프로시저는 다음과 같이 replicate 테이블에서 레코드를 읽어 이 sales 레코드를 원격 테이블에 복제합니다.

```
create procedure replicate_sales()
define pkey char(10);

foreach select key1
        into pkey
        from replicate
        where stable = "sales" and
                rflag = "N"

        insert into region11@corporate_hq:sales_replicate
            select * from sales
            where sales_id = pkey;

        update replicate
          set rflag = "U"
          where key1 = pkey;

end foreach;

end procedure;
```

이 내장 프로시저는 판매 테이블에 대해 삽입 트리거가 기록한 replicate 테이블에서 레코드를 읽습니다. stable(소스 테이블) 필드에 대해 필터하여 특정 레코드를 검색할 수 있으며 rflag 필드는 "N"(복제되지 않음)으로 설정됩니다. 이 예에서는 하나의 기본 키 컬럼을 검색하지만 여러 부분으로 이루어진 기본 키도 replicate 테이블에 저장하여 레코드 복제 시 검색할 수 있습니다.

기본 키는 SPL 변수에 저장되며, 이 변수는 소스 테이블(판매 테이블)에서 레코드를 검색하고 복제된 하나의 레코드를 region11@corporate_hq:sales_replicate 테이블로 삽입하는 데 사용됩니다. 레코드가 원격 테이블에 기록되면 현재 레코드에 대한 replicate 레코드가 업데이트되고 rflag 컬럼은 "R"로 설정되어 복제되었음을 표시합니다.

업데이트되는 replicate 테이블 레코드는 내장 프로시저의 제어 루프에서 사용되기 때문에 이 경우, replicate 테이블 업데이트가 제한될 수 있는데 이는 사용 중인 격리 모드에 따라 크게 좌우됩니다. 효과적인 대안은 foreach 루프에 유지 커서를 사용한 다음, update문에 where current of절을 사용하여 복제 테이블을 업데이트

하는 것입니다.

이 복제 스키마를 좀더 복잡하고 강력하게 구현하여 트랜잭션 내에서 레코드를 복제하는 프로세스를 수행하고 높은 수준의 데이터 무결성 및 제어를 제공할 수 있습니다. 이 복제 스키마는 복제에 대해 저장 및 진행 접근법을 사용하기 때문에 레코드의 실제 복제가 일어날 때 어느 정도 유연성이 있다는 점을 유의하십시오. 1분마다 또는 하루에 한 번씩 replicate 테이블을 읽고 레코드를 복제할 수 있습니다. 이것은 전적으로 복제가 수행되고 있는 응용프로그램의 필요에 따라 좌우됩니다.

IDS 문자열 처리 함수

IDS 버전 7.3에서는 오랫동안 기다려 왔던 문자열 처리 함수 세트가 IDS 엔진에 추가되었습니다. 이러한 함수를 통해 변수 매개변수를 사용하여 서브스트링 작업을 수행할 수 있게 되었습니다(SQL 구문과 달리 정수 상수 매개변수가 필요함). 문자열 함수는 문자열을 정리하고 문자열에서 문자를 바꾸고 문자로 문자열을 채우는 데에도 사용할 수 있습니다. 표 7-10은 IDS의 7.3 버전에서 사용할 수 있는 문자열 함수를 보여줍니다.

다음 절에서는 이들 각 함수를 자세하게 설명하고 각 사용의 예를 제시합니다.

표 7-10 새 IDS 문자열 함수

함수	설명
trim	문자열에서 트레일링 또는 리딩 공백을 정리합니다.
substring	입력 매개변수에 기초하여 문자열에서 서브스트링을 작성합니다.
substr	입력 매개변수에 기초하여 문자열에서 서브스트링을 작성합니다.
replace	문자열에서 지정된 문자를 바꿉니다.
lpad	문자열의 왼쪽부터 문자를 채웁니다.
rpad	문자열의 오른쪽부터 문자를 채웁니다.
upper	문자열을 대문자로 변환합니다.
lower	문자열을 소문자로 변환합니다.
initcap	문자열에서 단어의 첫 문자를 대문자로 변환합니다.

trim 함수

trim 함수는 함수로 전달된 첫 번째 매개변수에 따라 문자열 표현식에서 리딩 문자, 트레일링 문자 또는 둘 다를 정리합니다. 정리할 문자로 하나의 문자가 지정되며 기본값은 공백 문자입니다. 첫 번째 매개변수가 지정되면 앞과 뒤의 정리 작업이 모두 수행됩니다. 다음은 trim 함수의 구문입니다.

```
trim ( leading/trailing/both <single character to trim) from
<string to trim>) )
```

다음의 SQL문을 사용하여 문자열의 끝에서 의문 부호를 정리할 수 있습니다.

```
select sales_id, trim( trailing "?" from sales_code ),
sales_amt
from sales;
```

이 SQL문은 sales 테이블에서 뒤에 "?" 트레일링 문자를 사용할 수 있는 sales_code를 포함하여 여러 개의 컬럼을 검색합니다. 이들 문자는 표시된 구문을 사용하여 컬럼 문자열 값의 끝에서 정리됩니다.

substring 함수

substring 함수는 함수로 전달된 매개변수에 기초하여 함수로 전달된 문자열의 서브스트링으로 해석됩니다. 시작 위치와 종료 위치가 모두 함수로 전달됩니다. 종료 위치를 제거하면 이 함수는 전달되는 문자열의 끝까지 문자열을 리턴합니다. 문자열의 끝에서부터 역으로 계수하기 위해 음수 매개변수를 사용할 수 있습니다. 다음은 substring 함수의 구문입니다.

```
substring( <string expression> from <starting position> to
<ending position> )
```

컬럼에 저장된 문자열의 서브스트링을 가져오려면 다음 구문을 사용하십시오.

```
select sales_id, substring( region from 1 to 3 ), sales_amt
from sales;
```

이 쿼리는 sales 테이블에서 sales_id, 지역의 처음 세 문자 및 판매 금액을 검색합니다. 어떤 경우에는 컬럼 문자열의 특정 문자 위치에 특정 의미를 가질 수 있습니다. substring 함수는 이 정보를 추출하는 유용한 수단입니다.

substr 함수

substr 함수는 substring 함수와 동일한 기능을 수행하지만 구문이 다릅니다. 소스 문자열이 서브스트링의 시작 위치 및 종료 위치에 대한 매개변수와 함께 지정됩니다. 다음은 substr 함수의 구문입니다.

```
substr( source, start, length)
```

다음은 이전 substring 함수 쿼리와 동일한 기능을 수행하지만 대신 substr 함수를 사용하는 쿼리입니다.

```
select sales_id, substr( region, 1, 3 ), sales_amt
from sales;
```

replace 함수

replace 함수는 문자열의 문자를 바꾸는 데 사용됩니다. 다음은 replace 함수의 구문입니다.

```
replace( string_to_process, string_to_replace [,
replacement_string] )
```

세 개의 매개변수를 모두 사용하여 replace 함수를 호출하면, string_to_replace는 replacement_string 매개변수의 문자열 값으로

바꿉니다. 선택적 replacement_string 매개변수가 함수 인수에 포함되지 않으면 소스 문자열 내의 string_to_replace 문자열이 삭제됩니다. 다음은 replace 함수의 한 예입니다.

```
select sales_id, replace( region, "SW", "NW" )
from sales;
```

이 SQL문은 replace 함수를 사용하여 region 컬럼 내에 있는 "SW" 문자열을 모두 "NW"로 바꾼 다음 이 값을 리턴합니다.

lpad 함수

lpad 함수는 문자열의 왼쪽을 채우는 데 사용됩니다. 즉, 문자열의 왼쪽에 공백을 추가하는 것으로, 많은 키 필드의 공통적인 요구사항이기도 합니다. 이 함수의 구문은 다음과 같습니다.

```
lpad ( source_string, length [,pad_string] )
```

pad_string 매개변수를 제거하면 문자열을 공백으로 채웁니다. 컬럼 값을 테이블에 삽입하기 전에 컬럼 값의 왼쪽에 네 개의 공백을 채우려면 다음의 구문을 사용하십시오.

```
insert into sales ( sales_id, sales_code ) select sales_id,
lpad( sales_code, 4) from sales_db@remote_machine:sales;
```

rpad 함수

rpad 함수는 lpad 함수와 동일한 기능을 수행하지만 대신 오른쪽을 채웁니다. 문자열 표현식의 오른쪽에 지정된 수의 공백을 채웁니다. 이 함수의 구문은 다음과 같습니다.

```
rpad( source, length [, pad string ] )
```

대소문자 변환 함수

여러 가지 대소문자 변환 함수를 문자열에 적용할 수 있습니다. 이러한 기능은 데이터 삽입 및 검색 시에 모두 문자열을 대문자나 소문자로 전환하여 대소문자를 구분하지 않는 쿼리를 효과적으로 수행할 수 있게 합니다. 이들 함수는 표 7-11에 나와 있습니다.

표 7-11 대소문자 변환 함수

upper	문자열의 각 문자를 대문자로 전환합니다.
initcap	문자열에서 각 단어의 첫 문자를 대문자로 전환합니다.
lower	문자열의 각 문자가 소문자로 전환합니다.

문자열을 테이블에 삽입하기 전에 대문자로 전환하려면 다음 구문을 사용하십시오.

```
insert into sales_tmp (sales_id, sales_code)
select sales_id, upper(sales_code) from sales;
```

검색하기 전에 컬럼의 첫 문자를 대문자로 설정하려면 다음 구문을 사용하십시오.

```
select sales_id, initcap( salesman_name ) from sales;
```

새로운 버전 7.3 기능

IDS의 7.3 릴리스는 몇 가지 중요한 새로운 기능을 엔진에 추가했습니다. 이러한 기능을 통해 첫 번째 페이지 행을 신속히 리턴하고, 강제로 테이블을 메모리에 상주하게 하며, “case”문을 사용하여 쿼리에서 결정 논리를 실행하고, 옵티마이저가 쿼리 계획을 세우는 데 사용할 지시문을 제공할 수 있습니다. 이러한 기능들은 비록 수는 적지만 엔진에 막대한 유연성을 더해 줍니다. 다음 절에서는 이러한 새로운 기능 중 일부를 사용하는 것에 대해 자세히 설명합니다.

case 명령문

case문은 SQL 표현식으로 사용되며, 일련의 조건을 테스트하고 이들 조건 테스트에 기초하여 값을 리턴할 수 있도록 합니다. case 문은 select 목록이나 select문의 where절과 같이 모든 유효한 컬럼 표현식에서 사용할 수 있습니다. 다음은 case문의 구문입니다.

```
case [value]
    when <condition>
      then
          <expression>
        ...
      else
          <value>
end
```

case 표현식을 사용하여 select문에서 조건 값 대체를 수행할 수 있으며, 이는 SQL 쿼리의 능력을 크게 확장하는 유용한 기능입니다. 다음은 case문의 예입니다.

```
select sales_id, case region
                        when "SW"
                            then "South West"
                        when "NW"
                            then "North West"
                        when "SE"
                            then "South East"
                        else
                            "Unknown Region"
                      end,
            sales_amt,
            sales_date
from sales
```

이 SQL문은 sales 테이블에서 값을 검색하고 region 컬럼을 평가합니다. 관계형 데이터베이스에서는 흔히 region 컬럼에 지역 이름의 전체 텍스트를 사용하지 않고 대신 지역 이름과 관련된 세 문자

코드를 사용합니다. case문은 region 테이블에 조인하지 않고도 region 코드를 평가하고 지역 이름을 리턴하는 데 사용됩니다. 다른 조건 표현식이 TRUE로 평가되지 않으면 else 조건이 사용됩니다. else 조건이 제거되고 모든 조건 표현식이 FALSE를 리턴하면 case 표현식은 NULL 값을 리턴하게 됩니다.

관계형 데이터베이스의 공통적인 문제는 NULL 값에 대한 것입니다. NULL은 데이터가 없음을 의미하므로 수학적인 값이 없습니다. 수학적 표현식에서 컬럼이 사용되는 경우에는 종종 다음과 같이 NULL 값을 유효한 수로 변환할 필요가 있습니다.

```
select region, avg( case sales_amt
                         when NULL
                           then 0
                    end )
from sales
group by region
```

이 select문은 sales 테이블에서 데이터를 검색하고 sales_amt 컬럼에 존재할 수 있는 NULL 값을 영(0) 값으로 변환합니다. avg 함수는 지역의 평균 판매 금액을 계산하는 데 사용되지만, 이 avg 함수는 평균 계산에 NULL 값을 포함시키지 않습니다. 따라서 컬럼의 정확한 평균 계수를 구하려면 NULL 값을 영(0)으로 변환하는 것이 중요합니다.

decode 함수

decode 함수의 유용성은 case문의 경우와 비슷합니다. 조건을 평가하고 해당 조건에 기초하여 결과를 리턴합니다. 다음은 decode 함수의 구문입니다.

```
decode ( <expression>, <when expression>, <result expression> …)
```

이 명령문의 기능은 다음 예에서 볼 수 있듯이 case문의 경우와 비슷합니다.

```
select sales_id, decode( region,
                              "SW", "South West",
```

```
                                "NW", "North West",
                                "NE", "North East",
                                "SE", "South East",
                                "Unknown Region" ),
        sales_amt
from sales
```

이 select문에서 region 코드는 지역 이름으로 평가되고 지역 컬럼의 값에 기초하여 이 컬럼 값에 대한 지역 이름이 리턴됩니다. 평가할 일련의 표현식에 대해 when expression/result expression 쌍이 함수로 전달됩니다. 함수 인수 목록에 나타나는 마지막 값은 이전의 모든 조건문이 실패할 경우 사용되는 else 표현식 값입니다. 마지막 else 표현식이 함수로 전달되지 않으면 NULL 값이 리턴됩니다.

nvl 함수

nvl 함수는 NULL 값을 관리하기 위한 대체 수단입니다. nvl 함수는 한 쌍의 표현식 즉, expression1과 expression2를 평가합니다. expression1이 NULL인 경우 expression2가 리턴됩니다. 다음은 nvl 함수의 구문입니다.

```
nvl( expression1, expression2 )
```

sales 테이블의 sales_amt 컬럼에 있는 NULL 값을 관리하기 위해 다음의 표현식을 사용하십시오.

```
select region, avg ( nvl( sales_amt, 0 ) )
from sales
group by region
```

이 select문은 sales 테이블에서 데이터를 검색하고 각 지역의 평균 판매 금액을 생성합니다. avg 함수에서 원하지 않는 NULL 값의 결과를 제거하려면 nvl 함수를 호출하여 NULL 값이 avl 함수에 의해 평가되기 전에 모든 NULL 값을 영(0)으로 변환하십시오. (이 표현식은 이전 예의 case문과 동일한 목적을 수행합니다.)

메모리 상주 테이블 및 인덱스 작성

모든 IDS 데이터 페이지는 처리되기 전에 먼저 공유 메모리 버퍼로 읽혀집니다. 그러나 이러한 버퍼를 처리하기 위해 사용할 수 있는 메모리의 양은 제한되어 있습니다. 페이지를 메모리로 읽은 후 LRU 큐 관리에서는 새 페이지를 위한 공간을 마련하기 위해 이들 버퍼 페이지를 디스크로 플러시하는 시기를 결정합니다. 최근에 사용되지 않은 것을 먼저 사용하는(LRU) 메커니즘을 통해 최근에 사용되지 않은 페이지를 찾을 수 있으므로 필요한 페이지가 디스크로 플러시되는 현상을 최소화시켜야 합니다.

필요한 페이지가 디스크로 플러시되는 상황을 방지하기 위해 테이블을 메모리 상주 테이블로 정의하는 기능을 IDS 7.3에 추가했습니다. 그렇다고 프로세스에서 메모리 상주로 지정된 테이블의 데이터를 요청할 때 이 테이블이 항상 메모리에 있다고 보장할 수 없으므로, 데이터베이스 엔진은 메모리 상주로 지정된 테이블에서 이러한 페이지를 결국 디스크로 플러시하려고 노력하게 됩니다. 테이블이나 인덱스를 메모리 상주로 지정하기 위한 구문은 다음과 같습니다.

```
set table <table name> | index <index name> [ ( <dbspace list>
) ] memory_resident | non_resident [<fragmentation options>]
```

DBSPACE 목록을 통해 특정 DBSPACE에 있는 테이블이나 인덱스의 일부분을 메모리에 상주하도록 지정할 수 있습니다. 전체 sales 테이블(여러 DBSPACE에 있는 모든 분할 포함)을 메모리에 상주하도록 설정하려면 다음의 구문을 사용하십시오.

```
set table sales memory_resident;
```

sales 테이블의 특정 분할을 메모리 상주로 설정하려면 다음 명령문을 실행하십시오.

```
set table sales (dbspace1, dbspace2) memory_resident;
```

이 명령문은 dbspace1 및 dbspace2에 있는 sales 테이블의 분할을 메모리 상주 상태로 놓습니다. sales_idx1 인덱스를 상주로 설정하려면 다음의 명령문을 사용하십시오.

```
set index sales_idx1 to memory_resident;
```

상주 상태에서 sales 테이블을 제거하려면 다음의 명령문을 사용하십시오.

```
set table sales to non_resident;
```

테이블 상주는 set to resident문이 실행된 세션 중에만 유효하다는 점을 유의하십시오. 인스턴스가 종료되고 다시 시작된 후에는 상주 설정 명령문이 다시 실행될 때까지 테이블은 상주 상태로 유지되지 않습니다.

옵티마이저 지시문 사용

Informix 옵티마이저는 일반적으로 쿼리 경로에 대해 매우 양호한 결정을 내리지만 옵티마이저가 올바른 결정을 하는 데 필요한 통계 형태의 충분한 정보를 가지고 있지 않은 경우가 있습니다. 이러한 경우 최적의 쿼리 경로를 선택할 수 있도록 옵티마이저에 힌트를 제공하기 위해 옵티마이저 지시문을 사용할 수 있습니다(IDS 버전 7.3). 옵티마이저 지시문은 다음과 같이 여러 가지 형식으로 사용할 수 있습니다.

- 액세스 방법
- 조인 순서
- 조인 방법
- 최적화 목표

옵티마이저 지시문을 지원하지 않았던 이전 버전의 엔진과의 호환성을 제공하기 위해 옵티마이저 지시문은 SQL 주석 블록에 표시됩니다. 따라서 IDS 7.3 엔진과 IDS 7.3 이전 엔진에서 옵티마이저 힌트가 있는 동일한 쿼리를 지시할 수 있습니다. IDS 7.3 이전 엔진에서는 옵티마이저 지시문이 주석 블록에 표시되므로 무시됩니다. 주석 블록은 다음 중 하나로 표시됩니다.

```
- this is a comment
{ this is a comment }
```

옵티마이저 지시문은 “+” 부호를 첫 번째 문자로 사용하여 주석 블록에 표시됩니다. 옵티마이저 지시문의 구문은 다음과 같습니다.

```
{+ <optimizer directive>}
```

옵티마이저 지시문이 select, update 및 delete 명령문에서 사용할 수 있다는 점을 유의하십시오. 쿼리 경로를 선택할 때 옵티마이저는 여러 가지 요인을 고려합니다. 개발자는 옵티마이저 지시문을 사용하여 이러한 여러 지시문에 대한 힌트를 제공할 수 있습니다. 여러 지시문을 동일한 주석 블록에 표시하여 이들 지시문을 쿼리에 포함시킬 수 있습니다. IDS 7.3 옵티마이저 지시문은 표 7-12와 같습니다.

표 7-12 옵티마이저 지시문

지시문	유형	설명
INDEX	액세스 방법	테이블에 액세스하기 위해 식별된 인덱스만을 고려합니다.
AVOID_INDEX	액세스 방법	테이블에 액세스하기 위해 식별된 인덱스를 고려하지 않습니다.
FULL	액세스 방법	강제로 옵티마이저가 지정된 테이블에 대한 전체 테이블 스캔을 선택하도록 합니다.
AVOID_FULL	액세스 방법	지정된 테이블에 대한 전체 테이블 스캔을 방지하도록 옵티마이저에 지시합니다.
ORDERED	조인 순서 지시문	강제로 옵티마이저가 테이블이 from절에 나타나는 순서대로 테이블을 조인하도록 합니다.
USE_NL	조인 방법	강제로 옵티마이저가 중첩 루프를 선택하도록 합니다.
USE_HASH	조인 방법	강제로 옵티마이저가 해시 조인 방법을 선택하도록 합니다.
AVOID_NL	조인 방법	중첩 루프 조인 방법을 방지하도록 옵티마이저에 지시합니다.
AVOID_HASH	조인 방법	해시 조인 방법을 선택하지 않도록 옵티마이저에 지시합니다.
FIRST_ROWS	최적화 목표	지정된 행 수를 먼저 검색하는 쿼리를 처리하도록 옵티마이저에 지시합니다.
ALL_ROWS	최적화 목표	쿼리에 대한 모든 행을 검색하는 쿼리를 처리하도록 옵티마이저에 지시합니다(기본 동작임).

옵티마이저에게 쿼리의 처음 100개의 행을 효율적으로 검색하도록 지시하려면 다음 명령문을 실행하십시오.

```
select -+ FIRST_ROWS(100)
            sales_id, sales_code, sales_amt
from sales
where region = "NW"
```

옵티마이저에게 중첩 루프 조인을 사용하여 두 테이블의 행을 조인하도록 지시하려면 다음 명령문을 실행하십시오.

```
select {+ USE_NL( region ) }
         sales_id, r.region_name, sales_amt
from sales s, region r
where s.region = r.region;
```

옵티마이저가 쿼리에서 테이블에 액세스하기 위한 특정 인덱스를 선택할 수 있도록 하려면 다음 구문을 사용하십시오.

```
select -+ INDEX( sales sales_idx1 )
            s.sales_id, s.region, r.region_name
from sales s, region r;
```

이 쿼리는 sales 테이블을 지역 테이블에 조인하고 옵티마이저에게 특정 판매 인덱스를 사용하여 테이블에 액세스하도록 지시합니다.

요약

이 장에서는 기본 SQL select문부터 Informix 내장 프로시저 언어(SPL)에 이르기까지 관리자와 개발자에게 가장 중요한 SQL문을 지정하면서 Informix-SQL 쿼리 언어의 서브세트를 다루었습니다. 테이블 제한 조건과 트리거도 다루었으며 이러한 명령문을 실제로 적용하는 것에 대해서 예를 들어 설명했습니다.

문자열 처리 함수, SQL case 표현식 및 옵티마이저 힌트를 포함하여 오랫동안 기다렸던 여러 가지 기능이 Informix 7.3 릴리스에 추가되었습니다. 이 장에서는 이러한 새 기능에 대해 설명과 예를

들어 알아보았습니다.

다음 장 미리보기

다음 장은 Informix-4GL 프로그래밍 도구에 대한 설명으로 시작합니다. Informix-4GL은 Informix-SQL 또는 SPL에서는 불가능한 속도와 기능으로 데이터 검색, 포맷팅, 처리 및 업데이트를 수행할 수 있는 강력한 4세대 프로그래밍 언어입니다. Informix-4GL은 빠르게 배울 수 있으며 C 또는 C++와 같은 3세대 언어의 경우보다 사용이 간편합니다. 그리고 Informix-4GL은 여러 3세대 언어에서는 불가능한 일정한 수준의 언어 안전성을 제공합니다.

이러한 중요한 기능이 있는 Informix-4GL은 IDS 엔진을 사용하여 작업하는 데이터베이스 관리자나 응용프로그램 개발자에게 매우 귀중한 도구가 될 것입니다. 다음 장에서는 이 언어의 중요한 모든 측면을 자세하게 다룰 것입니다.

8장

Informix-4GL 소개

Informix-4GL은 Informix 초기부터 개발된 응용프로그램 언어이며 도구입니다. 짧은 역사에도 불구하고 Informix-4GL은 최고 수준의 제품으로서 한 때 Unix 응용프로그램 개발 시장의 상당 부분을 점유하기도 했던 매우 잘 설계된 제품입니다.

Informix-4GL은 IDS 데이터베이스의 훌륭한 보조 언어로서 광범위한 데이터 처리 및 보고 기능을 제공합니다. 또한 영어와 유사한 구문과 완전한 구조화된 쿼리 언어(SQL)가 통합된, 매우 배우기 쉬운 언어입니다. Informix-4GL은 내장 프로시저 언어(SPL)의 성능을 능가하는 기능과 성능을 제공하여, 데이터베이스 관리자와 응용프로그램 개발자가 보다 나은 보고서를 작성하는 데 용이하게 사용할 수 있는 도구의 사용 영역을 넓혔습니다. 데이터 웨어하우스와 데이터 저장 방법이 편재된 현 IT세계에서, Informix-4GL을 사용하면 C와 같은 언어를 사용하는 것과 비교해 볼 때 최소의 프로그래밍으로 데이터 필터링, 변경, 효율적인 집계 작업을 용이하게 수행할 수 있습니다.

Informix-4GL 언어

4세대 언어(4GL)는 비프로시저문을 사용하여 프로그래밍 수고를 줄이기 위한 목적으로 설계되었습니다. 프로시저문(예: `if`, `while`, `for`문)은 프로그래머가 컴퓨터에 어떤 작업의 수행 방법을 구체적으로 알려야 하는 반면, 비프로시저문을 사용할 때는 간단하게 컴퓨

터에 수행 방법만 알려주면 됩니다. 세부사항을 처리하는 것은 언어에 달려 있습니다. 4세대 언어는 그림 8-1에 표시된 대로 프로그래밍 언어의 진보에 따라 현재 최고로 발달된 언어임을 알 수 있습니다.

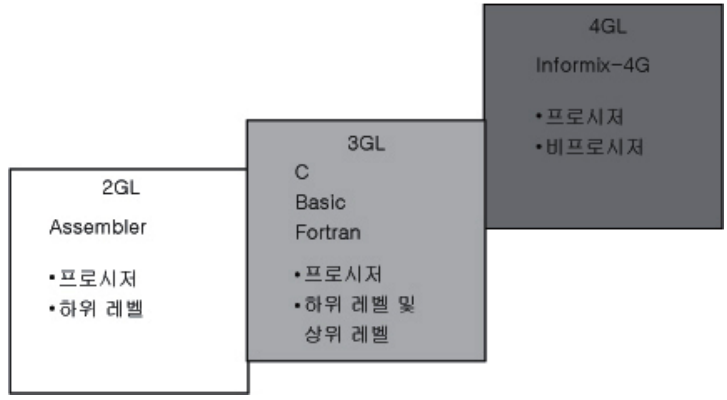


그림 8-1 언어의 세대별 역사

Informix-4GL이 다른 4세대 언어와 구분되는 점은 프로시저문과 비프로시저문이 유연하게 혼합된다는 점입니다. Informix-4GL의 설계자들에 따르면 4GL은 비프로시저문 한 쪽에만 의존할 수 없다고 합니다. 언어가 유연해지고, 복잡한 응용프로그램을 개발하는 데 충분한 기능을 제공하기 위해서는 언어에 적절한 양의 프로시저문이 포함되어 있어야 합니다. 적절한 프로시저문 세트가 없는 경우, 4GL 프로그래머는 응용프로그램을 완료하기 위하여 어쩔 수 없이 3GL문에 의존해야 합니다. 이러한 경우 처음에 4GL을 사용하게 했던 4GL의 모든 장점을 순식간에 사라지게 하는 지루한 프로세스가 될 수도 있습니다.

Informix-4GL 언어는 명령문, 연산자, 블록 및 함수로 구성되어 있습니다. Informix-4GL 프로그램은 유효한 입력점 및 유효한 종료 중 어떤 것으로도 임의로 조합하여 정의할 수 있습니다. 유효한 입력점은 기본 프로그램 블록을 통하며 유효한 종료는 내재적(기본 프로그램 블록의 종료를 표시하는 `end main`문에서와 마찬가지로) 이거나 `exit program`문을 가리킵니다.

명령문 및 연산자

Informix-4GL 명령문은 프로그램 연산을 제어합니다. 명령문에는 흐름 제어, 조건 분기 작업 또는 비프로시저문이 있습니다. 프로그램 블록, 함수 블록 또는 보고서 블록이 이 블록에 해당합니다.

프로그램 블록

프로그램 블록 중 메인 블록만이 필수이며, 다른 모든 Informix-4GL 블록은 선택적입니다. 모듈 프로그래밍의 원칙을 고수하기 때문에 분명하게 함수 블록의 사용을 장려하기는 하지만, 다른 프로그램 블록에 대한 요구사항은 없습니다.

런타임 시에는 메인 프로그램 블록을 우선 실행하는 상태에서 블록이 연속적으로 실행됩니다. 함수 블록이나 보고서 블록과 같은 추가 블록은 프로그램 제어 순서가 이들 블록 중 하나로 제어를 전달했을 때 실행됩니다. 프로그램 실행은 `exit program`문이 나타나거나 블록을 종료하는 `end main`문이 나타났을 때 종료됩니다.

함수는 함수 선언(`function function_name`)과 `end function`문 사이의 모든 언어 명령문과 연산자로 구성됩니다. 둘 이상의 함수가 소스 코드 모듈에 포함될 수도 있습니다. 함수 선언 내에서는 함수 선언을 작성할 수 없습니다.

언어 연산자

Informix-4GL 연산자는 더하기, 빼기, 곱하기, 나누기, 모듈과 같은 수학 연산자가 될 수도 있고 또는 문자열 연결 연산자와 같은 문자열 중심의 연산자가 될 수 있습니다.

연산자는 선택적으로 괄호를 사용하여 명시적으로 그룹화될 수 있습니다. 연산자의 선행 순서와 설명은 9장 “화면 폼”에서 설명합니다.

메모리 제어

Informix-4GL은 C와 같은 다른 언어에서는 일반적으로 사용할 수 없는 안전 레벨을 제공하며 모든 메모리를 할당합니다. 하지만

프로그래머는 프로그램 메모리로 직접 액세스할 수 없습니다. C 언어가 제공하는 자유로운 메모리 액세스에 익숙한 프로그래머에게 이것은 장애물처럼 보일 수도 있지만, C 언어에서처럼 악명 높은 메모리 누출을 추적하면서 며칠을 보내지 않아도 됩니다.

자료형 지정

Informix-4GL은 매우 엄격한 자료형 지정 기능을 제공합니다. 변수는 사용되기 전에 특정 자료형으로 선언되어야 합니다. 변수는 지정 명령문이 포함한 자료형으로 존재를 드러내지는 않습니다. 이러한 제어 레벨에서 프로그래머는 특정 시점에서 변수의 값과 특성에 대하여 확신할 수 있습니다.

엄격한 자료형 지정은 데이터 변환이 필요할 때 제한될 수 있습니다. Informix-4GL에서 데이터 변환의 범위는 지정 명령문 내에 포함됩니다. 변수에 포함된 값의 자료형을 변환하려면 프로그래머는 단지 변수를 원하는 자료형으로 지정하기만 하면 됩니다. 변환이 적절하고 수행 가능하면, Informix-4GL은 변환을 수행합니다.

데이터베이스 상호 작용

Informix-4GL 언어는 산업 표준 구조화된 쿼리 언어(SQL)를 통해 Informix와 밀접하게 바인드됩니다. Informix 자료형은 Informix-SQL 자료형과 동일하며 SQL은 유기적으로 언어와 통합됩니다.

4GL 응용프로그램의 파트

Informix-4GL 응용프로그램은 프론트엔드 응용프로그램(컴파일된 프로그램으로 구성)과 백엔드(기본적으로 IDS 데이터베이스)로 구성된 2계층 아키텍처를 사용합니다. ESQL/C 도구에서 사용되는 것과 동일한 통신 계층인 IDS 데이터베이스 연결 계층을 사용하여 프론트엔드에서 백엔드 프로그램으로 통신할 수 있습니다.

Informix-4GL 응용프로그램은 소스 코드의 모듈 세트로 구성되어 있습니다. 소스 코드 모듈에는 명령문 세트, 데이터 변수 및 데이터 변수에서 수행되는 연산이 들어 있습니다. 최소한 하나의 소스

코드 모듈에서는 메인 블록 명령문이 필수이며 다른 명령문은 모두 선택적입니다. 모듈은 실행 가능한 Informix-4GL 응용프로그램을 구성하기 위해 함께 컴파일되어 연결됩니다(그림 8-2 참조). 이 프로세스는 Unix make 유틸리티 또는 Informix-4GL 프로그래머의 환경을 사용하여 더욱 간소화할 수 있습니다.

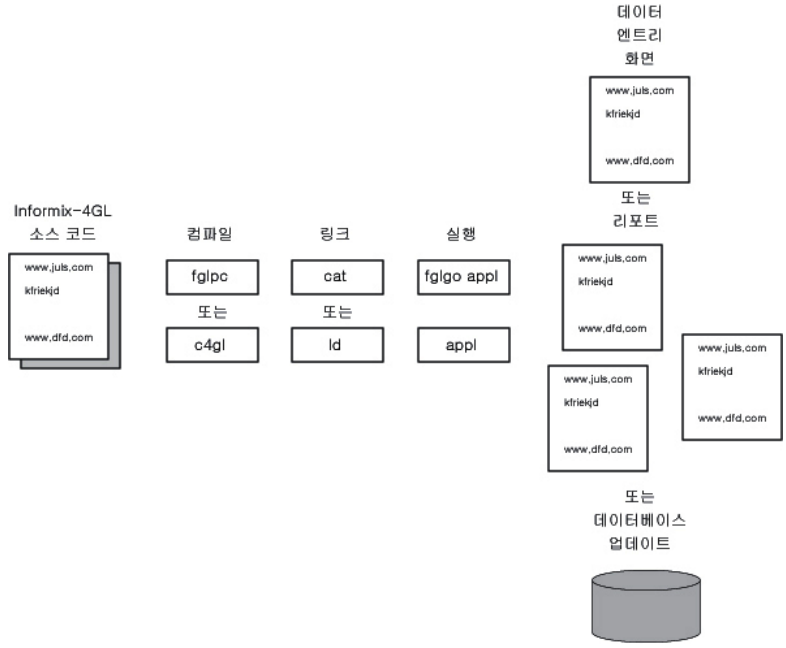


그림 8-2 Informix-4GL 개발

Informix-4GL 프로그램 모듈의 작성에 대한 구체적인 규칙은 없습니다. 대개는 특정 모듈이 함수 블록의 세트로 구성되면, 각 모듈은 응용프로그램의 전반적인 작업 중 일부분을 논리적으로 수행하는 Informix-4GL 함수 그룹을 포함하게 됩니다.

모듈 프로그래밍의 규칙을 고수하면 각 모듈이 명령문 블록 세트를 포함하는 소스 코드 모듈의 세트로 구성된 프로그램을 작성하게 됩니다. 정의상으로는 단지 하나의 메인 명령문 블록이 존재합니다. 여러 개의 함수 블록이 몇 가지 서로 다른 프로그램 모듈 전체에 걸쳐 분산되게 있어야 합니다. 각 함수 블록에는 제한된 수의 프로그램 명령문과 연산이 포함됩니다. 각 함수의 크기는 응용프로그램의 읽기 가능성과 유지 가능성을 개선하기 위하여 적은 수준에서 유지됩니다.

Informix-4GL과의 사용자 상호 작용

Informix-4GL은 화면 폼 기능을 사용하여 대부분의 화면 상호 작용을 수행합니다. 화면 폼은 데이터 입력 또는 화면으로 데이터 출력을 통해 사용자가 상호 작용을 수행할 수 있는 문자 기반의 기능입니다. Informix-4GL 화면 폼을 사용하면 데이터 화면을 텍스트 파일로 개발하므로 컴파일된 후 런타임 시 응용프로그램에서 사용될 수 있습니다. Informix-4GL 프로그램은 데이터를 화면에 입력하고 출력하는 일련의 명령문을 사용하여 화면 폼과 상호 작용합니다.

보고서 작성 및 데이터 필터링

Informix-4GL은 풍부한 비프로시저 보고 구문을 제공합니다. 이 구문을 사용하여 프로그래머가 데이터베이스에서 데이터를 스캔하고 정렬한 다음 보고서 함수로 데이터를 전달하여, 제어 중단을 생성하고 처리 중인 데이터를 포맷하도록 일련의 비프로시저문을 실행할 수 있습니다. Informix-4GL 제어 중단은 프로시저 언어 명령문을 사용하면서 스캔 중인 데이터를 처리할 때 유용한 지점입니다. 또한 이 기능은 데이터 웨어하우스 또는 데이터 마트로 삽입하기 위해 데이터를 필터링하고 변환할 때에도 매우 유용한 것으로 증명되었습니다.

Informix-4GL을 사용하여 모듈 프로그래밍

Informix-4GL은 균형 잡힌 프로시저문과 및 비프로시저문을 제공합니다. 이 기능은 모듈 방식의 응용프로그램 개발을 촉진시키려는 목적으로 설계된 블록 중심의 명령문을 사용하므로 가능합니다. 이 언어는 COBOL에 대한 대체 언어임에도 불구하고 COBOL의 모듈 프로그래밍 오류는 포함하고 있지 않습니다. 사실 이 설계는 COBOL이나 FORTRAN보다는 PASCAL, ALGOL, C와 같은 블록 중심의 모듈 언어에서 더 많이 유래되었습니다.

모듈 프로그래밍의 장점은 잘 알려져 있으며 한 때 문서로도 작성되었습니다. Informix-4GL은 모듈 프로그래밍을 염두에 두고 작성

되었습니다. 이 언어는 블록 중심이며, 모듈 응용프로그램의 개발을 더욱 용이하게 만드는 구조화된 프로시저문 및 구조화된 비프로시저문을 모두 다양하게 포함하고 있습니다.

모듈 프로그래밍은 소형에서 중형 크기의 소스 코드 모듈 세트에 이르기까지 응용프로그램을 개발하는 데 필요합니다. 복잡성은 모듈의 크기를 소형으로 유지하고 소형의 재사용 가능한 함수로 기능을 요약함으로써 관리됩니다. 많은 함수가 블랙 박스 함수로 될 수 있으므로 이로 인해 프로그래머가 신경 쓸 필요가 없는 코드가 포함될 수도 있습니다. 프로그래머는 작업의 구체적인 사항이 아니라 함수의 사용 방법에만 주의를 기울여야 합니다. 프로그래머는 어떤 매개변수가 함수로 전달될 것인지 어떤 매개변수가 리턴될 것인지 만을 알고 있으면 됩니다.

Informix-4GL은 이러한 방식으로 사용하기 위해 설계되었습니다. 모든 기능을 갖춘 함수의 구현을 통해 실제적으로 모든 유효한 자료형의 다중 함수 매개변수를 수신하고, 모든 값을 리턴하는 기능을 제공합니다. 이러한 기능의 구현은 모듈 함수 라이브러리 개발을 더욱 용이하게 해 줍니다.

Informix 자료형 및 연산자

IDS 데이터베이스와 상호 작용을 해야 하는 언어로서 Informix-4GL 언어는 IDS 자료형과 이러한 자료형을 처리하는 데 필요한 연산자를 모두 지원합니다. 다음 절에서는 Informix-4GL 자료형과 유효한 자료형 연산자에 대해 설명합니다.

Informix-4GL 자료형

Informix-4GL은 대부분의 3GL 언어에서 지원하는 모든 자료형을 지원합니다. Informix-4GL의 간단한 자료형은 BLOB(Binary Large Object) 자료형을 포함한 Informix 데이터베이스가 지원하는 자료형과 일대일 대응입니다. (배열과 레코드와 같은 더욱 복잡한 자료형에는 분명히 이러한 대응이 없습니다.) 표 8-1에는 스토리지 크기 및 최소 최대값과 함께 모든 Informix-4GL 자료형이 나열되어 있습니다.

표 8-1 유효한 Informix-4GL 자료형

자료형	길이	최소값	최대값
smallint	2	-32,767	+32,767
integer, int	4	-2,147,483,647	+2,147,483,647
smallfloat, real	4	플랫폼 종속	플랫폼 종속
float, double precision	8	플랫폼 종속	플랫폼 종속
decimal, dec, numeric	정밀도/2+1	-32 유효숫자	+32 유효숫자
money	정밀도/2+1	-32 유효숫자	+32 유효숫자
date	4	1/1/0001	12/31/9999
char, character	프로그래머 정의	1byte	32,767byte

Smallint

Informix-4GL의 smallint는 2byte의 정수이며 C 언어의 short 정수에 해당합니다. 이것은 부호가 표시되고 +/-32,767의 최대/최소값을 지원합니다. Informix-4GL 정수 오버플로는 오류입니다. Informix-4GL의 p 코드/RDS 버전에서 이 오류는 기본값으로 트랩됩니다. C 코드 컴파일 Informix-4GL에서 이 오류는 단지 whenever any error문이 유효한 경우에만 트랩됩니다.

Integer

Informix-4GL의 integer는 4byte의 정수이며 C 언어의 long 정수에 해당합니다. 이것 또한 부호가 표시되고 +/-2,147,483,647의 최대/최소값을 지원합니다. 정수 오버플로 오류는 smallint 자료형을 사용하였을 때와 마찬가지로 트랩됩니다.

안타깝게도 최대 정수값에 대한 테스트를 위해 제공되는 매크로 상수는 없습니다. 프로그래머는 이러한 값을 변수로 저장하고 이에 따라 점검해야 합니다. Informix-4GL 화면 폼은, 사용자가 이러한 값을 입력할 수 있는 경우 입력 중인 숫자 데이터를 자동으로 유효화하고, 입력된 숫자가 Informix integer에 대하여 허용된 최대값보다 큰 경우, 오류 코드를 리턴합니다.

Smallfloat

Informix-4GL의 smallfloat는 C 언어의 float 또는 real 자료형과 거의 유사합니다. smallfloat에는 공식적으로 머신에 종속적이지만 일반적으로 +/-1016의 최대값과 최소값이 있습니다.

Decimal

초기 Unix 머신에서 부동 소수점 숫자의 사용에 관련된 어려움과 변형으로 인해, Informix는 자체적인 폼의 부동 소수점 표시를 개발했습니다. 이것이 decimal 자료형입니다. 이 설계에서는 숫자 데이터가 머신 사이에서 일정한 포맷으로 저장되기 때문에 네트워크로 데이터를 이동시키는 것이 더욱 쉬워졌습니다.

Informix-4GL 프로그램에서 decimal 자료형은 내부적으로 표 8-2에서 표시된 대로 저장됩니다.

표 8-2 decimal 자료형 스토리지

구성 요소	스토리지
지수	2바이트 정수
소수 자리	2바이트 정수
자릿수	2바이트 정수
숫자	선언된 정밀도/2의 길이인 문자

decimal 자료형은 선언 프로세스 중 정밀도가 지정된 경우에는 고정 소수점이 되고, 정밀도가 지정되지 않은 경우에는 부동 소수점이 됩니다. 정밀도는 +/-1032 또는 32개의 유효 숫자로 매우 큽니다. 이러한 자료형에 대한 Informix-4GL 선언은 다음과 같습니다.

```
define dec_var  decimal
```

이 선언이 정밀도를 지정하고 있지 않기 때문에 이 변수의 정밀도 기본값은 16자리가 되고 10진수 소수점은 부동입니다. 그러나 변수가 다음과 같이 선언되면,

```
define dec_var  decimal(10)
```

최대 10자리의 유효 숫자를 사용하여 decimal 변수를 생성합니다. 그렇지 않으면 변수는 다음과 같이 정의됩니다.

```
define dec_var  decimal(10, 2)
```

이렇게 하면 정밀도 2(소수점에서 오른쪽으로 2자리)를 사용하여 10자리의 decimal 변수가 생성됩니다.

다음과 같이 m이 정밀도 이고 n이 스케일이면 선언된 decimal 데이터 변수에서 가능한 최대값은 $10^{m-n}-10^{-n}$입니다.

```
define dec_var  decimal(m, n)
```

이 경우 decimal은 다음과 같이 선언될 수 있습니다.

```
define dec_var  decimal(2, 0)
```

이 변수는 최대 $10^{2-0}-10^{-0}$ 즉, 99(100-1)를 저장할 수 있습니다. 이보다 큰 숫자를 저장하려는 모든 시도는 표현식 오류를 생성하게 되고 오류 검사의 모든 오류 모드가 유효해질 때마다 트랩되어야 합니다. (현재 Informix-4GL 버전은 기본적으로 이 오류를 트랩하지 않을 수 있습니다.)

Money

money 자료형은 특수 decimal 자료형입니다. 기본적으로 이는 스케일 2를 사용한 고정 10진수입니다. money 자료형을 표시했을 때 기본적인 통화 부호는 표시되는 숫자의 왼쪽에 표시됩니다. 기본적인 출력 통화 부호는 다음 포맷의 DBMONEY 환경 변수를 사용하여 변경할 수 있습니다.

```
DBMONEY=scb
```

이 포맷에서 s는 통화값 앞에 오는 부호이고, c는 전체 통화 부분을 소수 부분과 구분하는 부호이며, b는 통화값 다음에 오는 부호입니다. c에 대한 유효값은 “.” 또는 “,”입니다. s 또는 b에 대한 유효

값은 콤마 또는 마침표를 포함하지 않는 일련의 문자입니다.

Float

Informix-4GL의 float 자료형은 일반적으로 머신 플랫폼의 C double과 동일한 정밀도를 갖습니다. 정밀도는 머신에 종속되지만 일반적으로는 1032로 제한됩니다. 언어 정의 double precision은 float의 동의어입니다.

Date

Informix-4GL의 date는 내부적으로 1900년 1월 1일을 포함하여 이후의 날짜 수를 표시하는 4byte 정수로 표시됩니다. Informix-4GL은 기본 포맷 또는 DBDATE 환경 변수가 지정하는 포맷에 따라 이 값을 자동으로 포맷합니다. DBDATE 환경 변수는 다음과 같이 설정됩니다.

```
DBDATE=dmync
```

여기서, d는 설정된 날이고, m은 설정된 월, y는 연도, n은 연도 자리 이후의 숫자(유효 자릿수는 2자리 또는 4자리)입니다. c는 날짜의 구성 요소를 구분하는 데 사용되는 문자입니다.

Datetime

datetime 자료형은 시간으로 인스턴스를 표시하는 방법으로, 여기에는 날짜와 시간 구성 요소가 모두 포함됩니다. datetime 자료형에는 가변의 정밀도가 있습니다. 사용자는 datetime 변수를 특정 정밀도로 정의할 수 있지만 필요에 따라 정밀도를 동적으로 변경할 수 있습니다. datetime 자료형은 년, 월, 일, 시간, 분, 초, 초의 소수 필드로 구성됩니다. 이 자료형은 레코드를 사용하여 타임스탬프를 저장해야 하는 응용프로그램 사용 시 유용합니다. datetime 자료형은 다음과 같이 선언됩니다.

```
define variable_name datetime first to last
```

이 예에서 first 또는 last 식별자는 표 8-3의 필드 중 하나입니다. 동일한 datetime 정의에서 last 엔트리에 대한 레벨은 리스트의 first 엔트리보다 앞에 올 수 없습니다.

*표 8-3 **Datetime** 식별자*

필드	유효한 엔트리	레벨
년	1-9999 사이의 숫자	1
월	1-12 사이의 숫자	2
일	1-31 사이의 숫자(31은 월에 따라 다름)	3
시간	0-23 사이의 숫자	4
분	0-59 사이의 숫자	5
초	0-59 사이의 숫자	6
소수 초	5개의 소수 자리	7

Interval

interval 자료형은 시간의 범위를 표시하는 데 사용됩니다. 이 자료형은 두 datetime 간의 차이를 나타내며 또한 다양한 밀도를 지원합니다. 밀도는 런타임 시 변수를 정의하거나 변경할 때 정의할 수 있습니다.

interval 자료형은 날짜와는 독립적으로 시간의 범위를 표시하도록 설계되었기 때문에 이 자료형은 날짜와 시간을 혼합할 수 없는 특정 구성 요소로 표시되어야 합니다. interval 유형에는 년/월 시간 간격을 표시하는 유형과 일/시/분/초를 표시하는 유형의 두 가지가 있습니다. interval은 다음과 같은 포맷으로 선언됩니다.

```
define interval_var interval first to last
```

이 예에서 first 또는 last 식별자는 표 8-4와 8-5의 필드 중 하나입니다. 동일한 datetime 정의에서 last 엔트리에 대한 레벨은 리스트의 first 엔트리보다 앞에 올 수 없습니다.

표 8-4 년/월 ***Interval***

필드	유효 엔트리	레벨
년	1-9999 사이의 숫자(기본) 1-999,999,999 사이의 숫자(최대)	1
월	1-99 사이의 숫자(기본) 1-999,999,999 사이의 숫자(최대)	2

표 8-5 일/시 ***Interval***

필드	유효 엔트리	레벨
일	1-99 사이의 숫자(기본) 1-999,999,999 사이의 숫자(최대)	1
시	1-99 사이의 숫자(기본) 1-999,999,999 사이의 숫자(최대)	2
분	1-99 사이의 숫자(기본) 1-999,999,999 사이의 숫자(최대)	3
초	1-99 사이의 숫자(기본) 1-999,999,999 사이의 숫자(최대)	4
소수 초	1-999 사이의 숫자(기본) 1-99,999 사이의 숫자(최대)	5

기본 엔트리는 interval 변수의 선언 시 무시할 수 있는 시스템 기본값입니다. 다음 선언 명령문에서 이를 알 수 있습니다.

```
define interval_var interval year(5) to month
```

이 예에서 interval 변수는 년도에서 기본값으로 4자리 대신 5자리를 사용할 수 있도록 정의했습니다.

Character

Informix-4GL의 character 자료형은 길이가 최대 32,767자인 문자열입니다. 정의한 길에는 항상 공백이 채워져 있습니다. 일반적으로 NULL로 종료되지만, 길이의 결정은 NULL 터미네이터에 의존하지 않습니다. character 변수가 정의되면 길이는 고정됩니다.

콤플렉스 자료형

다수의 4GL 언어와는 달리 Informix-4GL은 콤플렉스 자료형을 지원합니다. 레코드, 배열, 변수에 대한 데이터베이스 바인딩에 사용할 수 있습니다. 이러한 콤플렉스 자료형은 Informix-4GL 언어 기능의 중요한 부분입니다.

배열

Informix-4GL 배열은 동종의 데이터 요소의 컬렉션입니다. 배열은 3차원까지 선언할 수 있습니다. 컴파일 시점에서 배열의 크기가 고정되므로 현재 동적으로 증가할 수 없습니다.

기본적으로 컴파일된 C Informix-4GL에서 배열 범위는 런타임 시 점검되지 않습니다. 그러나 특수 컴파일러 플래그를 컴파일러에 전달할 수 있어 이 컴파일러가 런타임 시 배열 범위를 점검하도록 합니다. 모든 배열 참조 작업에서 배열 범위를 점검하는 것은 비용이 많이 들 수 있으므로, 이 플래그는 디버깅 단계 중에서만 사용하고 프로그램의 생산 버전에는 사용하지 않을 것을 권장합니다.

레코드의 배열은 사용할 수 있는 반면, 배열의 배열은 사용할 수 없습니다. 배열은 함수 매개변수로서 전달될 수 없습니다.

레코드

Informix-4GL은 실제로 제한되지 않은 길이의 같은 유형이 아닌 자료형 레코드를 지원합니다. 레코드 내의 요소는 레코드와 같은 유효한 모든 기본 자료형 또는 콤플렉스 자료형이 될 수 있습니다. 또한 배열은 레코드 구성 요소가 될 수도 있습니다.

레코드가 함수 매개변수로서 전달될 수도 있습니다. 이 때 함수는 값 단위 호출을 사용하여 이를 처리한다는 점에 유의하십시오. 즉, 전체 레코드 변수를 함수 스택에 넣고 빼게 됩니다. (이는 레코드 또는 구조가 포인터를 사용하여 참조 단위로 호출되므로 C 언어와는 다릅니다. 즉, 함수가 단지 변수의 값이 아니라 실제 변수에서 실행됩니다.)

데이터베이스 바인딩

Informix-4GL을 사용하여 변수를 컴파일 시 데이터베이스 테이블에 바인드할 수 있습니다. 이러한 변수는 데이터베이스 테이블과 직접 대응됩니다. 이렇게 하면 언어의 안전성이 높아지고 실제로 데이터베이스에서 데이터를 검색할 때 매우 일반적인 자료형 오류가 제거됩니다. 데이터베이스 바인딩은 다음과 같이 변수 선언에서 like 키워드를 사용할 때 발생합니다.

```
define lcustomer like customer.first_name
```

다음 선언 예의 별표(*)는 데이터베이스 테이블의 모든 컬럼을 분석합니다.

```
define lcustomer_rec record like customer.*
```

이 명령문을 실행하면 순서와 자료형이 일치하는 모든 컬럼(참조된 데이터베이스 테이블 내에 있는 컬럼)의 레코드를 생성합니다.

through절은 레코드 선언에서 사용할 데이터 테이블 컬럼의 일부만을 식별하는 데 define문과 함께 사용할 수 있습니다. 이 절의 예는 다음과 같습니다.

```
define lcustomer record like
    customer.first_name through customer.last_name
```

이 기능의 장점을 과소평가하면 안 됩니다. 이러한 방식으로 레코드를 선언하는 기능을 사용하면 프로그래머가 변수의 선언 방법을 찾기 위해 데이터베이스 스키마를 일정하게 참조하지 않아도 됩니다. 가능할 때마다 이 기능은 대문자로 처리해야 합니다.

이 명령문을 사용할 때 발생하는 데이터베이스 바인딩은 컴파일 단계 동안 발생한다는 점에 유의하십시오. 이로 인해 like 키워드를 사용하여 레코드가 customer 테이블에 바인드되고 프로그램이 컴파일된 후 customer 테이블이 프로그램 실행 이전에 수정되면, 런타임 시 유형 불일치 오류가 발생할 수 있습니다. 구조화된 개발 환경에서 데이터베이스 스키마는 소스 코드와 동일하거나 더욱 엄격한 제어 하에 있으므로 이러한 오류의 가능성을 최소화해야 합니다.

NULL 데이터 값

NULL 자료형은 데이터의 부재를 표시하는 특별한 데이터 값이며 평가 중인 자료형에 따른 여러 가지 내부 값이 있습니다. NULL 자료형을 사용하여 숫자 영(0), 공백, 또는 날짜 영(0)과 같은 값과 이러한 자료형의 알 수 없는 값을 구분할 수 있습니다.

예를 들어, 계정의 잔액을 저장하는 컬럼의 값이 영(0)일 수 있습니다. 여기서 해당 계정의 잔액은 영(0)이 아닌데, 트랜잭션이 값을 영(0)으로 설정했을 수 있습니다. 이 영(0)은 계정이 한번도 사용된 적이 없음을 나타내는 NULL 값과는 차이가 있습니다.

Informix-4GL 표현식에서 NULL을 사용하면 문제가 발생할 수 있습니다. 모든 부울 표현식에서 NULL 데이터 값은 전체 표현식에 대해 FALSE를 리턴합니다. 다음 예를 고려해 봅시다.

```
define val1, val2 integer
let val2=100000
initialize val1 to NULL

if (val2 > val1) then
      display "val2 is the larger number"
else
      display "val1 is the larger number"
end if
```

위의 부울 표현식은 FALSE로 평가되며, `val2`의 값이 `val1`의 값보다 더 큰 것처럼 보이는 경우에도 표시문은 "val2 is the larger number"라는 메시지를 표시하지 않습니다. 이 값은 NULL 값이 데이터의 부재를 나타내는 값이고 다른 어떤 값보다 크거나 적은 값을 가질 수 없다는 이론적인 주장에 근거한 의도적인 동작입니다.

이는 표현식이 프로그래머의 예상대로 동작하도록 하려면, 데이터 변수가 NULL 값을 포함할 수 있는 경우 다음과 같이 표현식 내에서 NULL 값에 대해 데이터 변수를 명시적으로 테스트해야 함을 의미합니다.

```
define val1, val2 integer
let val2 = 100000
initialize val1 to NULL
```

```
if val1 IS NULL and val2 IS NOT NULL then
      display "val2 is a number and val1 is nothing."
end if
if val2 IS NOT NULL and val1 IS NOT NULL and val2 > val1 then
      display "val2 is the larger number"
else
      display "val1 is the larger number"
end if
```

Informix-4GL 프로그래밍에서 숫자 값을 사용할 때 때때로 적용하는 기법은 사용 전에 데이터 값을 스캔하고 NULL을 영(0)값으로 매핑함으로써 표현식의 NULL을 제거하는 것입니다. 이 기법은 다음과 같습니다.

```
define val1, val2 integer
let val2 = 100000
initialize val1 to NULL

   ....

if val1 IS NULL then
     let val1 = 0
end if
if val2 IS NULL then
     let val2 = 0
end if

if val2 > val1 then
      display "val2 is the larger number"
else
      display "val1 is the larger number"
end if
```

지정문의 데이터 변환

Informix-4GL은 변환이 어떤 단순한 규칙을 따르는 한, 표현식

에서 자동 데이터 변환을 수행합니다. 이러한 내재적 데이터 변환에서는 정수가 간단한 지정 연산이 있는 문자열로 변환될 수 있습니다. 다음 명령문은 이러한 변환을 수행합니다.

```
define x     integer
define str   char(10)

let x = 1 / 2 * 3
let str = x
```

이 명령문은 정수값 1이 있는 문자열을 작성합니다.

어떤 경우 정수값은 오른쪽으로 이동되고 그 빈자리에 공백이 남아 있습니다. 대부분의 경우 이는 바람직하지 않습니다. 정수 문자열이 데이터베이스 키로서 데이터베이스에 저장되어 있고 리턴 시 정렬되는 경우에는 특히 그렇습니다. 이런 경우, 정렬된 데이터는 인식할 수 있는 순서로 표시되지 않습니다. 이 문제에 대한 간단한 해결책은 데이터를 포맷하여, 문자열에 지정되었을 때 더 이상 오른쪽으로 이동되지 않도록 하는 것입니다. 이것은 using절을 이용하여 수행할 수 있습니다. using절은 일반적으로 표시 또는 인쇄 옵션에서만 사용됩니다. 구문은 다음과 같습니다.

```
let sir = x using "<<<<"
```

왼쪽 화살표는 포맷 중인 정수가 문자열에 배치될 때 오른쪽으로 이동되어야 한다는 것을 나타냅니다. using절에 대한 포맷 연산자의 전체 세트가 표 8-6에 나열되어 있습니다.

표 8-6 using절을 사용한 포맷 문자

포맷 문자	작업
#	0에 공백을 삽입합니다.
&	0에 0을 삽입합니다.
<	결과를 왼쪽같이 표현합니다.
$	숫자의 끝에 $를 표시합니다.
*	0에 *을 삽입합니다.
.	숫자 스트림에서 이 자리에 마침표를 표시합니다.

<계속>

표 8-6 using절을 사용한 포맷 문자

포맷 문자	작업
,	숫자 스트림에서 이 자리에 콤마를 표시합니다.
( )	음수를 괄호 안에 표시합니다.
-	숫자가 음수이면 숫자에 음부호를 표시합니다.
+	숫자가 양수이면 숫자에 양부호를 표시합니다.
mm	월 숫자를 삽입합니다.
dd	일 숫자를 삽입합니다.
mmm	월의 이름을 삽입합니다.
ddd	요일을 삽입합니다.
yy	연도를 2자리로 삽입합니다.
yyyy	연도를 4자리로 삽입합니다.

이 포맷 문자를 사용하여 선택적 컬럼을 특정 위치에 지정하려면, 다음 포맷 문자열을 사용하십시오.

```
let x =  133444555
let str = x using "<<<,<<<,<<<<"
```

결과는 다음과 같습니다.

```
133,444,555
```

Informix-4GL은 필요하다고 인식할 때 다양한 변환을 수행합니다. 표 8-7에서는 이러한 많은 변환에 대하여 자세히 설명합니다.

표 8-7 Informix-4GL의 지정 변환 규칙

표현식	기본 규칙
let string = numeric	왼쪽 이동이며 공백으로 채워집니다.
let integer = string	문자열이 정수를 나타내는 ASCII 문자 스트림인 경우 허용됩니다. 그렇지 않은 경우 치명적 오류가 발생합니다.
let integer = decimal	정수로 절단됩니다.

<계속>

표 8-7 Informix-4GL의 지정 변환 규칙

표현식	기본 규칙
let string1 = string 2	string1의 길이가 string2를 포함할 정도로 충분히 긴 경우 오류나 경고가 발생하지 않습니다. string1의 길이가 string2 보다 짧은 경우 절단이 발생하며 경고 신호를 보냅니다. 오류는 아닙니다.
let decimal1 = decimal 2	정밀도를 초과하지 않은 한 필요 시 왼쪽 인수의 스케일을 변경합니다. 정밀도를 초과하면 치명적 오류가 발생할 수 있습니다.
let date = datetime	datetime은 연도에서 날짜까지의 포맷이어야 합니다.

표현식에서의 지정 및 변환

Informix-4GL은 표현식 중에 유형 변환을 수행하기 위해 내부 규칙 세트를 사용합니다. 각 표현식을 분석하고, 각각의 인수는 자료형 및 인수의 길이와 함께 Informix-4GL 스택에 위치합니다. 인수를 스택에서 빼내면 변환을 인식할 수 있는 한 유형 변환이 발생합니다.

문자 변환

Informix-4GL은 오류 발생 없이 모든 유효한 자료형을 문자열로 변환합니다. 절단이 발생하면 경고 메시지가 표시됩니다. 이 경고가 트랩되지 않은 경우, 프로그램은 계속하여 실행됩니다. 표현식의 양끝이 문자이고, 표현식의 왼쪽이 오른쪽보다 더 짧은 경우, 경고가 발생하지만 프로그램은 계속 실행됩니다. 다음의 코드 조각으로 이러한 개념을 알 수 있습니다.

```
define str1 char(5)
define str2 char(10)

let str2 = "0123456789"
let str1 = str2
display "str1 is" , str1
```

이 코드를 실행하면 다음과 같은 결과가 산출됩니다.

```
01234
```

문자열이 숫자를 포함할 정도로 길지 않으면 문자열이 절단되고 경고가 발생합니다. 프로그램은 계속 실행됩니다. 이 동작은 문자 변환 표현식의 오른쪽에서 사용된 모든 숫자 자료형에 적용됩니다.

산술 연산자와 숫자 변환 규칙

Informix-4GL은 수학적 연산자의 전체 범위를 지원합니다. 이러한 연산자에는 정확하게 사용자가 예상하는 것은 아니지만 특정한 평가 순서가 있습니다. 표 8-8에는 유효한 Informix-4GL 연산자와 해당 평가 또는 그룹화 레벨의 목록이 나열되어 있습니다.

표 8-8 연산자의 그룹화

연산	부호	평가 레벨
결합	( )	1
누승	**	1
곱하기	*	2
나누기	/	2
계수	mod	2
더하기	+	3
빼기	-	3

숫자 연산에서 모든 피연산자는 Informix-4GL의 `decimal` 자료형으로 변환됩니다. 그 결과로 얻은 숫자는 10진수입니다. 결과의 정밀도와 스케일에 대한 규칙은 표 8-9에 상세하게 설명되어 있습니다. 결과를 왼쪽 값에 배치하는 경우, 오류가 리턴되지 않습니다. 유효 숫자가 상실되면, 때때로 치명적 오류가 리턴됩니다. 오류 검사의 `whenever any error` 모드를 사용하면 모든 표현식 오류에 대해 오류 검사를 수행합니다.

표 8-9 수학적 연산 결과의 정밀도와 스케일

연산	정밀도	스케일
더하기 또는 빼기	MIN(32, MAX(p_1-s_1, p_2-s_2)+ MAX(s_1, s_2) + 1)	MAX(s_1, s_2)
곱하기	MIN(32,p_1+ p_2)	s_1 + s_2
나누기	32	32- p_1 + s_1 - s_2

위의 테이블은 수학적 연산 결과의 정밀도와 스케일을 자세하게 설명하고 있습니다. 여기서, p_1과 s_1은 첫 번째 피연산자의 정밀도와 스케일이며 p_2와 s_2는 두 번째 피연산자의 정밀도와 스케일입니다.

초기화

정의에 따라 Informix-4GL 변수는 정의되었을 때 초기화되지 않는다는 사실에 유의하십시오. 컴파일러의 p-코드 버전은 데이터 영역을 2진수 0으로 초기화합니다. 이는 p-코드 응용프로그램을 체계적으로 Informix-4GL의 컴파일된 버전으로 이동시킵니다. 정의에 따르면 C 언어는 변수의 값을 결정하지 않은 상태에서는 지정될 때까지 변수를 초기화하지 않는 언어이므로, 대부분의 프로그래머가 알고 있는 것처럼 결정되지 않은 데이터 값이 응용프로그램에 치명적인 영향을 가져올 수 있습니다.

언어 상수

Informix-4GL의 한 가지 한계는 상수가 부족하다는 것입니다. 상수는 이름별로 참조할 수 있지만 프로그램에서 일단 설정된 이후 변경할 수 없는 프로그램 값입니다. 확실한 해결책은 상수의 자리에 변수를 사용하는 것입니다. Informix-4GL 언어가 컴파일 시 수정에서는 변수를 제한하지 않기 때문에 상수가 동작하도록 하는 것은 프로그래머에게 달려 있습니다. 상수 변수를 쉽게 식별하는 스타일 규칙을 사용하여 이 작업을 비교적 용이하게 수행할 수 있습니다. 가장 간단한 선택은 상수(공통적인 C 언어 규칙)의 이름을 정할 때 대문자만을 사용하는 것입니다. 다음 예에서 이 규칙을 보여줍니다.

```
- Create global 'constants'
-
globals
      define MAXRECS integer
      define MAXARR integer
end globals
main
      - Initialize CONSTANTS
let MAXRECS = 100
let MAXARR = 200
...
```

이 예에서 정수 변수 MAXRECS와 MAXARR은 전역 변수로 정의되어 있습니다. 메인 프로그램 블록이 시작될 때 이러한 변수는 특정값으로 초기화됩니다. 이 시점에서부터 프로그램의 다른 어떤 상수보다도 이 변수를 더 많이 사용할 수 있습니다. 다른 자료형의 상수도 확실하게 동일한 방식으로 선언되고 초기화되면 이와 같이 사용할 수 있습니다.

Informix-4GL의 흐름 제어문

Informix-4GL은 표준 반복 흐름 제어문 세트를 제공합니다. 반복적 루프를 제공하는 명령문인 for문과 while문을 사용할 수 있습니다. foreach문을 사용하여 SQL select문이 리턴하는 결과 세트를 통해 반복적 루프를 제공할 수도 있습니다. 표 8-10은 Informix-4GL에서 반복되는 흐름 제어문의 목록입니다.

표 8-10 Informix-4GL 제어문의 반복 순서

명령문	매개변수	기능
for	초기화 표현식, 제한 표현식, 선택적 단계 표현식	초기화 표현식에서 시작하는 인덱스 변수를 사용하여 루프를 수행하며, 인덱스 변수 값이 제한 표현식을 초과할 때까지 루핑합니다.
while	부울 평가 표현식	부울 표현식이 거짓으로 평가될 때까지 루프를 수행합니다.

<계속>

표 8-10 Informix-4GL 제어문의 반복 순서

명령문	매개변수	기능
foreach	커서 이름	커서 페치 작업이 조건을 찾을 없다고 리턴할 때까지 루프를 수행합니다.

for 명령문

for문을 사용하여 지정된 반복 세트에 대한 루프를 수행할 수 있습니다. 명령문은 초기화와 종료 값에 대한 복잡한 표현식을 평가할 수 있습니다. 사용 중인 인덱스 변수는 루프 내에서 변경 가능합니다. 하지만 구조화된 프로그래밍의 규칙을 엄격하게 준수하는 경우에는 변경할 수 없습니다. for 루프는 다음의 예와 같이 함수 호출의 리턴 값을 사용하여 초기화할 수 있습니다.

```
for n = get_val (a,b,c) to limit_val (a,b,c)
        do_this (a,b,c)
        do_that (a,b,c)

end for
```

이 예에서 get_val과 limit_val 함수는 루프에 대한 초기값 및 종료값을 구하기 위해 루프의 첫 번째 반복에서 호출됩니다. 그러나 루프 인덱스 값이 루프 내에서 변경될 수 있으므로, 이 표현식은 루프의 각 반복에서 다시 평가되어야 합니다. 루프 인덱스 변수가 루프 내에서 변경되고 있지 않은 경우, 다시 평가하는 것은 효율적이 아닙니다. 변수의 초기값 및 종료값을 저장하는 것은 루프 작업의 최적화를 제공합니다. 이러한 접근 방식의 예는 다음과 같습니다.

```
let x = get_val (a,b,c)
let z = limit_val (a,b,c)

for n = x to z
      do_this (a,b,c)
      do_that (a,b,c)
end for
```

Informix-4GL의 조건식

Informix-4GL의 조건식은 부울 표현식의 평가에 따라 제어 분기의 조건 순서를 제공합니다. Informix-4GL의 부울 표현식은 C 프로그래밍 언어를 사용할 때와 마찬가지로 평가됩니다. 영(0)이 아닌 값은 TRUE로 평가하고 영(0)인 값은 FALSE로 평가합니다. C 언어에 익숙한 프로그래머는 이 개념을 따르는 것이 용이합니다. 다음 명령문을 실행하면 "It's true"라고 표시됩니다.

```
if (1) then
      display "It's true"
end if
```

반면 다음 명령문은 출력이 없습니다.

```
if (0) then
      display "It's false"
end if
```

Informix-4GL에서는 프로그래머는 부울 표현식을 부정할 수 있으므로, 위의 명령문을 다음과 같이 작성할 수 있습니다.

```
if NOT (0) then
      display "It's false"
end if
```

이와 마찬가지로 영(0)이 아닌 값을 부정하면 다음의 표현식에서와 같이 FALSE 값이 생성됩니다.

```
if NOT(1) then
      display "It's true"
end if
```

위의 명령문은 아무 것도 출력하지 않습니다.

Informix-4GL에서 흥미로운 것은 부울 명령문의 표현식을 처리하는 것입니다. 부울 명령문에서는 부울 연산자와 상관없이 모든 표

현식을 평가하는 규칙이 있습니다. 따라서 다음과 같은 부울 명령문에서는 모든 표현식이 and 연산자를 사용하여 그룹화됩니다.

```
if (funct1(x,y,z) and funct2(x,y,z) and funct3(x,y,z)) then
      do_this_funct (x,y,z)
end if
```

여기서 명령문을 참으로 만들려면 모든 조건이 TRUE를 테스트하도록 해야 합니다. 이를 위해서는 왼쪽에서 오른쪽으로 명령문을 평가하고, FALSE를 평가하자마자 명령문의 평가를 중지해야 합니다. 하나의 FALSE 값이라도 명령문에 오류를 가져올 수 있으므로, 이렇게 해야 합니다.

Informix-4GL은 이러한 가정을 하지 않습니다. 부울 명령문은 명령문 그룹화에 대한 결론이 없는 상태에서 왼쪽에서 오른쪽으로 평가합니다. 괄호를 사용한 그룹화가 없으면, 결과를 다음 표현식에 적용하면서 표현식을 한 번에 두 개씩 평가합니다. 선행 명령문의 최적화를 위해 다음과 같은 명령문을 작성할 수 있습니다.

```
if ( func1 (x,y,z)) then
   if ( func2 (x,y,z)) then
      if ( func3 (x,y,z)) then
         do_this_funct (x,y,z)
      end if
   end if
end if
```

위의 코드 조각에서 표현식 “func1”이 실패하면 다른 어떤 명령문도 실행되지 않습니다. 동일한 상황이 표현식 “func2”에도 적용됩니다. 필요하지 않는 명령문은 실행되지 않을 가능성이 더 높기 때문에 이 상황은 어느 정도 레벨의 최적화를 제공합니다.

연결 연산자

Informix-4GL에서 문자열 연결 연산자는 콤마(,)입니다. 연산자는 다음 구문을 사용하여 두 개의 문자열을 연결할 수 있습니다.

```
let str1 = str2, str3
```

Informix-4GL에서 문자열은 항상 정의된 길이로 채워져 있습니다. 이렇게 채워진 문자열을 제거하려면 clipped 연산자를 사용해야 합니다. 다음 예를 보십시오.

```
define str1 char(40), str2 char(40), str3 char(40)

let str2 = "012345"
let str3 = "012345"

let str1 = str2, str3
```

이 예에서 str2와 str3을 합하여 채워진 길이가 목표 대상인 str1(40)의 정의된 길이보다 더 크기 때문에 마지막 명령문은 런타임 시 거부됩니다. 다음과 같이 clipped 연산자를 사용하면 이 문제를 해결할 수 있습니다.

```
define str1 char(40), str2 char(40), str3 char(40)

let str2 = "012345"
let str3 = "012345"

let str1 = str2 clipped, str3 clipped
```

이 명령문을 실행하면 클립된 길이가 12이므로 40byte 문자열인 str1에 쉽게 들어갈 수 있습니다.

Informix-4GL 응용프로그램에서 SQL 사용

Informix-4GL의 비프로시저 구성 요소 중 가장 중요한 요소는, SQL과 언어로 자연스럽게 통합되는 특성입니다. SQL이 올바르게 사용되는 경우, SQL은 응용프로그램을 프로그래밍하는 데 필요한 프로시저 코드의 양을 상당히 줄일 수 있습니다. 즉, 길고 장황한 프로시저 단계는 하나의 SQL문으로 압축될 수 있습니다.

실제로 Informix-4GL에는 사소한 제한사항이 있지만 모든 유효한

SQL문을 지원합니다. 이는 중요한 기능입니다. 하지만 이러한 기능을 십분 활용하기 위해서는 기본적인 데이터베이스가 잘 설계되고 표준화되어야 합니다. 그렇지 않을 경우, SQL의 장점은 사라지고 유용성 측면에서 프로그래머에게 ISAM 유형의 함수 호출과 다른 좋은 기능을 제공할 수 없게 됩니다.

소수의 일부 Informix-SQL문은 Informix-4GL에서 지원하지 않습니다. 이 명령문의 대부분은 관리적인 특성이 있고 dbaccess 프로그램에서 인식하고 처리하나 엔진에서는 해석하지 않습니다. 예를 들어, 다음 info문은 Informix-4GL에서 인식하지 못합니다.

```
info tables
info status for <table_name>
info indexes for <table_name>
info columns for <table_name>
```

이들은 관리적인 명령문입니다. 그러나 Informix-4GL 응용프로그램이 일부 데이터베이스 관리 기능을 처리하는 것은 일반적입니다. 이러한 인스턴스의 경우, 다음 select문을 사용하여 테이블에 대한 정보를 검색할 수 있습니다.

```
select tabname
from systables
where tabid > 99
```

모든 Informix 시스템 카탈로그 테이블에 tabid <= 99가 있기 때문에 이 명령문은 시스템 카탈로그 테이블을 제외한 현재 데이터베이스의 테이블 목록을 리턴합니다. 정보용 명령문인 info columns for table_name은 다음 명령문으로 구현될 수 있습니다.

```
select colname, coltype
from syscolumns, systables
where syscolumns.tabid  =  systables.tabid and
systables.tabname = <table_name>
```

컬럼이 NULL 값을 허용하는 경우, 컬럼 coltype에 대한 정수값은 특정 비트 패턴으로 마스킹됩니다. 4GL을 사용하여 이 마스크를 제거하려면 다음과 같이 명령문을 실행하십시오.

```
let pcoltype = pcoltype mod 256
```

그렇지 않은 경우, 명령문이 내장된 SQL 모듈 연산자를 사용할 수 있으면 다음과 같이 select문으로 통합할 수 있습니다.

```
select colname, mod (coltype, 256)
from  syscolumns, systables
where syscolumns.tabid  =  systables.tabid and
      systables.tabname  =  table_name
```

이와 마찬가지로 정보 명령문인 info indexes for table_name은 다음 SQL문으로 구현될 수 있습니다.

```
select idxname
from  systables, sysindexes
where systables.tabid  =  sysindexes.tabid and
      systables.tabname  =  table_name
```

이 명령문은 where절에 사용된 테이블 이름에 대한 인덱스의 모든 이름을 리턴합니다.

SQL문을 사용하여 유형 변환

Informix-4GL에서 값을 지정하는 다른 명령문과 마찬가지로 SQL문은 타당한 경우에 유형 변환을 지원합니다. 문자열이 전체 값을 포함할 정도로 충분히 길면, 어떠한 경우에도 모든 Informix 자료형을 문자열로 변환할 수 있습니다.

select문은 데이터베이스에서 데이터를 검색하고, into절을 사용하여 값을 호스트 변수로 지정합니다. 호스트 변수에 있는 것보다 더 적은 수의 데이터 요소를 검색하는 것은 오류가 아닙니다. Informix-4GL은 이를 오류가 아니라 경고로 인식합니다.

이 동작은 이 텍스트 후반에서 동적 SQL문을 사용하여 데이터를 검색할 때 사용됩니다.

데이터를 검색하는 중에 데이터를 변환하는 것은 변수 지정 중에 수행하는 것만큼 자유롭게 발생합니다. 다음에 예가 나와 있습니다.

```
define str1 char(20), str2 char(20), str3 char(20)
- column int_col contains an integer value
- column dec_col contains a decimal value
- column date_col contains a date value

select int_col, dec_col, date_col
into str1, str2, str3
from sample_table
```

위의 예에서 모든 자료형은 기본값에 따라 변환됩니다. 그 결과로 얻은 값은 문자열의 길이가 적절하다면 문자열에 배치됩니다. 문자열이 충분히 길지 않은 경우, 별표가 변수 문자열의 크기까지 변수에 배치됩니다.

Informix-4GL에서의 데이터베이스 바인딩

Informix-4GL의 가장 유용한 기능 중 하나는 like 키워드를 사용하여 프로그램 변수 정의를 데이터베이스 테이블 컬럼으로 바인딩하는 기능입니다. 이 기능을 사용하여 다음과 같이 간단한 명령문 세트로 데이터베이스에서 검색 중인 데이터의 무결성을 확인할 수 있습니다.

```
database mycust

define customer_rec record like customer.*

select customer.*
into customer_rec.*
where cust_numb = 1
```

이 코드를 사용하면 자료형의 비호환성에 대한 염려 없이 고객 번호 1에 대한 고객 레코드를 프로그램 변수로 검색할 수 있습니다. 데이터베이스 명령문은 데이터 바인딩 define문 이전에 나타나야 한다는 점에 유의하십시오.

데이터 바인딩 기능은 Informix-4GL 컴파일러가 컴파일 시 데이터베이스 테이블에 대한 스키마를 읽고, 변수의 자료형 및 길이를 결정하며, 서수의 위치와 유형으로 데이터베이스 테이블 컬럼 정의와

일치하는 변수에 대한 정의를 생성하게 함으로써 구현됩니다. 컴파일러는 레코드 정의를 적절한 구성 요소로 확장합니다. 소스 코드 모듈에서는 데이터베이스 명령문이 define like문보다 앞에 와야 한다는 점에 유의하십시오. 선행하는 정의에 의해 생성된 결과의 C 코드는 다음과 같습니다.

스키마 사용:

```
create table customer (
      first_name char(15),
                middle_i char(1)
                last_name char(15),
                phone_number char(10),
                address1 char(20),
                address2 char(20) )
```

Informix-4GL 선언:

```
define customer_rec record like customer.*
```

생성된 C 코드:

```
struct{
      char first_name[16];
      char middle_i [2];
      char last_name [16];
      char phone_number [11];
      char address1 [21];
      char address2 [21];
}     customer_rec;
```

그러나 이 기능은 자료형 또는 순서에서 데이터 테이블의 구조가 변경되었을 때 문제가 발생할 수 있습니다. Informix-4GL은 가능할 때마다 자료형을 변환하기 때문에 유효한 데이터가 레코드로 읽혀지고, 오류는 분명하지 않을 수도 있습니다. 다음 예에서, 초기에 customer 테이블을 다음의 스키마를 포함하도록 정의했다고 가정해 봅시다.

스키마 1

```
create table customer (
                  first_name char(15),
                  middle_i char(1)
                  last_name char(15),
                  phone_number char(10),
                  address1 char(20),
                  address2 char(20) )
```

그러나 이 스키마는 다음 구조로 변경됩니다.

스키마 2

```
create table customer (
                  first_name char(15),
                  last_name char(15),
                  phone_number char(10),
                  address1 char(20),
                  address2 char(20) )
```

스키마 1을 사용할 때 이 프로그램 예를 컴파일하고, 스키마 2를 사용할 때 이 예를 실행한 경우, 스키마 1 정의와 같이 정의된 데이터 레코드는 잘못된 값을 수신합니다. 이로 인해, 프로그램에 오류가 항상 발생하는 것은 아닙니다. 프로그램은 사용자 성의 첫 자를 레코드 구성 요소 `middle_i`에 지정하고 고객의 `last_name` 필드를 전화 번호 필드에 지정함으로써 자동으로 작업을 진행합니다. 첫 번째 `address1` 레코드 구성 요소는 데이터베이스 레코드의 `address2` 부분을 수신하지만, 프로그램 레코드의 `address2` 필드에는 아무 영향이 없습니다. 즉, 이 값은 미결정 값이 됩니다. Informix-4GL에서는 모든 데이터 값을 NULL로 종료한다고 가정하므로 이 레코드의 구성 요소를 처리하려는 경우 런타임 시 경우에 따라 실패할 수도 있습니다.

prepare 명령문

Informix-4GL에서 prepare문의 목적은 두 가지입니다. 첫 번째, Informix-4GL에서 prepare문은 동적 SQL을 수행하는 데 사용됩니다. prepare문을 사용하여 런타임 시 동적으로 쿼리를 빌드할 수 있습니다.

그러나 prepare문은 엔진과 프론트엔드 사이의 프로세스 간 통신(IPC) 트래픽을 줄이는 데 사용되므로 최적화 도구로 인식되기도 합니다. prepare문은 실행 전에 최적화되기 때문에 실행할 때마다 연속적으로 명령문을 다시 최적화해야 할 필요가 없습니다.

동적 SQL을 작성하기 위해 prepare문은 문자열 변수를 매개변수로 가져올 수 있습니다. 이 문자열 변수 내에서는 실제로 유효한 SQL문이 모두 지원됩니다. 예외로는 close, declare, execute, load, open, prepare 및 unload문이 있습니다.

prepare문은 전달된 문자열을 단순히 패키지 처리한 뒤 평가하기 위해 엔진으로 보냅니다. 즉, prepare문은 SQL문을 검토하지는 않습니다. 프론트엔드가 아니라 데이터베이스 엔진이 모든 구문 분석 및 최적화를 수행합니다.

Informix-4GL에서 아직 지원하지 않는 구문을, 데이터베이스 엔진에서 지원하는 경우가 있습니다.

이 문제에 대한 해결책은 다음과 같이 SQL문을 문자열에 배치하고 prepare문을 사용하여 엔진에 전달하는 것입니다.

```
let qstr = "execute procedure myproc( )"
prepare ex_stmt from qstr
execute ex_stmt
```

위의 예에서는 내장 프로시저 myproc을 호출하려고 시도해도 사용 중인 Informix-4GL 버전은 현재 execute procedure문을 지원하지 않으므로 호출되지 않습니다. 따라서 명령문을 문자열에 배치한 후 prepare문을 사용하여 문자열을 엔진으로 전달해야 합니다. 엔진이 명령문을 인식하기 때문에 명령문을 처리하고 명령문 ID를 Informix-4GL 프로그램으로 리턴하는 데에는 문제가 없습니다. 그러면 Informix-4GL 프로그램은 이 명령문 ID에 대해 execute문을 발행합니다. 내장 프로시저가 데이터를 검색하면 해당 명령문 ID에 커서를 선언해야 한다는 점에 유의하십시오. 그렇지 않은 경우, 명령문 실행 시 런타임 오류가 발생합니다.

prepare 명령문을 사용하여 최적화

Informix-4GL에서 쿼리를 실행할 때, 데이터베이스 엔진은 쿼리 구문을 분석하고, 유효성과 구문의 타당성을 검토하며, 쿼리를 최적화하고, 필요한 경우 임시 테이블을 생성하며, 데이터 파일에 액세스한 후, 데이터 검색을 시작해야 합니다. 쿼리를 실행할 때마다 이 모든 단계가 필요한 것은 아닙니다.

데이터베이스 엔진에 처음으로 쿼리가 발생하면 쿼리 구문을 분석해야 합니다. 런타임까지 SQL 데이터베이스 객체를 검토하지 않기 때문에, 런타임까지는 쿼리의 완전한 구문 분석 및 최적화를 수행할 수 없습니다.

쿼리를 점검할 때 데이터베이스 엔진은 구문이 올바른지 확인하기 위해 SQL문을 검토합니다. 이 단계는 데이터베이스 엔진이 명령문의 데이터베이스 객체의 위치를 결정하기 위해서도 반드시 필요합니다. 데이터베이스 객체를 검토한 후 데이터베이스를 검색하여 이 객체가 존재하는지 여부를 결정합니다.

다음 단계에서 엔진은 쿼리를 최적화합니다. Informix 버전 4.0 이상의 엔진은 비용 기반의 옵티마이저를 사용합니다. 이 옵티마이저는 해당 액세스 경로의 다양한 구성 요소에 대한 비용과 함께 가능한 모든 액세스 경로를 검토함으로써 쿼리를 최적화합니다. 옵티마이저는 가장 적은 비용의 액세스 경로 즉, 가장 효율적인 액세스를 선택합니다. 이 단계에서 쿼리의 각 실행 간에는 검토된 쿼리 요소가 변경되지 않을 가능성이 높으므로, 반드시 이 단계를 반복할 필요는 없습니다. 이 단계를 생략하는 데 prepare문을 사용할 수 있습니다.

트랜잭션 처리 환경에서는 쿼리 준비에 사용된 CPU 주기가 다른 응용프로그램의 처리에 사용되었을 수 있는 CPU 주기이므로 prepare문을 사용한 최적화 또한 중요합니다.

SQL 처리: 프론트엔드 대 백엔드 활동

초기 Informix 제품은 두 가지 프로세스 아키텍처를 사용하도록 설계되었습니다. 버전 6.0 제품에서 이것이 약간 변경되기는 했지만, 프론트엔드와 백엔드 처리의 개념은 여전히 유효합니다. 응용프로그램의 프론트엔드는 응용프로그램을 제어하고, 백엔드는 데이터베이스 상호 작용을 수행합니다. SQL문의 특정 부분은 응용프로그램에서 처리하고 다른 부분은 엔진에서 처리합니다.

Informix-4GL에서 발생된 모든 SQL문은 몇 가지 단계로 처리됩니다. 먼저, 구문을 분석하여 SQL 구문의 유효성을 결정합니다. 유효한 명령문을 사용 중인지 확인하기 위해 Informix-4GL 컴파일러는 일부 커서 구문을 점검할 수 있습니다. 하지만 대량의 SQL문 구문은 데이터베이스 엔진에 의해 분석됩니다.

데이터베이스 엔진의 구문 분석기는 SQL문의 다양한 하위 절이 올바른지 검토합니다. 이러한 명령문이 일정하게 참조하는 객체는 몇 가지 기준에 따라 평가됩니다. 먼저, 현재 데이터베이스에 이러한 객체가 존재하는지를 결정해야 합니다. 그런 후 현재 절에서 유효한 데이터베이스 객체인지를 결정해야 합니다. 다음으로 표현식을 분석하고, 자료형 일치를 평가한 후, 모든 필요한 변환을 수행합니다. 마지막으로, 최적화를 수행해야 합니다. 데이터베이스 엔진 옵티마이저는 다양한 액세스 경로를 평가하고 최적의 액세스 경로를 선택합니다.

prepare문과 declare문을 사용하여 최적화

SQL문이 반복적으로 실행되면, 명령문이 사용되기 전에 prepare문을 사용하여 명령문 구문을 분석하고 최적화함으로써 성능을 향상시킬 수 있습니다. 또한 select문의 경우, declare문을 사용하여 유사한 최적화를 수행할 수 있습니다.

SQL문을 한 번만 실행하려면, prepare 또는 declare문을 사용하여 최적화할 수 없습니다. 다음 예에서는 이러한 내용을 보여주고 있습니다. 이 예에서는 응용프로그램 시작 프로시저 동안 코드가 한 번만 실행된다고 가정합니다.

```
    -build the query
prepare qstmt from "select application_name",
                   "from control_file",
                   "where control_code = 1"

declare curs1 for qstmt

    - fetch the data
fetch curs1 into appl_name
```

위의 예에서는 전체 쿼리의 내용이 컴파일 시 알려진다고 가정합니다. 이 유형의 단일 쿼리에 대해 prepare, declare, fetch를 수행하는 데 따른 이점은 없습니다. 단지 다음을 실행하는 것만이 효율적입니다.

```
-execute the query
select application_name
into appl_name
from control_file
where control_code = 1
```

이 명령문은 더 적은 라인의 코드를 필요로 하며 더 명확합니다. 여기서 select문을 읽어 무엇을 선택하고 있는지를 이해하는 것이, 커서를 선언한 코드에서 제거된 단독 fetch문을 읽어 무엇을 검색하고 있는지를 이해하는 것보다 쉽습니다.

화면에 입력 중인 데이터의 유효성을 확인하는 데 사용하고 있는 SQL문의 경우, 하드 코드 SQL문을 prepare된 SQL문으로 변경한다 하더라도 효율적이지는 않습니다. 그러나 백만 개의 행을 스캔하고 행마다 기타 SQL문을 실행해야 하는 보고서의 경우, 실행을 반복하기 이전에 명령문을 prepare하는 것이 올바른 방법입니다. 다음의 코드 샘플은 이 점을 보여줍니다. 이 예는 보고서 본문 내의 prepare문 사용을 보여줍니다.

```
on every row

        select fld_stats
        into fld_stats
        from field_stats
        where empl_numb = pempl_numb
```

이 쿼리는 보고서로 전달된 각 세트마다 실행됩니다. 보고서가 많은 행을 읽고 처리하고 있는 경우, 이 쿼리를 실행 및 구문 분석하면 보고서의 실행 시간이 느려집니다. 이 프로세스의 속도를 높이기 위하여 명령문은 다음과 같이 prepare될 수 있습니다.

```
first page header

declare fld_stats_curs cursor for
```

```
    select fld_stats
    into fld_stats
    from field_stats
    where empl_numb = pempl_numb
  ....

on every row

        open stmt 1
        fetch stmt1 into fld_stats
```

이 예에서는 명령문이 첫 번째 페이지 헤더 블록에 준비되며, 한 번만 실행됩니다. 일단 이 코드 블록이 실행되면, 보고서로 전달된 모든 행마다 on every row 블록의 최적화된 명령문이 실행됩니다. 여기서는 데이터베이스 객체 및 최적화를 평가하고, 관련 IPC 트래픽을 검사하는 불필요한 단계를 생략합니다.

일반적으로 한 번에 한 행에서 작업하도록 실행되는 SQL update 또는 delete문의 경우, prepare문을 사용하여 이러한 작업을 최적화할 수 있습니다. 다양한 기준에 따라 많은 행을 업데이트해야 하는 배치 업데이트 프로그램의 경우, prepare문을 사용하면 현저하게 성능이 향상됩니다. 이러한 예는 다음에 나와 있습니다.

```
on every row

        if (do_update) then
        update customer
        set customer.addr1 = customer_rec.addr1,
        customer.addr2 = customer_rec.addr2

  ....
```

이 예에서는 응용프로그램의 다른 부분의 플래그 세트에 따라 배치 업데이트를 보고서 프로그램을 사용하여 수행하고 있습니다. 많은 행을 처리하고 있다고 가정할 경우, 이 업데이트의 최적화는 이 프로그램의 실행 속도에 상당히 큰 영향을 줄 수 있습니다. 다음 코드는 이러한 최적화를 보여줍니다.

```
first page header

   prepare upd_stmt
from "update customer set customer.addr1 = ?, ",
     "customer.addr2 = ?" ,
     "where customer_num = ?"

...

on every row

if (do_update) then
      execute upd_stmt using customer_rec.addr1,
                             customer_rec.addr2,
                             customer_rec.cust_numb,

...
```

위의 예에서 페이지 헤더 블록은 업데이트 명령문을 준비하는 데 사용됩니다. on every row문을 실행할 때, do_update 플래그를 검사한 후 그 값이 TRUE인 경우, upd_stmt 명령문 ID를 통하여 업데이트 명령문이 실행됩니다. 많은 수의 행을 처리하는 대형 배치 프로그램에서는 현저하게 성능이 향상될 수 있습니다.

최적화는 prepare문 사용으로 제한되지 않아야 합니다. 적절한 인덱스의 존재, 정렬 기준 및 인덱스의 중복 등 많은 다른 요인들이 쿼리의 성능에 영향을 줄 수 있습니다. 데이터베이스 액세스를 병목 현상으로 본다면 이 요인을 검토해야 합니다.

Informix-4GL의 커서

한 번에 한 세트를 처리하는 SQL의 세트 중심 명령문을 사용하여 Informix-4GL의 프로시저문과 같이, 커서는 한 번에 한 행을 처리하는 연속 언어를 조정하는 방법입니다. select문의 경우, 커서는 select문이 참조하는 활성 데이터 세트로의 포인터를 제공합니다.

foreach 명령문

foreach문은 프로시저문의 등록 정보 일부와 비프로시저 SQL문의 등록 정보를 조합합니다. foreach문은 커서와 커서 내의 모든 세트를 관련시키므로, 이러한 측면에서 foreach문은 커서의 확장입니다. 그러나 또한 while 루프의 구조와 유사한 루프 구조를 작성하는데 사용하기도 합니다. 발견되지 않은 조건을 만족시킬 때까지 루프 처리를 하는 것이 while 루프입니다. 이 지점에서 프로그램 실행은 foreach문의 end 다음의 첫 번째 명령문에서 시작합니다.

발견되지 않은 조건을 발견할 때까지, 계속하여 루프 처리하는 while 루프 앞에 오는 open된 커서가 foreach 대신 사용됩니다. 그러나 foreach문은 이러한 논리의 프로그래밍에 더욱 매력적인 대안입니다. foreach문은 다수의 행을 스캔하는 배열을 로드하여 보고서를 생성하는 유용한 구조를 제공합니다.

SQL문에서 사용된 데이터 변수

SQL 표현식에서 사용된 Informix-4GL 변수는 컴파일 시 자료형 및 길이에 대해 평가됩니다. 이러한 변수는 런타임의 내용을 점검합니다. SQL문에서 사용할 값은 몇 가지 대안 중 하나를 사용하여 런타임 시 제공할 수 있습니다. 명령문은 SQL문의 where절에서 매개변수로 정의된 변수를 사용하여 커서로 선언할 수 있습니다. 그렇지 않으면, 런타임 시 대체할 값의 자리 표시자로서 '?'를 사용하여 명령문을 준비할 수 있습니다.

한 가지 방법이 다른 방법에 대하여 실질적인 성능상의 이점을 가지고 있지는 않습니다. 내부적 구현은 실제로 동일합니다. 고려할 유일한 문제는 프로그래밍의 편의성과 양식입니다.

런타임 값을 제공하는 한 가지 방법은 단순히 커서 선언에서 프로그래밍 변수 참조를 포함시키는 것입니다. 이 방법에 대한 예는 다음과 같습니다.

```
declare cursor c1 cursor for

        select *
        from customer
        where cust_numb = customer_rec.cust_numb
```

```
open c1
fetch c1 into customer_rec.*
```

이 예에서는 Informix-4GL 변수 레코드 요소를 참조하는 SQL where절의 필터 기준을 사용하여 커서를 선언합니다. 이 레코드 요소는 커서에 바이딩됩니다. 커서를 열었을 때 호스트 변수의 내용을 검토하여 필터 기준 대신 사용합니다. 이 명령문은 using절을 지원하지 않으므로 이 방법이 매개변수를 foreach문으로 전달하도록 지원하는 유일한 방법입니다.

다음 예에서는 필터 기준에 자리 표시자를 사용하여 커서가 작성됩니다.

```
prepare s1 cursor for "select * ",
                      "from mycust",
                      "where cust_numb = ?"

declare c1 cursor for stmt1

open c1 using customer_rec.cust_numb
```

위의 예에서 명령문 ID는 먼저 select문에 설정된 후 해당 명령문 ID에 커서를 선언합니다. 마지막으로 using절을 사용하여 커서를 엽니다. 커서를 열기 위해서는 커서가 필터 기준에 액세스해야 하므로 using절은 이 예에서 필수입니다. (현재 Informix-4GL 버전에서는 커서 선언이 소스 코드 모듈에서의 커서 사용보다 선행되어야 함에 주의하십시오.)

Informix-4GL에서 데이터 변수를 바인드하는 올바른 방법이 하나만 있는 것은 아닙니다. 표시된 두 방법 모두 유효하며 폼에서 모듈로 인식될 수 있습니다. 첫 번째 예에서는 약간 더 적은 양의 코딩이 필요하나, 두 번째 예는 커서 필터에 대한 기준을 발견할 수 있는 open문이 사용되어 결과가 분명합니다. 어떤 방법을 사용할 것인지는 프로그래머에게 달려 있습니다.

Informix-4GL의 커서

커서는 데이터베이스 엔진의 데이터 제어 구조를 참조하는 프론트엔드의 데이터 구조입니다. 엔진에서 이 데이터 제어 구조를 사용하려면 프로세스 간 통신(IPC) 기능을 사용해야 합니다. 이 IPC 트

래픽의 감소는 커서 사용을 최적화하는 하나의 방법입니다.

커서로 인한 활동은 응용프로그램의 프론트엔드와 백엔드 둘 모두에서 실행됩니다. declare문을 통해 커서가 작성되면 응용프로그램 데이터 영역에 데이터 공간이 할당됩니다. 이 데이터 공간은 커서에 대한 정보(데이터베이스 객체 또는 커서가 가리키는 객체, 커서가 검색하는 테이블 컬럼, 명령문 내에서 매개변수에 발생할 모든 데이터 바인딩 및 명령문이 검색할 데이터 값)가 들어 있는 구조입니다.

커서를 열면 SQL문 구문이 분석되며 모든 치명적인 오류가 프로그램에 보고됩니다. 데이터베이스 객체의 가용성을 검토하고, Informix 비용 기반의 옵티마이저를 사용하여 쿼리에 대한 많은 액세스 경로를 평가합니다. 또한 데이터에 대한 액세스 경로를 결정합니다. 임시 테이블이 필요한 경우에는 이 테이블을 엽니다.

foreach 또는 fetch문을 사용하여 데이터베이스에서 데이터를 검색하면, 특정 양의 버퍼링이 발생합니다. 버퍼링은 데이터베이스 엔진에서 발생하고 제한된 범위로 응용프로그램에서도 발생합니다. 이 버퍼는 일반적으로 크기가 약 1K정도이고 프로그래머가 변경할 수 없습니다.

커서 유형

Informix-4GL에는 시리얼(또는 보통) 커서, 스크롤 커서, 업데이트 커서 및 삽입 커서의 네 가지 유형이 있습니다. 이러한 커서에 대한 설명은 다음과 같습니다.

시리얼 커서는 가장 일반적인 커서 유형입니다. 이 커서는 세트 시작에서 시작하여 세트 끝까지 진행되는 시리얼 방식으로 select 세트를 읽습니다. 이 커서는 세트를 통하여 앞뒤로 이동할 수 없습니다.

시리얼 커서가 직접 인덱스를 통하여 테이블이나 테이블 세트에 액세스 할 수 있는 반면, 스크롤 커서는 임시 테이블 생성을 요구합니다. 스크롤 커서는 데이터 세트를 통해 앞뒤로 이동할 수 있으므로 특정 시점에서 데이터의 스냅샷을 요구합니다. 세트의 시작 또는 세트의 현재 위치와 관련하여 앞뒤로 이동할 수 있습니다. 임시 테이블을 필요로 하므로 이 커서 유형을 사용할 경우, 어느 정도 성능 저하를 예측할 수 있습니다.

스크롤 커서

다음 코드는 스크롤 커서의 사용을 보여줍니다. 이 응용프로그램은 고객 테이블의 화면 폼을 표시하고 사용자가 테이블을 찾아볼 수 있도록 합니다. 사용자는 레코드를 앞뒤로 이동할 수 있고, first 옵션을 사용하여 select 세트의 시작으로 점프하거나 end 옵션을 사용하여 select 세트의 끝으로 점프할 수 있습니다. 현재 세트 내에서의 이러한 이동은 스크롤 커서를 사용하여 수행됩니다.

```
        define customer_rec record like customer.*
        define recno integer

        open form f1 from "customer"
        display form f1

- select the key fields
declare c1 scroll cursor for

        select  *
        from customer

open c1
fetch c1 into customer_rec.*
display customer_rec.* to customer.*

      menu "Customer"
          command "Next" "Display next record"
            fetch relative 1 c1 into customer_rec.*
            display customer_rec.* to customer.*
          command "Previous" "Display previous record"
            fetch relative -1 c1 into customer_rec.*
            display customer_rec.* to customer.*

          command "First" "Display first record"
            fetch first c1 into customer_rec.*
            display customer_rec.* to customer.*

          command "Last" "Display last record"
            fetch last c1 into customer_rec.*
```

```
        display customer_rec.* to customer.*

      command "Goto" "Go to a specific record"
        prompt "Enter record number:" for recno
        fetch absolute recno c1 into customer_rec.*
        display customer_rec.* to customer.*

      command "Quit" "Quit this menu"
        exit menu

  end menu
```

위의 예에서 scroll cursor가 고객 레코드에 작성됩니다. 그런 후 스크롤 커서를 열고, fetch문을 사용하여 첫 번째 레코드를 검색합니다. (기본적으로, 스크롤 커서에 레코드 번호 규정자가 없는 fetch문은 fetch next문 즉, 다음 레코드를 페치하도록 요청하는 명령문으로 간주됩니다.) 이 레코드는 화면에 표시됩니다.

다음으로 메뉴를 표시하여 사용자가 특정 레코드로 이동하는 데 next, previous, first, last의 옵션을 선택하거나 quit 옵션을 사용하여 메뉴를 종료할 수 있게 합니다. next 및 previous 옵션의 경우, fetch 관련 명령문은 next 옵션에 대해서는 +1을, previous 옵션에 대해서는 -1을 사용하여 실행됩니다. 이렇게 하여 커서는 +1인 경우에는 다음 레코드로, -1인 경우에는 이전 레코드로 이동합니다. 이러한 커서의 이동은 현재 커서 위치와 관련되어 있습니다. 그렇지 않으면, fetch previous문을 사용하여 이전 레코드를 페치할 수 있거나 또는 fetch next문을 사용하여 다음 레코드로 페치할 수 있습니다. first 또는 last 옵션은 단순히 첫 fetch first문을 실행하여 첫 번째 레코드를 검색하거나 fetch last문을 실행하여 마지막 레코드를 검색합니다.

goto 옵션을 사용하면 사용자가 현재 세트의 레코드 번호를 지정하고 해당하는 레코드 번호로 이동할 수 있습니다. 이 경우, 이동은 세트의 시작에 상대적이므로 fetch absolute문이 사용됩니다. 활성 세트의 현재 위치에 상대적으로 이동시키는 것이 바람직한 경우, fetch relative문을 사용할 수 있습니다. 이 두 명령문은 모두 인수로서 정수 상수 또는 정수 표현식을 가져옵니다.

스크롤 커서의 문제

스크롤 커서의 구현과 연관된 문제에는 두 가지가 있습니다. 이전의 예에서 이미 이 두 가지 문제가 발생했으나, 둘 모두 쉽게 해결되었습니다.

첫 번째 문제는 사용하지 않는 데이터의 문제입니다. 스크롤 커서를 처음으로 실행하면, 임시 테이블이 생성되며 select 세트의 현재 상태에 대하여 스냅샷을 가져옵니다. 이 스냅샷은 커서가 존재하는 동안 사용됩니다. 즉, 사용자가 세트에서 행을 찾아보는 동안 다른 사용자가 데이터를 변경할 수 있다는 의미입니다.

두 번째 문제는 임시 테이블의 사용과 연관된 것입니다. 선택된 세트의 크기가 크고 행의 수가 많은 경우, 임시 테이블의 크기가 상당히 커질 수 있습니다. 이로 인해 많은 시스템 자원이 사용되며 작성하는 데 많은 시간이 걸릴 수 있습니다.

이러한 문제의 해결책은 목표 테이블 또는 테이블에서 기본 키를 검색하는 커서와 나머지 데이터를 검색하는 커서, 이 두 가지 커서를 사용하는 것입니다. 다음의 예에서는 이러한 접근 방식을 보여줍니다.

```
        define customer_rec record like customer.*
        define recno integer

        open form f1 from "customer"
             display form f1

- select the key fields
declare c1 scroll cursor for
        select  customer_num
        from   customer

declare c2 scroll cursor for
    select *
    from customer
    where customer_num = customer_rec.cnum

open c1
open c2
```

```
fetch c1 into customer_rec.customer_num
fetch c2 into customer_rec.*

display customer_rec.* to customer.*

menu "Customer"

        command "Next" "Display next record"
          fetch relative 1 c1 into customer_rec.customer_num
          open c2
          fetch c2 into customer_rec.*
          display customer_rec.* to customer.*

        command "Previous" "Display previous record"
          fetch relative -1 c1 into customer_rec.customer_num
          open c2
          fetch c2 into customer_rec.*
          display customer_rec.* to customer.*

        command "First" "Display first record"
          fetch first c1 into customer_rec.customer_num
          open c2
          fetch c2 into customer_rec.*
          display customer_rec.* to customer.*

        command "Last" "Display last record"
          fetch last c1 into customer_rec.customer_num
          open c2
          fetch relative c2 into customer_rec.*
          display customer_rec.* to customer.*

        command "Goto" "Go to a specific record"
          prompt "Enter record number:" for recno
          fetch absolute recno c1 into customer_rec.customer_num
          open c2
          fetch c2 into customer_rec.*
          display customer_rec.* to customer.*

        command "Quit" "Quit this menu"
```

```
            let cont = "N"
            exit menu

end menu
```

위의 예에서 두 개의 커서가 선언됩니다. 하나는 스크롤 커서이고 나머지는 단순 커서입니다. 스크롤 커서가 기본 키만을 검색하며 각 행을 탐색할 때, 나머지 데이터 레코드는 단순 커서를 사용하여 데이터베이스로부터 검색됩니다. 이 때 데이터가 화면에 표시됩니다.

업데이트 커서

업데이트 커서는 시리얼 커서와 유사하게 행 검색을 수행합니다. 즉, 업데이트 커서는 행을 검색하면서 데이터 세트를 통해 연속적으로 이동합니다. 이 커서는 행을 뒤로 이동할 수 없으며 한 번에 한 행만 앞으로 이동할 수 있습니다.

업데이트 커서는 두 가지 측면에서 시리얼 커서와 다릅니다. 첫 번째로 업데이트 커서는 검색 시 각 행의 공유 로크를 가져온 후 다음 행으로 이동할 때 로크를 해제합니다. 두 번째로 업데이트 커서에는 사용과 연관된 많은 제한사항이 있습니다. 일반적인 제한사항으로 업데이트 커서의 사용 목적은 단일 데이터베이스 테이블 행을 업데이트하는 것이므로 기본적인 select문은 하나의 테이블만 참조할 수 있고 또한 이 테이블을 직접 참조해야 한다는 사실입니다. select문에는 order by절이 있을 수 없습니다. 이것은 목표 테이블 대신 정렬 테이블이 작성되어 읽혀져야 할 수도 있기 때문입니다.

업데이트 커서를 사용할 때 한 가지 중요한 이점은 데이터베이스 업데이트를 수행할 때 얻는 성능상의 장점입니다. 현재의 커서 위치를 사용하여 레코드를 업데이트할 수 있으므로 업데이트할 레코드를 찾는 비용을 들이지 않아도 됩니다.

업데이트 커서를 때로는 로크 커서라고 합니다. 업데이트 커서는 한 행을 액세스할 때 그 행에 공유 로크를 설정합니다. 공유 로크를 사용하면 다른 사용자가 해당 행을 읽을 수 있지만 공유 로크를 획득한 사용자가 로크를 해제할 때까지 행을 업데이트할 수 없습니다. 커서가 다음 행으로 이동하면 이 로크가 해제됩니다.

삽입 커서

언급했던 다른 커서와는 달리 삽입 커서는 데이터베이스에서 행을 검색하지 않습니다. 삽입이라는 이름이 의미하듯이 이 커서는 행을 데이터베이스에 삽입하는 데 사용됩니다. 그리고 단일 행을 데이터베이스 테이블에 삽입하는 단순 insert문과는 달리, 삽입 커서는 데이터베이스 엔진으로 전송한 단일 삽입 요청으로 복수의 행을 삽입하면서 테이블에 삽입하기 전에 행을 버퍼로 지정합니다.

단순 삽입 명령문을 사용하면, 단일 행이 IPC 기능을 통해 데이터베이스에 작성하는 데이터베이스 엔진으로 전송됩니다. 프론트엔드 응용프로그램이 행을 엔진으로 전송할 때, Informix-4GL 프로그램의 경우, 엔진으로부터 행을 수신했다는 응답을 기다립니다. 여러 행을 데이터베이스로 삽입하는 프로세스의 속도를 늦추는 것은 바로 이 응답과 IPC 기능에 액세스하는 데 소요되는 시간에 해당합니다. 그래서 이 단계는 삽입 커서를 사용하여 주로 제거됩니다.

행은 삽입 커서를 사용하여 엔진으로 전송되기 전에 프론트엔드 프로세스에서 버퍼가 지정됩니다. 프론트엔드 버퍼의 크기는 보통 약 1K이며, 30byte 행의 경우 약 330개의 행이 엔진으로 전송되기 전에 버퍼가 지정되도록 합니다. 이 때 행에 대해 330개의 별도의 응답을 기다리는 대신 버퍼가 지정된 행의 전체 세트에 대하여 하나의 응답만 발행됩니다. 따라서 20,000개의 행을 삽입하는 프로세스에서는 단지 60개의 버퍼가 전달되고 60개의 응답이 리턴됩니다. 이에 비해 단순 삽입 명령문을 사용하면 20,000개의 버퍼가 전달되며 20,000개의 응답이 리턴됩니다. 이는 파이프 트래픽이 98% 감소하는 결과를 가져오고, 약 40%의 성능 향상 효과를 가져옵니다.

데이터의 배치 로드와 같은 프로그램의 단일 실행 중에 다수의 행을 데이터베이스에 입력할 경우 삽입 커서가 유용합니다. 단순한 데이터 입력 프로그램과 함께 사용되고 있는 삽입 커서는 아마도 현저한 성능 향상 효과를 가져오지 않을 것입니다. 데이터 입력 프로그램에서 사용자가 데이터를 입력하는 데 필요한 시간이 행을 버퍼링하여 얻는 성능 향상의 가능성을 상쇄시킵니다. 응용프로그램은 사용자가 행을 입력할 때까지 기다려야 하기 때문에 데이터베이스의 경우 행을 더 빠르게 작성할 수 없습니다. 삽입 커서 구현의 예는 다음과 같습니다.

```
define journal_rec record like journals.*

declare ins_curs cursor for insert into journal_hist ←
        values ( journal_rec.* )
declare j_curs cursor for select * from journals where←
        journal_data < "1/1/1993"

open ins_curs

foreach j_curs into journal_rec.*

        put ins_curs

end foreach

delete from journals where journal_date < "1/1/1993"

close ins_curs
```

위의 예에서 두 개의 커서가 선언되었습니다. 한 커서는 journals 테이블에서 특정 일 이전의 행을 읽습니다. 이 읽혀진 행은 삽입 커서를 사용하여 실행 기록 테이블(journal_hist)에 입력됩니다. for each 루프 종료 후에는 삽입 커서가 닫혀집니다. 이 커서는 버퍼에 남은 행을 플러시합니다. 삽입 커서를 닫는 것은 중요합니다. 그렇지 않으면 마지막 버퍼에 남아 있는 행을 데이터베이스로 출력할 수 없습니다.

삽입 커서를 사용하면 데이터 무결성 문제가 발생하기도 합니다. 프론트엔드에 버퍼가 지정된 행은 데이터베이스 엔진의 데이터 무결성 기능의 제어를 받지 않습니다. 머신에 오류가 발생하는 경우, 이러한 행이 상실됩니다. 이전 예에서 언급한 대로 이와 같은 현상으로 인해 상당한 수의 행이 상실될 수 있습니다. 이러한 유형의 오류에 대한 대책을 제공하는 것은 프로그래머의 책임입니다.

커서 닫기

커서를 닫는 프로세스는 반드시 IPC 기능이 있는 데이터베이스 엔진과 통신해야 합니다. 데이터베이스 엔진에는 프론트엔드가 커서를 닫을 정보가 반드시 필요합니다. 커서를 성공적으로 닫았으면,

데이터베이스 엔진은 커서가 닫혔거나 프로세스가 실패했다는 응답 또는 반응을 리턴합니다. 필터 기준이 변할 때 커서가 여러 번 다시 열릴 수도 있는 응용프로그램에서는 단순히 커서 닫기 작업을 제거함으로써 성능을 향상시킬 수 있습니다.

커서 열기란 커서를 닫아서 그 커서를 다시 여는 작업을 의미하므로 Informix-4GL에서 사용할 수 있습니다. 이 기법을 사용하면 불필요한 IPC 트래픽이 제거되며 응용프로그램의 스펙에 따라 성능을 크게 향상시킬 수 있습니다.

뷰

SQL에서 가장 많이 지나치는 기능 중 하나가 뷰입니다. 뷰는 SQL문과 연관된 이름입니다. 이 이름을 언급하는 것은 실제적으로 SQL문을 언급하는 것과 같습니다. 이러한 SQL문의 예는 다음과 같습니다.

```
create view cust_profile
                (cust_name,
                 cust_num,
                 cust_city,
                 cust_balance) as
               select cust_name,
                      customer_num,
                      city,
                      balance
                      from customer, cust_balance
                      where customer.customer_num =
cust_balance.cust_num
```

위의 SQL문은 customer 테이블을 cust_balance 테이블과 조인하는 SQL문과 관련된 뷰를 생성합니다. 일단 이 뷰를 작성한 경우, cust_profile을 언급하면 다음과 같은 select문이 실행됩니다.

```
select*
from cust_profile
where cust_num = 101
```

이 명령문은 뷰에 포함된 where절 기준에 표시된 선행하는 select문의 where절 기준을 추가하여, cust_profile 뷰에 액세스하고 기본적인 쿼리를 실행합니다. 선행하는 SQL문은 다음과 같은 SQL 문과 동일합니다.

```
select cust_name,
       customer_num,
       city,
       balance
from  customer, cust_balance
where customer.customer_num = cust_balance.cust_num and
      customer.customer_num = 101
```

SQL 뷰를 사용하여 분명히 다량의 복잡한 쿼리를 데이터베이스에 저장할 수 있으므로, 프로그래머의 데이터 액세스를 간소화할 수 있습니다. 데이터 액세스의 어려움과 복잡성을 줄이면 모든 4GL의 목표인 응용프로그램 개발에 필요한 코드를 줄일 수 있습니다.

또한 SQL 뷰를 사용하여 테이블에 대한 제한된 액세스를 제공할 수 있습니다. SQL 키워드 users와 제어 테이블을 사용하여 모든 테이블에 대해 제한된 뷰를 제공할 수 있습니다. 예는 다음과 같습니다.

```
- define a permissions table
create table user_perms ( logname char(20),
                          perm_level );

- table to access
create table menu_header ( menu_id integer,
                           menu_name char(20),
                           perm_level );
- create view to access menu_top table based on user login name

      create view menus as
        select  menu_name,
                menu_id
        from    menu_header, user_perms
        where   logname = user      and
                user_perms.perm_level = menu_header.perm_level
```

이 예에서는 먼저 각 사용자의 로그인 이름을 사용 권한 레벨과 연관시키는 테이블을 생성합니다. 그런 후, 이 테이블은 사용자 로그인 이름과 사용 권한 레벨(표시되지 않음)로 채워집니다. 다음으로 perm_level 컬럼을 사용하여 user_perms 테이블과 menu_header 테이블을 조인하는 뷰를 설정합니다. 이 관계는 각 perm_level에 대한 일대다 관계입니다. menu_header 테이블에는 여러 개의 menu_header 엔트리가 있습니다. (이 menu_header 테이블 엔트리는 이 사용자가 볼 수 있는 메뉴를 표시합니다.) 뷰 메뉴가 SQL의 내장 키워드인 user를 사용한다는 점에 유의하십시오. 이 키워드는 SQL문을 실행하는 사용자의 로그인 이름이 됩니다. 발행 중인 조인의 결과는 menu_header 테이블의 엔트리에 대한 제한된 뷰가 됩니다. 이 선택된 뷰를 추출하기 위해서는 다음 쿼리만 실행하면 됩니다.

```
select *
from menus
```

뷰는 기본적으로 user_perms 테이블과 menu_headers 테이블의 두 세트의 자연스러운 조인을 나타냅니다. 뷰는 이 관계의 요점입니다. 데이터 관리에 대한 이러한 세트 중심의 접근은 SQL의 핵심입니다. SQL은 Informix-4GL로 엄격하게 통합되기 때문에 이 언어를 신중하게 사용하면 Informix-4GL 사용의 의미 있는 결과를 얻는 데 도움이 됩니다.

모듈과 SQL

Informix-4GL 응용프로그램에서 모듈 사용을 유지보수하려는 전반적인 목표를 위해 SQL문을 중앙 집중화하고 하나 이상의 소스 코드 모듈로 함께 보관해야 합니다. 이 모듈은 데이터 검색 또는 데이터 업데이트와 같은 유사한 작업에만 사용해야 합니다. 또한, 읽기 가능성과 유지보수 가능성을 위해 커서 또는 명령문 ID의 선언은 커서를 사용할 지점에서 가까운 소스 코드 라인에 배치해야 합니다. 명령문 ID에 대해서도 동일한 원칙이 적용됩니다.

코드에서 사용될 장소와 가까이 커서 선언을 배치하는 또 다른 이유는, 소스 커서에서 사용되기 전에 커서를 선언해야 한다는 Informix-4GL의 요구사항 때문입니다. 다시 말해, 다음과 같이 커서는 소스 코드의 이전 라인에서 선언되어야 합니다.

```
- define a module scope variable as a flag
define first_pass smallint

function f()

let first_pass = TRUE

while ( continue = TRUE )

        call get_data(key1, key2) returning data_rec.*

...

function get_data(k1, k2)
...

if(first_pass) then
        declare data_curs cursor for . . .
              let first_pass = FALSE
end if
```

이 예에서, 모듈 범위를 갖는 변수 first_pass가 선언됩니다. 모듈 범위 변수는 함수 호출 간의 값을 유지하면서 변수가 선언되어 정적인 상태로 있는 전체 소스 코드 모듈 내에서 볼 수 있습니다. 이 변수는 선언된 소스 코드 모듈 밖에서는 볼 수 없습니다. 이 기법은 정적이지 않은 로컬 변수의 단점을 극복합니다. 이 유형의 변수는 단순 로컬(자동) 변수에는 없는 기능인 함수 get_data 호출 사이의 값을 유지해야 하기 때문에 first_pass 플래그에 필요합니다.

함수 get_data 내에서는 first_pass 플래그를 검사합니다. 플래그가 TRUE로 설정되어 있으면, if/then문 내의 코드 초기화가 실행됩니다. 여기에는 커서 선언이 포함됩니다. 플래그가 FALSE로 설정되어 있으면, 코드 초기화가 실행되지 않습니다. 이로써 과다한 커서 선언의 오버헤드를 방지할 수 있고, 커서와 관련된 모든 코드를 동일 위치에 함께 지정함으로써 모듈 기능을 향상시킬 수 있습니다.

요약

이 장에서는 Informix-4GL 언어를 소개했습니다. 이 장에서 자세히 설명한 대로, 이 언어를 사용하면 IDS 데이터베이스와의 자연스러운 상호 작용이 가능합니다. Informix-4GL 자료형은 IDS 데이터베이스의 자료형을 미러링하며, 이 언어는 Informix-4GL 프로그램에서 실제로 제한없이 사용할 수 있는 SQL문 전체 세트를 제공합니다.

IDS 데이터베이스에서 검색한 데이터는 원시 프로그램 변수에 배치될 수 있고 프로그램 변수의 데이터는 IDS 데이터베이스에 삽입될 수 있습니다. 4GL 언어는 Informix-4GL의 한 번에 한 레코드 논리와 비프로시저 SQL의 한 번에 한 세트 논리를 일치시킬 수 있는 커서를 제공합니다. 커서에는 이와 연관된 다양한 기능이 있어, `select`문에서 정의된 세트에 무작위로 액세스할 수 있거나, 업데이트 커서 또는 삽입 커서를 사용하여 효율적으로 업데이트 또는 삽입할 수 있습니다.

다음 장 미리보기

다음 장에서는 Informix-4GL 프로그램을 사용한 입력 및 출력 수행 프로세스에 대해 다룹니다. Informix-4GL의 보고서 쓰기 구문을 사용하여 보고서를 생성하는 프로세스와 함께 화면 폼 즉, 데이터 터미널과의 상호 작용을 위한 방법에 대해서도 다룹니다.

9장

화면 폼 및 보고서

Informix-4GL 화면 폼은 화면 상호 작용에 대한 템플리트입니다. 이 화면 폼은 화면에 출력할 데이터, 데이터의 특성 및 데이터를 표시할 때 사용할 속성을 나타냅니다. 필요에 따라 상당한 양의 데이터 바인딩이 발생할 수 있습니다. 화면 필드를 테이블 컬럼에 바인드할 수 있습니다. 즉, 컬럼의 자료형과 길이는 폼이 작성될 때 해당 컬럼의 자료형 및 길이와 일치합니다.

Informix-4GL은 form4gl이라고 하는 폼 컴파일러를 사용합니다. 이 컴파일러는 화면 폼 이미지를 읽고 화면 폼의 컴파일된 이미지를 출력합니다. 이 파일의 데이터는 런타임 시 Informix-4GL 프로그램이 사용하는 내부 구조로 매핑됩니다. Informix-4GL 화면 폼 인터페이스는 동적 인터페이스로 설계되어 있습니다.

터미널의 특성은 런타임 시 termcap 또는 terminfo 정의 파일에서 읽습니다. 이는 런타임 때까지 특정 터미널 특성이 알려지지 않음을 의미합니다. 따라서 컴파일러는 특정 화면 폼 대화형문의 유효성을 결정할 수 없습니다. 이로 인해 화면 폼 개발 프로세스가 어려워질 수 있습니다. 이 장에서는 이 프로세스를 보다 쉽고 예측 가능하게 만들기 위한 몇 가지 요령에 대해 설명하겠습니다.

화면 크기 관리

화면 폼 크기의 기본값은 라인 당 80자로 24라인이지만 화면 폼에서 screen size문을 사용하여 변경할 수 있습니다. 터미널들은 화면 크기가 다를 수 있으므로 이러한 매개변수의 유효성은 런타임 때까지 검사되지 않습니다. 터미널 기능이 응용프로그램을 실행하는

데 필요한 정도만 되면 응용프로그램이 실행됩니다.

화면 폼 프로그래밍

Informix-4GL 화면 폼은 database 섹션, screen 섹션, tables 섹션, attributes 섹션 및 선택적 instructions 섹션으로 구성되어 있습니다. screen 섹션에는 화면의 이미지가 있습니다. 이미지는 일반 텍스트와 화면 필드로 구성되어 있습니다. 화면 필드는 입력 또는 표시가 발생할 위치를 표시하는 레이블입니다. 이들 필드는 여는 중괄호, 첫 번째 알파벳 문자 및/또는 추가 알파벳 문자와 숫자로 구성된 화면 레이블로 식별됩니다. 레이블 뒤에서 필드는 정의된 필드 길이까지 공백으로 채워집니다. 필드의 끝은 닫는 중괄호로 식별됩니다. 화면 필드 레이블에 사용된 알파벳 문자는 대문자이거나 소문자이며 화면 폼 컴파일러는 대소문자를 구분하지 않습니다. 단일 문자 필드에 대한 필드 레이블을 작성하기 위해 단일 문자를 사용할 수 있으며 화면 폼에 넣을 수 있는 최대 단일 문자 필드 수는 26개입니다.

화면 폼에서 데이터베이스 바인딩

Informix-4GL 화면 폼은 데이터베이스 테이블과 상호 작용하도록 설계되었기 때문에 화면 폼 인터페이스 내에서 데이터베이스 바인딩 폼을 사용할 수 있습니다. 화면 폼의 database 섹션에서 데이터베이스를 식별한 후, attributes 섹션에서 데이터베이스 테이블 필드를 화면 폼의 필드로 바인드할 수 있습니다. 이들 필드의 자료형은 데이터베이스 테이블에 있는 필드의 자료형으로 추정됩니다. 그러나 필드의 길이는 화면 폼에 있는 필드의 길이입니다. 런타임 시 이들 필드에 데이터가 입력되면 데이터베이스 테이블 필드의 자료형과 비교하여 이들 데이터의 자료형을 검사합니다. 입력된 데이터를 테이블 필드의 자료형으로 변환할 수 없는 경우 오류를 신호하고 사용자 입력은 같은 필드에 그대로 남아 있습니다. 데이터 길이가 다른 경우 절단할 때 중요한 숫자가 유실되지 않으면 절단이 수행됩니다. 이 경우에는 오류를 신호하지 않습니다.

attributes 섹션은 화면 폼 필드에 대한 추가 속성을 지정하는 수단을 제공합니다. 이들 속성의 대부분은 화면 폼에서와 마찬가지로 Informix-4GL 응용프로그램 내에서 지정할 수 있습니다. 양쪽에 모두 지정되는 경우 약간의 경쟁이 발생할 수 있습니다.

화면 폼의 instructions 섹션은 선택적입니다. Informix-4GL에 사용된 필드 구분 문자를 재정의하거나 화면 레코드를 작성하는 데 사용할 수 있습니다. 화면 레코드는 화면 폼 필드들을 단일 레코드에 함께 그룹화하는 유용한 수단으로, Informix-4GL 화면 대화형 문에서 참조될 수 있습니다.

formonly 화면

화면 폼이 특정 데이터베이스와 상호 작용하지 않을 경우 데이터베이스 명령문은 데이터베이스로 formonly를 지정할 수 있습니다. 이 경우, tables 섹션은 사용할 수 없습니다.

formonly 화면 폼은 일반 화면 즉, 특정 데이터베이스에 바인드되지 않은 화면을 설계하기 위한 훌륭한 메커니즘을 제공합니다. 이들 화면은 다양한 테이블의 데이터 처리에 사용할 수 있는 라이브러리 루틴을 개발하는 데 사용할 수 있습니다. 이러한 루틴은 모듈식 응용프로그램의 빌딩 블록(building block)입니다.

formonly 화면은 database 섹션에서 데이터베이스를 사용하지 않는 폼으로 표시되도록 지정됩니다. 이에 대한 구문은 다음과 같습니다.

```
database formonly
screen {

...
```

물론, 폼을 formonly로 지정하면 폼 내에서 데이터베이스 바인딩을 할 수 없게 됩니다. 데이터베이스 바인딩을 사용하면 화면이 formonly 필드로 제한되지 않습니다. 다음의 폼이 보여주듯이 테이블 컬럼에 바인드된 필드를 formonly 필드와 혼합할 수 있습니다.

```
database inventory
screen
{

Product Name              [f001             ]
Wholesale Price           [f002    ]
Markup                    [f003       ]
```

```
Total                        [f004       ]

}

tables
inventory
stock

attributes
f001 = stock.product_name;
f002 = stock.curr_price;
f003 = inventory.markup_amt;
f004 = formonly.total;
...
```

이 예에서는 폼에 있는 대부분의 필드가 inventory 데이터베이스의 테이블에서 온 것이므로 이 데이터베이스가 선택됩니다. 그러나 필드 중에서 total 필드는 계산 결과 필드입니다. 이 필드의 데이터는 데이터베이스 테이블에서 온 것이 아닙니다. 따라서 이 필드는 formonly 필드로 지정됩니다.

formonly 필드에 대해서는 자료형이 선언되지 않았으므로 formonly 필드는 공백을 포함하여 화면 폼의 화면 폼 필드와 같은 길이인 문자 필드로 처리됩니다. 이 필드에는 숫자가 표시되지만 대부분의 Informix-4GL문에서와 같이 데이터 변환이 자유롭게 이루어지므로 아무 문제가 없습니다. 다음의 구문을 사용하여 이 필드에 대해 선택적으로 자료형을 선언할 수 있습니다.

```
....

attributes
f004 = formonly.total, type decimal(5,2);
```

Informix-4GL로 화면 폼 처리

Informix-4GL은 화면 폼을 통해 화면의 데이터를 처리합니다. 화면 폼을 윈도우에 표시할 수 있으며 여러 윈도우와 화면을 동시에 열고 표시할 수 있습니다. 열린 각 화면 폼은 열린 파일을 나타내기 때문에, 열린 화면 폼 수를 실제로 제한하는 것은 오직 응용프로그램

플랫폼에서 사용할 수 있는 파일 설명자 수입니다.

open form문은 화면 폼을 여는 데 사용됩니다. 이 명령문을 사용하여 화면 폼 식별자를 화면 폼과 연관시킬 수 있습니다. 또한 이 명령문은 런타임 응용프로그램이 화면 폼을 여는 데 필요한 컴파일된 화면 폼 파일을 찾을 수 있는 위치를 식별합니다. 이 정보의 유효성은 런타임 때까지는 검사되지 않습니다.

화면 폼 식별자

화면 폼은 응용프로그램에서 전역 범위를 갖는 식별자에 의해 식별됩니다. 이 식별자는 정적이며 컴파일 시간에 작성됩니다. 런타임 시에는 작성될 수 없습니다. 이것은 하나의 Informix-4GL 소스 코드 모듈에서 열려진 폼이 다른 소스 코드 모듈에서 인식된다는 것을 의미합니다.

모든 전역 식별자와 마찬가지로 이 식별자는 문제를 일으킬 수 있습니다. 동일한 함수 내에서 열어 사용하고 닫은 화면 폼은 관련 없는 함수에서 열어 사용한 화면 폼과 이름이 충돌할 수 있습니다. 이 문제에 대한 한 가지 해결책은 다음 예에서와 같이 일반 화면 폼에 대해 밑줄 문자를 앞에 사용하는 이름을 지정하는 것입니다.

```
open form _custf1 from "cust_form"
display form _custf1
```

모듈성과 화면 폼 상호 작용

응용프로그램을 개발할 때 화면 폼과 관련하여 몇 가지 결정을 내려야 합니다. 한 가지 프로그래밍 스타일은 동일한 함수에서 모든 화면 폼을 열도록 선택하는 것입니다. 이들 폼은 응용프로그램을 초기화하는 동안 열리고 사용 가능해집니다. 따라서 응용프로그램이 화면 폼을 사용해야 한다고 판단할 때 화면 폼을 열 필요가 없습니다. 폼이 이미 열려 있으므로 화면 폼만 표시하면 됩니다.

화면 폼을 열 때는 약간의 오버헤드가 발생합니다. 화면 폼 파일을 열어야 하며, 이 때 운영 체제 오버헤드가 요구됩니다. 파일의 내용을 Informix-4GL의 내부 화면 폼 구조로 읽어야 하며 이러한 구조에 대해 데이터 영역을 할당해야 합니다. 이는 오래되고 효율성이 떨어지는 플랫폼에서는 문제가 될 수 있지만 대부분의 최신 하드웨어는 실제로 거의 느낄 수 없는 사이에 이러한 기능을 수행합니다.

모듈성을 향상시키고 유지보수 가능성을 개선하는 코딩 스타일은 모든 Informix-4GL 화면 폼 작업을 하나 이상의 화면 폼 모듈에 함께 넣어 두는 것입니다. 이 모듈에서는 화면 폼이 열리고 표시되며 필요하면 닫힙니다.

화면 폼 열기용으로만 지정된 단일 함수에서 여러 화면 폼을 동시에 여는 대신 화면 폼이 표시되어, 사용될 함수에서 화면 폼을 열 수 있습니다. 이러한 시도의 장점은 유지보수 가능성입니다. 프로그래머가 화면을 표시하는 데 사용되는 화면 폼을 결정하려면 현재 함수만 보면 됩니다.

다중 윈도우 관리

Informix-4GL 화면 폼의 올바른 윈도우 크기를 결정하는 것은 일종의 예술이 되었습니다. 화면 폼을 사용하여 윈도우를 열 수 있지만 기본 윈도우 크기가 항상 응용프로그램에 적당한 크기는 아닙니다. 다음의 공식을 사용하여 Informix-4GL 폼에 대한 윈도우의 크기를 지정할 수 있습니다.

필요한 라인 수 = (폼 라인 – 1) + (폼 길이 + 주석 라인을 위한 1)
+ 2(윈도우에 경계선이 있는 경우)
필요한 컬럼 수 = 가장 오른쪽 컬럼의 위치 + 2
(윈도우에 경계선이 있는 경우)

기본적으로 폼 라인은 화면의 라인 1에서 시작합니다. 이것은 options문을 사용하여 변경할 수 있습니다. 표시된 공식이 나타내듯이 폼 라인을 변경하면 화면의 크기 지정에 영향을 줄 수 있습니다.

현재 윈도우

Informix-4GL을 사용하여 여러 화면을 열고 표시할 수 있습니다. 한 윈도우에서 다른 윈도우로 제어를 전달할 수 있으며 각 윈도우는 개별 화면 폼을 표시하고 개별 입력 작업을 수행합니다.

current window문은 윈도우 간에 제어를 이동시키는 데 사용됩니다. 명령문이 current window is screen으로 실행되면 하단 층 윈도우의 빈 화면이 맨 위에 놓여지고 다른 모든 윈도우는 그 아래에 놓여집니다. 이 명령문이 실행되면 표시되는 모든 윈도우나 폼은 이 빈 화면에 표시됩니다. 다음 프로그램은 이 기능을 보여줍니다.

```
main

open window w1 at 5, 5
    with 10 row, 10 columns
    attribute(border)
display "1" at 1, 1

current window is screen
open window w2 at 10, 10
    with 10 rows, 10 columns
    attributes(border)
display "2" at 1, 1

current window is screen
open window w3 at 15, 15
    with 7 rows, 10 columns
    attribute(border)
display "3" at 1, 1

sleep 10

end main
```

이 프로그램은 경계선이 있는 세 개의 윈도우를 화면에 표시하는데 각 윈도우에는 별도의 번호가 표시됩니다. 그러나 각 윈도우가 표시될 때 current window is screen문이 실행되고 이전 윈도우를 빈 화면으로 가립니다. 그래서, 마지막 윈도우가 표시되면 그림 9-1과 같이 유일하게 볼 수 있는 윈도우가 됩니다.

층으로 된 윈도우 세트를 표시하려면 다음 코드와 같이 current window is screen문을 제거해야 합니다. 그러면 세 개의 층으로 된 윈도우 세트가 표시됩니다.

```
main

open window w1 at 5, 5
    with 10 rows, 10 columns
    attribute(border)
display "1" at 1, 1
```

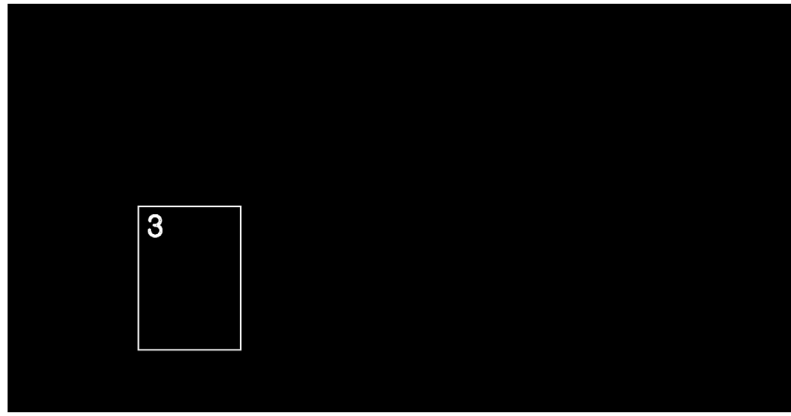


그림 9-1 Informix-4GL 윈도우 관리

```
open window w2 at 10, 10
    with 10 rows, 10 columns
    attribute(border)
display "2" at 1, 1

open window w3 at 15, 15
    with 7 rows, 10 columns
    attribute(border)
display "3" at 1, 1

sleep 10

end main
```

이 코드에서는 각 윈도우가 순차적으로 표시됩니다. 그러면, 첫 번째 표시되는 윈도우는 맨 아래 층이고, 다음에 표시되는 윈도우는 두 번째 층이며, 세 번째와 마지막으로 표시되는 윈도우는 맨 위 층으로 나타냅니다. 이 결과는 그림 9-2와 같습니다.

현재 윈도우를 쉽게 결정할 수 있는 방법은 없습니다. 그러나 이로 인해 윈도우 처리 명령문을 윈도우가 사용되는 코드에 가깝게, 보통은 동일한 함수 블록 내에 유지할 수 있는 장점이 있습니다. 이 점을 따른다면, 현재 함수의 시작을 보기만 해도 어느 윈도우가 현재 윈도우인지 쉽게 판별할 수 있습니다. 다음의 코드 샘플은 이러한 개념을 보여줍니다.

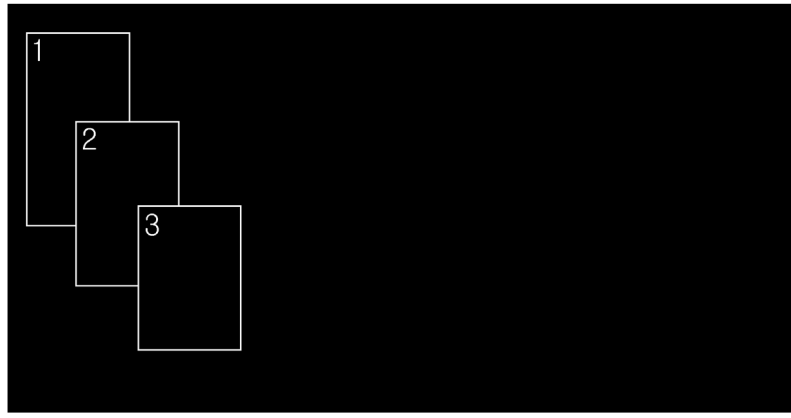


그림 9-2 Informix-4GL 윈도우 관리

```
define wflags array[3] of smallint

main

initialize wflags to NULL

call f1()
call f2()
call f3()

end main

function f1()

if wflags[1]   IS NULL then
    open window w1
    open form f1
    let wflags[1] = TRUE
end if
current window w1

...

end function
```

```
function f2()

if wflags[2]  IS NULL then
    open window w2
    open form f2
    let wflags[2] = TRUE
end if
current window w2

...

end function

function f3()

if wflags[3]  IS NULL then
    open window w3
    open form f3
    let wflags[3] = TRUE
end if
current window w3
end function

function close_windows()
define n smallint

for n = 1 to 3

    if wflags[n] is NOT NULL  then
        case n
    when 1 close window w1
        close form f1
    when 2 close window w2
        close form f2
    when 3 close window w3
        close form f3
   end case
  end if
```

```
end for

end function
```

이 샘플 응용프로그램은 윈도우 및 폼 관리에 대한 모듈식 접근법을 보여줍니다. 먼저 세 개의 smallint 변수 배열(wflags)이 모듈 범위로 선언됩니다. 이들 변수는 화면 폼을 사용하는 함수에서 플래그로 사용됩니다. 기본 프로그램 블록에서 플래그 배열은 NULL로 초기화됩니다.

그런 다음, 함수를 순서대로 호출합니다. 각 함수의 본문에서 플래그 변수를 테스트합니다. 플래그 변수가 NULL이면, 해당 함수의 윈도우와 화면 폼이 열리고 화면 폼이 표시됩니다. 플래그가 설정되지 않으면, 현재 윈도우가 적절한 윈도우로 설정되고 함수는 계속 실행됩니다. close_windows 함수는 열린 윈도우만 닫는 데 사용됩니다. 문제의 윈도우에 대해 wflags 배열 요소를 평가하여 이 정보를 판별할 수 있습니다.

화면 윈도우 작업 모듈성에 대한 이러한 접근법은 대부분의 윈도우 작업을 윈도우를 사용하는 함수에 두기 때문에 이러한 작업에 대한 제어를 모두 가깝게 보존할 수 있습니다. 이 접근법의 장점은 모듈성과 효율성이 개선된 것입니다. 윈도우와 폼은 사용되는 경우에만 열립니다.

display 명령문

display문은 현재 화면이나 현재 화면 폼의 특정 필드에 출력을 표시하는 데 사용됩니다. 현재 화면 폼을 결정하기 위해 지원되는 방법이 없기 때문에 프로그래머는 현재 열린 화면 폼에 대해 알고 있어야 합니다.

display문의 가장 간단한 폼을 사용하여 터미널 화면의 현재 위치로 간단한 데이터 출력을 수행할 수 있습니다. 이러한 형식의 명령문은 사용될 때 현재 터미널 화면을 정리하거나 초기화하려고 하지 않습니다. 이 명령문은 C 언어 puts 함수 호출과 같은 역할을 수행합니다. 즉, 간단하게 데이터 변수를 전달하고 문자열 자료형으로 변환하고 현재 터미널 커서 위치로 데이터를 출력합니다. 이는 다음 코드의 예와 같이 가장 간단한 폼의 display문을 사용하여 간단한 보고서를 생성할 수 있음을 의미합니다.

```
main
define salesm record like salesman.*

declare sales_curs cursor for
                    select *
                    from salesman
                    where region = "NE"
foreach sales_curs into salesm.*
        display "Salesman:   First Name: ",
                 salesm.first_name,
                " Last ", salesm.last_name,
                " Phone: ", salesm.phone

        end foreach

end main
```

이 예에서는 북동부 지역에 있는 모든 판매원의 보고서가 터미널 화면에 인쇄됩니다. 터미널 화면을 재설정하는 화면 대화형문이 실행되지 않았기 때문에 display문은 간단하게 출력을 가져와 터미널 화면으로 보내며 출력 끝에 개행 문자를 표시합니다. 다음과 같이 Unix lp 명령을 사용하여 이 프로그램의 출력을 시스템 프린터로 보낼 수 있습니다.

```
rpt_prog | lp
```

간단한 폼의 display문은 Informix-4GL의 단순 데이터 변수를 가져와 화면에 출력할 수 있습니다. 그러나 현재 폼에서는 표현식이 순 자료형으로 평가되더라도 표현식을 인수로 사용할 수 없습니다. 다음의 예가 이러한 문제를 보여줍니다.

```
function funct_s()

        return  "hello world"

end function

display funct_s()
```

이 코드 예는 표현식 funct_s()가 단순 문자열로 해석되더라도 funct_s가 display문에 대해 너무 복잡한 자료형으로 간주되기 때문에 컴파일에 실패합니다. 다음 명령문도 컴파일되지 않습니다.

```
define x,y smallint
let x = 1
let y = 2
display ( x + y / 2 )
```

컴파일러는 간단한 수학적 표현식도 display문에 대해서는 아주 복잡한 것으로 간주합니다. 이러한 제한을 해결하는 방법은 단지 다음과 같이 임시 변수를 사용하여 이들 표현식의 결과를 저장하는 것입니다.

```
let str = s()
let x = x + y / 2

display x,str
```

이 display문은 컴파일될 뿐 아니라 오류 없이 실행됩니다.

display 명령문과 화면 폼 상호 작용

display문은 by name절을 사용하여 데이터베이스 바인딩을 명시적으로 사용할 수 있습니다. 이 절을 사용하면 Informix-4GL 런타임 화면 폼 구성 요소는 사용 중인 레코드의 레코드 구성 요소 이름과 일치하는 컬럼 이름을 검색합니다. 이러한 레코드 구성 요소 이름은 화면 폼 필드 이름과 정확히 일치해야 합니다. 그렇지 않으면 명령문이 실패합니다. 이 명령문의 예는 다음과 같습니다.

```
 database sales
screen
{
                                Customer Record

            First Name     [f001                  ]
            Last Name      [f002                  ]
```

```
            Address       [f003                  ]
        }
tables
customer

attributes
f001 = customer.first_name;
f002 = customer.last_name;
f003 = customer.address;

....

- Informix-4GL program

      define cust_rec record like customer.*

...
      display by name cust_rec.* to customer.*
```

이 예에서는 화면 폼의 필드가 고객 테이블의 컬럼과 동일한 순서 및 동일한 이름으로 지정되기 때문에 display문에서 참조될 수 있습니다.

by name절을 사용하기 위해 전체 테이블을 참조할 필요는 없습니다. 하지만 한 테이블의 일부분이나 여러 테이블의 부분들은 참조할 수 있습니다. 컬럼의 이름이 레코드 요소 이름과 동일하며 적절한 자료형 변환을 수행할 수만 있으면 by name절을 사용할 수 있습니다. 다음의 예는 화면 레코드를 사용하는 기능을 보여줍니다.

```
database sales
screen
{
                                    Customer Record

            First Name           [f001                 ]
            Last Name            [f002                 ]
            Salesman Number      [f003                 ]

     }
tables
```

```
customer
salesman

attributes
f001 = customer.first_name;
f002 = customer.last_name;
f004 = salesman.number;
instructions
screen record scr1 ( customer.first_name,
       customer.last_name,
       salesman.number)

...

- Informix-4GL program to display data to the screen form

       define scr_rec record like customer.first_name,
       customer.last_name,
       salesman.first_name,
       salesman.last_name
...
       display by name scr_rec.* to scr1.*
```

이 예에서 salesman 테이블 및 customer 테이블의 두 테이블과 상호 작용하는 화면 폼이 작성됩니다. 이들 두 테이블에서 사용되는 컬럼은 화면 폼의 attributes 섹션에서 식별되고 참조됩니다. 그런 다음 이들 컬럼은 화면 레코드에 놓여집니다. Informix-4GL 프로그램에서는 이 화면 레코드가 display문의 대상으로 사용됩니다. scr_rec 레코드의 전체 내용은 scr1 화면 레코드에 표시되며 이 때 레코드 요소의 레코드 이름은 컬럼 이름과 일치합니다.

input 명령문

input문은 Informix-4GL 화면 폼을 사용하여 터미널에서 데이터를 검색합니다. 대부분의 데이터 입력 제어는 Informix-4GL 프로그램에 달려 있지만 화면 폼 정의도 데이터 입력에 영향을 줍니다. Informix-4GL 프로그램은 화면 폼과 상호 작용하여 터미널에서 데이터를 검색합니다. 이 상호 작용은 input문을 사용하여 수행

됩니다.

input문은 대상 폼의 각 필드에서 데이터를 검색합니다. 내부 입력 목록의 맨 위부터 시작하여 입력 목록의 마지막 필드로 종료하는 일련의 형식으로 필드 사이를 이동합니다. 입력 순서는 화면 레코드를 사용하지 않는 경우 화면 폼의 attributes 섹션에 나열된 필드의 순서에 따라 제어되고, 화면 레코드를 사용하는 경우 화면 레코드에 나열된 필드의 순서에 따라 제어됩니다.

```
define cust_rec record like customer.*

- the order in which the fields appear in the screen
- form will control the order of input

input cust_rec.* from customer.*
```

input문을 실행하는 동안 여러 이벤트가 트랩될 수 있습니다. 이들 이벤트 트랩은 Informix-4GL 언어의 보다 독특하고 흥미 있는 면 중 하나입니다. 이벤트는 before field, after field 및 on key에 해당합니다. before field문은 이름이 지정된 필드를 입력하기 전에 입력 작업을 제어하는 데 사용됩니다. after field문은 커서가 필드에서 떠난 다음에 제어할 수 있게 합니다. on key문은 사용자가 정의된 기능 키를 누를 때 응용프로그램을 제어하는 데 사용됩니다. 이들 명령문의 예가 다음 절에 나와 있습니다.

입력 필드 통과

커서를 화면 폼의 필드에 들어가지 못하도록 하는 것이 적절한 경우가 있습니다. 이러한 필드는 조회 정보나 화면 폼의 다른 필드에서 파생된 필드를 표시하는 데 사용될 수 있습니다. 이 필드를 스킵하는 방법에는 두 가지가 있습니다. 다음의 코드 샘플은 before field 블록에 next field문을 사용하여 화면 필드를 스킵합니다.

```
database sales
screen
{

                Salesman Information
```

```
First Name       [f001                ]
Last Name        [f002                ]
Territory Code   [f003                ]
Name             [f004                ]
}

tables
salesman
territory

attributes
f001 = salesman.first_name;
f002 = salesman.last_name;
f003 = salesman.territory;
f004 = territory.territory_name,  reverse;

instructions

screen record sc1 (salesman.first_name, salesman.last_name,
salesman.territory, territory.territory_name)

...

define scr_rec record like salesman.first_name,
salesman.lastname, salesman.territory, territory.territory_na
me;

input scr_rec.* from  sc1.*

before field territory_name
    display scr_rec.territory_name to
    territory_name_
    next field salesman.first_name
...
```

이 예에서, 커서가 territory_name 필드로 들어가기 전에 이 필드의 before field문이 salesman.first_name 필드로 입력의 방향을 지정하기 때문에 input문은 territory_name 필드를 스킵합니다. 다음과 같이 화면 폼의 화면 필드 속성이 noentry 필드가 되

도록 선언하면 이와 동일한 기능을 제공할 수 있습니다.

```
...
f004 = territory.territory_name, reverse, noentry;
```

모두 올바른 방법입니다. 속성 항목을 화면 폼에 놓으면 Informix-4GL 응용프로그램에 필요한 코드가 줄어드는 장점이 있습니다. 그러나 제공된 응용프로그램의 입력 루프를 검토하는 유지보수 프로그래머는 화면 폼 필드에서 사용되고 있는 속성을 즉시 알지 못합니다. 속성 정보를 input문 블록 내에 놓으면 프로그래머는 구조화된 구문을 사용하여 동일한 기능을 수행합니다.

현재 필드 강조 표시

종종 그림 9-3과 같이 현재 입력 필드를 강조 표시하는 것이 바람직할 때가 있습니다. 이렇게 하면 사용자가 현재 입력되고 있는 필드를 보다 쉽게 알아볼 수 있습니다. 불행하게도 Informix-4GL에는 현재 이 동작을 트리거하는 절이 없습니다. 그러나 다음의 예에서와 같이 이 기능을 수행하는 것은 어려운 일이 아닙니다.

```
Input  Insert  Update  Query  Next  Previous  Table ...
Query a row
               acct_num       100001
               acct_group     Expense
               acct_type      5
Table          acct_num       Programming Expense
accounts
```

그림 9-3 현재 필드 강조 표시

```
- highlight the current field in the input loop

input scr_rec.* from scr1.*

before field first_name
    display scr_rec.first_name to first_name
```

```
        attribute ( reverse )

after field first_name
    display scr_rec.first_name to first_name
        attribute ( normal )

before field last_name
    display scr_rec.last_name to last_name
        attribute( reverse )

after field last_name
    display scr_rec.last_name to last_name
        attribute( normal )
```

이 예에서는 attribute (reverse)절을 사용하여 현재 필드의 데이터를 화면에 표시함으로써 before field문에서 현재 필드를 강조 표시합니다. 그러면 사용자가 데이터를 입력하고 있으며 커서가 이 필드에 남아 있는 동안 전체 필드(화면 폼에서 정의된 길이)를 역상 속성 상태로 둡니다. 커서가 필드를 떠날 때 역상을 제거하려면 after field문의 코드가 필요합니다. 이 코드는 입력 중에 변경되었을 수 있는 데이터 레코드의 현재 내용을, 정상 비디오 속성을 사용하여 화면 필드에 다시 표시합니다. 정상 속성이 display문의 기본 속성이기 때문에 이 display문의 attribute문은 불필요하지만 코드를 보다 읽기 쉽게 하기 위해 여기에 포함되었습니다. 이 절을 사용함으로써 정상 속성이 강조되고 있으므로 유지보수 프로그래머는 display문의 용도를 이해할 수 있을 것입니다.

termcap 문제

많은 Informix-4GL 기능은 사용 중인 터미널의 정의에 따라 결정됩니다. Unix 시스템 사용 시, 이러한 정의는 /etc 디렉토리의 termcap이라는 파일에 들어 있습니다. 터미널 속성을 사용할 수 없는 경우(termcap 파일에 정의되지 않음), Informix-4GL이 이 속성을 호출하려고 시도하면 이 명령문은 일반적으로 화면 폼 런타임 구성요소에 의해 무시됩니다.

예를 들어, 터미널이 역상을 지원하지 않는 경우 간단하게 역상이 표시되지 않습니다. 또한 터미널의 termcap에 기능 키가 지정되지 않은 경우 Informix-4GL은 이러한 기능 키에 대한 액세스 시도를

실패합니다.

기능 키 재정의

Informix-4GL은 options문을 통해 특정 키를 재매핑할 수 있도록 합니다. 프로그래머가 입력 중 특정 키를 해제하려는 경우, options문을 사용하여 이들 키를 사용 중인 터미널에 대해 정의되지 않은 기능 키로 재정의할 수 있습니다.

input array에서 insert key 및 delete key 해제

응용프로그램이 사용자가 행을 삽입하거나 삭제할 수 없도록 하는 input array문을 사용하는 경우, 다음과 같이 options문을 사용하여 insert key 또는 delete key 키를 사용 중인 터미널에 대해 정의되지 않은 기능 키로 매핑할 수 있습니다.

```
options
insert key f39,
delete key f40
```

이 코드는 insert key를 기능 키 39로 재정의하고 delete key를 기능 키 40으로 재정의하는데, 이들 키는 대부분의 터미널에서 생성할 수 없습니다. 터미널이 이러한 기능 키를 허용하지 않을 경우 insert key와 delete key는 input array 중에 해제됩니다.

termcap 파일에서 기능 키 추가

Informix-4GL termcap에 기능 키를 추가하는 것은 비교적 간단합니다. 하지만 프로그래머는 먼저 터미널에 의해 전송된 제어 시퀀스를 결정해야 합니다. 이는 보통 기능 키를 누르고 출력을 파일로 방향 재지정함으로써 수행할 수 있습니다. 그런 다음, 제어 키를 표시하는 텍스트 편집기나 파일 내용의 ASCII 또는 16진 값을 표시하는 od와 같은 Unix 유틸리티를 사용하여 이 파일을 읽을 수 있습니다.

기능 키 제어 시퀀스가 결정되었으면 termcap 파일에 적절하게 입력만 하면 됩니다. 예를 들어, 사용 중인 터미널이 기능 키 1(f1)에 대해 제어 시퀀스 \E(Esc) OP를 보내고 있는 것으로 나타난 경우, 이 터미널의 termcap에 대한 올바른 입력은 다음과 같습니다.

```
v1|vt100|vt-100|DEC vt100:\
...
        :k0=\EOP:k1=\EOQ:k2=\EOR:k3=\EOS:\
...
```

이전 페이지 또는 다음 페이지 키가 필요한 경우 이 프로세스는 기능 키의 프로세스와 비슷합니다. 프로그래머는 먼저 전송된 제어 시퀀스를 판별해야 합니다. 제어 시퀀스가 판별되면, 이전 페이지 및 다음 페이지 키를 트랩하는 데 사용할 기능 키 번호에 이 Esc 키 시퀀스를 관련시키기 위해 termcap 입력이 필요합니다. 이 프로세스는 다음과 같습니다.

```
v1|vt100|vt-100|DEC vt100:\
...
        :k31=\EOP:k32=\EOQ:k2=\EOR:k3=\EOS:\
...
```

그런 다음, 다음과 같이 응용프로그램에 대한 options문에서 next key 키 및 previous key에 대한 기능 키를 page-up 및 page-down 키에 대해 매핑된 기능 키로 설정해야 합니다.

```
options
next key f31,
previous key f32
```

위의 단계들을 완료한 다음에는 input array 또는 display array 명령문이 실행될 때마다 터미널에 대한 page-up 및 page-down 키를 사용하여 화면 배열을 적절하게 앞이나 뒤로 스크롤할 수 있습니다.

interrupt key

Informix-4GL은 응용프로그램이 사용자 응답을 기다리고 있을 때 interrupt key를 트랩하는 기능을 제공합니다. 이 인터럽트 트랩은 defer interrupt문을 사용하여 수행됩니다. 기본적으로 프로그램이 인터럽트 시그널을 받으면 응용프로그램은 중단됩니다. 이 동작이 항상 바람직한 것은 아니므로 응용프로그램에서 interrupt key를 트랩하는 것이 유용합니다. interrupt key가 트랩된 경우 input문의 끝에서 인터럽트 플래그(int_flag)를 검사하여 사용자가 interrupt key를 눌렀는지 여부를 판별해야 합니다. 다음 코드는 이러한 방법을 보여줍니다.

```
database accounting

main
define ar record like accounts.*

defer interrupt

options message line last - 1

open form accts from "accts"
display form accts

let int_flag = FALSE
input by name ar.*

    before field acct_num
        message "Enter Account Number"
        display ar.acct_num to acct_num attribute( normal )
end input
if int_flag then
    error "Input Aborted."
    initialize ar.* to NULL
    return ar.*
else
    return ar.*
end if
```

이 예에서는 화면 폼이 열리고 표시되기 전에 defer interrupt 옵션이 설정됩니다. 입력 루프가 시작되기 전에 int_flag가 FALSE로 설정됩니다. 마지막 인터럽트 이후 응용프로그램이 int_flag를 재설정하지 않은 경우의 일을 허용하기 위해 이러한 조치가 필요합니다.

입력 블록 다음에 int_flag가 테스트됩니다. int_flag가 설정되었으면 오류 메시지가 표시된 후 int_flag가 재설정되고, ar 레코드는 NULL로 설정되고 함수는 NULL 레코드를 리턴합니다. int_flag가 설정되지 않았으면 함수는 간단하게 이 레코드만을 리턴합니다.

Informix-4GL 보고서

Informix-4GL 보고서 작성기를 작성하는 데 있어서 개발자들은 Informix-4GL 보고서 작성기의 선구자 격인 Ace의 한계를 극복하는 생산적인 보고서 생성 환경을 제공하고자 노력했습니다. Ace의 구문은 Informix-4GL 보고서 작성기에 사실상 거의 그대로 적용되었으며 모든 4GL 프로시저 및 비프로시저 명령문과 결합되었습니다. 그 결과 Informix-4GL 보고서 기능이 만들어졌습니다.

Informix-4GL 보고서 기능은 보고서 생성을 위한 블록 지향의 구조화된 명령문 세트를 제공하는 프로그램 명령문 세트입니다. 포맷을 정의하고 제어 변수를 초기화하며 보고서 출력을 생성할 수 있도록 제어 블록이 제공됩니다. 보고서 블록을 사용하면 모든 프로시저 명령문과 대부분의 비프로시저 명령문(SQL 포함)을 실행할 수 있습니다. 계산을 수행할 수 있고 데이터베이스 테이블을 업데이트할 수 있으며 보고서 명령문 블록 내에서 보고서 출력을 포맷할 수 있습니다. 표 9-1과 9-2는 Informix-4GL 보고서 기능을 구성하는 명령문의 목록입니다.

사실상 다른 모든 Informix-4GL문은 거의 제한 없이 보고서 함수의 format 블록에서 실행될 수 있지만 일부 제한이 적용됩니다. close form, input, display 및 construct와 같은 화면 대화형문은 보고서의 본문 내에서는 실행될 수 없습니다. (이러한 제한에 대한 간단한 해결책은 화면 대화형문을 실행하는 함수를 호출하는 것입니다.)

표 9-1 보고서 명령문

명령문	기능
report report_name(parameter1, parameter2,)	보고서 블록의 시작을 나타냅니다.
define	보고서에 대해 로컬인 정적 변수를 정의합니다.
format	보고서로 전달 중인 데이터를 포맷하는 방법을 정의한 블록의 시작을 나타냅니다.
on every row	format 섹션 내에서 전달된 모든 행에 대해 데이터를 포맷하는 방법을 나타냅니다.
after group of / before group of field_name	format 섹션 내에서 보고서로 전달되고 있는 데이터 스트림에 대한 제어 중단을 나타냅니다.
on last row	format 섹션 내에서 보고서 끝에서 수행할 조치를 나타냅니다.

보고서 명령문

보고서에는 start report문에 의해 트리거되는 특정 시작과 finish report문에 의해 트리거되는 특정 종료가 있습니다. start report문은 출력할 파일이나 장치를 엽니다. 파일이나 장치로 데이를 출력하려면 output to report문을 사용합니다. 보고서에 출력할 각 조에 대해 이 명령문을 호출합니다. finish report문은 사용 중인 버퍼의 나머지를 출력 장치로 플러시하고 장치를 닫습니다. finish report문을 발행하지 못하면 보고서가 완료되지 않습니다(공통 오류). 이들 명령문의 결과는 표 9-2와 같습니다.

표 9-2 보고서 대화형문

명령문	기능
start report <report name>	보고서를 시작합니다. 선택적으로 출력 장치를 식별할 수 있습니다.
output to report <report name>	보고서 장치/파일로 출력을 보냅니다.
finish report <report name>	보고서를 완료한 후 출력 장치를 닫습니다.

변수는 define 섹션 내에 선언하며 다른 Informix-4GL 함수 변수와 달리 이들 변수는 연속 호출 사이에 값을 유지합니다. 이는 현재 Informix-4GL 내에서 유일한 정적 로컬 범위 변수의 예입니다.

보고서 블록은 report문을 사용하여 선언합니다. 보고서가 반복될 때마다 output to report문을 사용하여 보고서로 전달할 매개변수를 식별하는 함수와 비슷한 구문을 사용할 수 있습니다. 이들 매개변수는 보고서로 전달되어야 하며, 수 및 위치 순서가 일치해야 하고 호환 가능한 자료형을 가져야 합니다. output to report문에 있는 매개변수 수가 보고서 선언문에 선언된 매개변수 수와 일치하지 않으면 컴파일러는 치명적 오류 신호를 보냅니다. output to report문에 전달된 자료형을 변환할 수 없는 경우 치명적 런타임 오류 신호를 보냅니다. 그림 9-4는 보고서 명령문의 결과를 보여줍니다.

보고서 선언

보고서의 본문은 다음과 같이 Informix-4GL 함수의 구문과 비슷한 구문을 사용하여 선언합니다.

```
report sales( sales_rec )
define sales_rec record like sales_person.*
```

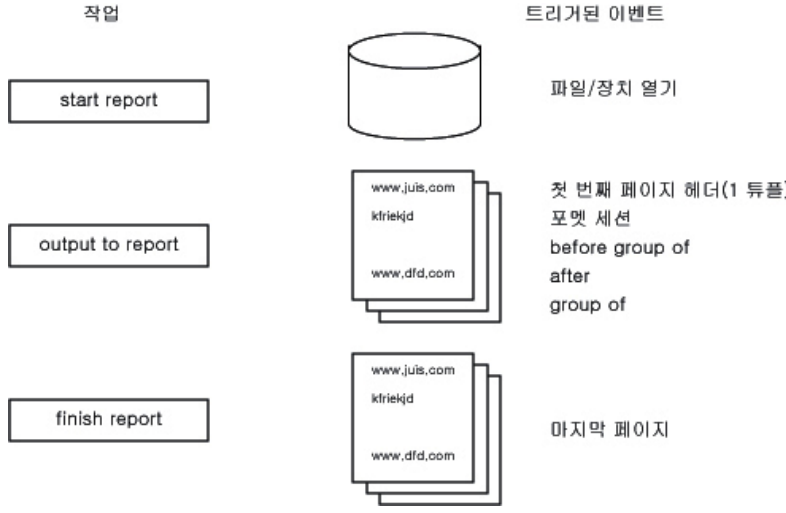


그림 9-4 보고서 명령문의 결과

이 코드는 보고서 sales_report를 선언하고 이 보고서가 sales_rec 레코드를 받을 것임을 나타냅니다. 보고서의 define 섹션에는 sales_rec이 sales_person 테이블의 구조와 동일한 구조를 가진 레코드로 식별되어 있습니다. 보고서 선언이 보고서를 시작하고 실행하는 모듈과 동일한 모듈에 나타날 필요는 없습니다. 내부적으로 보고서 선언은 다음과 같이 output to report문이 실행될 때 호출될 함수를 작성합니다.

```
start report sales_report

foreach report_curs into srec.*

output to report sales_report ( srec.* )

end foreach

finish report sales_report
```

이 예에서는 output to report문이 foreach 루프 내에서 실행됩니다. foreach 루프는 보고서를 생성하는 데 훌륭한 메커니즘을 제공합니다. 이 메커니즘은 보고서 매개변수가 작성된 다음, 보고서로 출력되는 루프를 작성하는데, 이는 모두 동일한 제어 블록 내에서 수행됩니다. foreach 루프가 시작되기 전에 start report문을 실행할 수 있습니다. foreach 루프가 종료되면 보통 finish report문이 실행됩니다.

기본 보고서 생성

Informix-4GL에는 코드를 거의 사용하지 않고 간단한 기본 보고서를 작성하는 기능이 있습니다. 생성된 보고서는 매우 간단하고 포맷팅이 부족하기는 하지만 보고서를 빠르고 쉽게 생성됩니다. 다음 코드는 이러한 기능을 보여줍니다.

```
database sales

main
define sr record like sales_person.*
```

```
declare c1 cursor for
     select *
     from sales_person

start report sales

foreach c1 into sr.*
     output to report sales( sr.* )
end foreach

finish report sales

end main

report sales( sales_rec )
define sales_rec like sales_person.*

format
     every row
end report
```

이 예에서는, 프로그램 모듈의 첫 번째 라인에서 레코드 선언에 사용할 데이터베이스가 sales 데이터베이스임을 컴파일러에 알려줍니다. 그런 다음, sales_person 테이블과 동일한 구조를 갖는 레코드를 선언합니다. 또한 이 테이블에서 모든 행과 해당 행의 모든 데이터 요소를 선택하도록 커서를 선언합니다. 그런 다음, foreach 루프를 시작하고 검색된 데이터를 기본 보고서로 출력합니다. 이 보고서의 출력은 다음과 같습니다.

```
first_name   Frank
last_name    Salesguy
territory    Buffalo
quota        1,000,000

first_name   Greg
last_name    Techguy
territory    SW
quota        1,000,000
```

output 섹션

보고서의 output 섹션은 이름이 나타내는 바와 같이 데이터가 출력될 위치를 정의하며 이 출력에 대한 기본적인 포맷팅 지시사항을 간단히 제공합니다. 이 보고서 섹션은 선택적이며, 이를 생략할 경우 다양한 기본값이 적용됩니다. 표 9-3은 보고서 output 섹션의 기본값 목록입니다.

표 9-3 output 섹션 기본값

매개변수	기본값
report to <character>	터미널 화면(stdout)
left margin <integer>	5
right margin <integer>	132
top margin <integer>	3
bottom margin <integer>	3
page length <integer>	66
top of page <character>	사용하지 않음

이들 옵션의 매개변수는 상수이어야 하며 변수는 대체할 수 없습니다. 이는 프로그램 컴파일 시간에 보고서 포맷 옵션을 강제로 정의하는 것으로 나타날 수 있지만, 다음 절에서 볼 수 있듯이 실제로 이들 매개변수는 동적으로 쉽게 조정할 수 있습니다.

report to문은 보고서의 출력을 특정 위치로 방향 지정하는 데 사용됩니다. 기본적으로 보고서 데이터는 연속 스트림으로 터미널 화면으로 보내진다는 점을 유의하십시오. pause문을 보고서 본문에 삽입하지 않으면, 보고서는 완료될 때까지 화면에서 계속 스크롤됩니다.

report to문이 보고서 본문에 사용되었고 start report문에 to 절이 있는 경우, start report문이 우선 순위를 가지며 보고서 본문의 report to문은 무시됩니다.

report to문은 출력 장치로 printer를 지정할 수 있습니다. 이 경우, 환경 변수 DBPRINT의 내용을 조사하여 이 환경 변수에 지정된 프로그램으로 출력을 보냅니다. Unix 운영 체제 하에서 이 환경 변수가 설정되지 않은 경우, 프로그램 "lp"가 수신 파이프로 열려져 데이터는 이 프로그램으로 출력됩니다.

output 섹션의 right margin 매개변수는 기본 보고서 내에서 또는 wordwrap절이 사용된 경우에만 유효합니다. 이외의 경우, right margin문은 Informix-4GL 보고서 작성기에서 무시됩니다. 이는 대부분의 경우 Informix-4GL이 오른쪽 여백을 시행하지 않는다는 것을 의미합니다. 다음 절들에 나와 있는 논리와 비슷한 논리를 사용하여 이를 수행하는 것은 프로그래머에게 달려 있습니다.

order by 섹션

order by문은 보고서 그룹화 명령문(before group of/after group of)이 실행되는 순서를 지정하는 데 사용됩니다. 여기에 보고서의 정렬 순서가 지정되어 있으면 보고서 프론트엔드가 데이터를 정렬합니다. Informix는 엔진 또는 백엔드 정렬의 속도를 개선하기 위해 상당한 노력을 기울여 왔습니다. 백엔드는 사용할 수 있는 인덱스를 활용하거나 병렬 정렬 스키마를 사용하거나 병합 정렬 기법을 사용할 수 있는데, 이러한 방법은 프론트엔드 정렬 기능에서 반드시 사용할 수 있는 방법이 아닙니다. 데이터가 예상된 순서대로 보고서에 출력될 경우 순서를 지정할 필요가 없습니다.

프론트엔드보다는 엔진으로 데이터를 정렬하는 것이 보다 효율적이기 때문에 정렬을 수행하는 것을 권장합니다. 이 작업은 SQL문에서 order by절을 사용하여 올바른 정렬 순서를 지정함으로써 수행될 수 있습니다.

format 섹션

format 섹션은 보고서의 본문을 나타냅니다. 이 섹션 내의 부속절은 보고서를 준비하는 동안 수행할 조치를 식별합니다. 이는 Informix-4GL 프로시저 또는 비프로시저 명령문을 이 블록 내에 사용하여 Informix-4GL 보고서 작성기를 간단한 보고서 이상에 사용할 수 있도록 합니다.

first page header 섹션

first page header 섹션은 보고서마다 한 번씩 실행됩니다. 이 섹션은 보고서 함수로 첫 번째 조를 전송함으로써 트리거됩니다. fi

rst page header는 page header 블록을 트리거하지 않으므로, 첫 번째 페이지에 대해 페이지 헤더를 인쇄해야 하는 경우 페이지 헤더 인쇄 명령문은 보고서의 first page header 섹션에도 나타나야 합니다. 이 섹션은 보통 첫 번째 페이지 헤더를 인쇄하는 데 유용하게 사용되지만 보고서에서 사용할 변수를 초기화하는 데도 유용합니다. 이에 대한 코드의 예는 다음과 같습니다.

```
define

  col1 integer,
  col2 integer,
  col3 integer,
  page_header1 char(50),
  page_header2 char(50)

first page header

- initialize variables
let col1 = 10
let col2 = 20
let col3 = 30
- create the page header strings to be printed on every page
let page_header1 = "Regional Sales Report"
let page_header2 = "Date: ", today, " Time: ", time

- print the page header
print col1, page_header1
print col1, page_header2
...
```

이 예에서 first page header 섹션은 보고서에 인쇄할 데이터에 대한 여러 컬럼 위치를 보유하는 데 사용할 정수 변수를 초기화할 때 사용됩니다. 그런 다음, 보고서의 페이지 헤더를 보유하기 위한 두 문자 문자열을 초기화합니다. 마지막으로 페이지 헤더를 인쇄합니다.

이 예는 보고서의 first page header 섹션 내에서 수행되는 변수의 초기화를 보여줍니다. 이들 변수는 보고서 함수의 로컬 변수이며 보고서 함수 외부에서는 볼 수 없습니다. 그러나 Informix-4GL의

다른 로컬 변수와 달리 보고서가 반복될 때마다 값을 보유합니다. 따라서 소계와 총계를 유지할 수 있으며 로컬 보고서 변수를 컬럼 위치를 보유하는 데 사용할 수 있습니다. 이 점에 대해서는 다음 절에서 설명합니다.

보고서에서 동적 컬럼 위치 작성

보고서를 개발하는 중에 보고서의 포맷을 계속해서 변경해야 하는 경우는 드물지 않습니다. 정수 변수를 사용하여 컬럼 위치를 보유하면 보고서를 쉽게 수정할 수 있습니다.

모든 컬럼 1 데이터가 정수 변수에 저장된 컬럼 위치에서 인쇄되는 경우, first page header 섹션에서 이 정수 변수의 값을 한 번 변경하여 보고서 전체에서 데이터를 효과적으로 이동할 수 있습니다. 이러한 방식으로 컬럼 위치를 저장하면, 필요할 경우 런타임 시 보고서에서 데이터를 이리저리 이동시킬 수 있습니다.

이 방법을 변형한 방법은 다음과 같습니다. 이 방법을 사용하는 경우 모든 컬럼 위치는 시작 컬럼(컬럼 1) 위치와 관련됩니다. 이러한 방식으로 한 컬럼에 대한 컬럼 위치를 조정하면 나머지 컬럼의 컬럼 위치가 해당 컬럼의 오른쪽으로 자동 조정됩니다. 이 기법은 다음과 같습니다.

```
first page header

     let col1 = 3
     let col2 = col1 - 10   - sales man first name
     let col3 = col2 + 5    - sales man last name
     let col4 = col3 + 5    - region name
     let col5 = col4 + 10   - sales amount in that region

on every row
     print column col1, sr.first_name,
           column col2, sr.last_name,
           column col3, sr.region_name,
           column col4, sr.sales_amount
```

이 예에서는 정수 상수 값이 지정되는 첫 번째 컬럼을 제외하고 컬럼 위치가 모두 이전 컬럼의 컬럼 위치와 관련되어 설정됩니다.

따라서 보고서의 수정이 간단해집니다. 판매원 이름의 길이를 현재의 10자 대신 15자로 만들기 위해 보고서를 조정하려는 경우, 다음과 같이 col2 지정 명령문만 수정하면 됩니다.

```
let col2 = col1 + 15
```

이 지정은 컬럼 2의 시작 컬럼 위치에 다섯 개의 컬럼 위치를 더합니다. 컬럼 2보다 큰 모든 컬럼에 컬럼 2와 관련된 컬럼 위치가 지정되기 때문에, 이들 컬럼을 다섯 개의 컬럼 위치로 밀어넣는 효과를 지닙니다.

format 섹션 – before group of/after group of 절

Informix-4GL 보고서 기능은 before group of/after group of 명령문을 사용하여 보고서 제어 중단 기능을 제공합니다. 그룹화 논리는 데이터가 정렬된 형태로 보고서에 보내지고 있는 것으로 가정하며, 이 그룹화의 기본 순서는 format 섹션에 나타나는 순서입니다. 앞에서 언급한 바와 같이 이를 무시하고 보고서의 output 섹션에서 order by절을 사용하여 특정하게 지정할 수 있습니다. 이 절이 외부 키워드 없이 사용된 경우 보고서는 들어오는 매개변수를 자체적으로 정렬하여 보고서 데이터가 정렬되도록 합니다. 보고서는 데이터가 보고서로 어떻게 보내지는 알 수 없으므로 전체 정렬을 수행합니다.

보고서의 본문 내에 다양한 집계 함수를 사용할 수 있습니다. 이러한 함수는 보고서를 생성하는 데 필요한 코드를 줄일 수 있는 강력한 비프로시저 기능을 제공합니다. 함수의 목록은 표 9-4에 나와 있습니다.

표 9-4 집계 함수

함수 이름	용도
sum	필드의 합계를 계산
avg	필드의 평균을 계산
count	전달된 모든 행 수를 계산
percent	총계의 백분율을 계산(count(*)/total)
min	필드의 최소값
max	필드의 최대값

집계는 where절의 사용도 지원합니다. 이 절을 사용하면 집계 함수를 계산하는 데 사용될 데이터를 필터할 수 있습니다. where절 기준을 충족시키는 데이터만 집계 계산에 사용하고 다른 모든 행은 무시합니다. 집계에 대해 사용하는 where절의 구문은 SQL문에 사용하는 where절의 구문과 비슷합니다. 여러 부울 연산자를 사용하여 행이 승인되기 전에 충족시켜야 하는 조건 목록을 작성할 수 있습니다. 다음의 예는 집계 그룹 함수와 함께 where절을 사용하는 예를 보여줍니다.

```
...

on last row

print "New Jersey Sales Person Count: ", count(*) where
sales_man.region = "NJ"
```

이 예에서 판매원의 지역이 "NJ"인 행의 수는 보고서의 마지막 행에서 구할 수 있습니다. 이 계수는 Informix-4GL 집계 함수인 count(*)와 연결한 where절을 사용하여 구할 수 있습니다.

보고서의 배치 처리

다양한 처리는 before group of/after group of절 내에서 수행할 수 있습니다. 생산 시스템에서는 일반적으로 보고서 실행 중에 배치 처리를 수행합니다. 이는 보고서가 생성될 때 데이터베이스 테이블이 업데이트되는 전형적인 배치 처리 보고서 실행을 나타냅니다. 다음은 이러한 보고서의 예를 보여줍니다.

```
{schema for acct_master and acct_balances table}

create table acct_master ( acct_code char(10),
                           sub_code chat(3)
                           acct_group char(3),
                           acct_name char(15)  );
create table acct_balances ( acct_code char(10),
                           acct_group char(3),
                           acct_period smallint,
```

```
                                acct_balance decimal(12,2)  );

____________________________

{function and report for acct_group report }
function acct_rpt_drv()

define accts record like acct_master.sub_code,*

declare c1 cursor for
    select acct_master.acct_code,
        acct_master.acct_name,
        acct_master.acct_group,
        acct_balances.balance
    from acct_master,
        acct_balances
    where acct_master.acct_code = acct_balances.acct_code and
        acct_balances.period = 1
    order by acct_master.acct_code,
        acct_master.acct_group
start report accts

foreach c1 into accts.*
    output to report accts( accts.* )
end foreach
finish report accts

end function

_____________________________

report accts( act_rec )
define act_rec record like acct_master.*
define sum_amt array[2] like acct_master.balance

output
    left margin 5
    top of page "^L"

format
```

```
after group of acct_code
        let sum_amt[1] = group sum(balance)

      print "Acct Code Balance: ", sum_amt[1]
      update acct_subs
      set  sub_bal = sum_amt[1]
      where sub_code = act_rec.sub_code

after group of acct_group

      let sum_amt[2] = group sum(balance)
        print "Acct Group Balance: ", sum_amt[2]

      update acct_groups
      set   group_bal = sum_amt[2]
      where group_code = act_rec.acct_group

on last row

            print file "report.trailer"

end report
```

이 보고서의 목표는 계정 코드 및 계정 그룹의 계정 잔액과 총액을 결정하기 위한 것입니다. 먼저 acct_master 테이블을 정렬된 순서로 읽습니다. 계정 코드 및 계정 그룹의 제어 중단에서 잔액 소계가 각 해당 테이블에 저장됩니다. (이러한 롤업 테이블을 사용하는 이유는 계정 테이블을 스캔하여 정보를 검색하는 것보다 부속 계정 테이블에서 행을 검색하는 것이 더 빠르기 때문입니다.)

보고서는 계정 코드와 계정 그룹의 순서대로 간단하게 acct_master 테이블에서 데이터를 검색하는 SQL문에 대한 커서를 선언함으로써 시작됩니다. 데이터는 보고서가 예상한 순서대로 검색되기 때문에 보고서 정렬 기능을 사용하여 데이터를 정렬할 필요가 없습니다. 또한 after group of문이 format 섹션에 나타나는 순서는 그룹화가 실행되어야 하는 순서이기 때문에 format 섹션에서 order by절을 사용하여 순서를 지정할 필요가 없습니다. 이에 해당되지 않는 경우, 명시적 order by절이 보고서의 output 섹션에 포함됩니다.

after group of 블록에서 잔액의 소계를 구해 Informix-4GL 변수에 저장합니다. 나중에 이 변수는 소계로 acct_subs 및 acct_

groups 테이블을 업데이트하는 데 사용됩니다. 소계는 Informix-4GL 보고서 기능의 내장 집계 명령문 즉, group sum문을 사용하여 구합니다.

on last row절의 명령문은 finish report문에 의해 트리거됩니다. 앞의 예에서 on last row절에는 보고서 트레일러를 인쇄하기 위한 print file문을 사용하는 명령문이 포함되어 있습니다. print file문은 보고서의 현재 위치에 지정된 텍스트 문서를 인쇄합니다.

print 명령문

print문은 using절을 사용합니다. 이 명령문은 보고서 블록의 format 섹션에서만 인식됩니다. print문 내에서 이 using절의 구문은 다음과 같습니다.

```
print column 10, acct_name, col 15,
      acct_balance using "Balance:    $<<<<<.<<"
```

보고서 인쇄에서 어려운 점 중 하나는 중요한 정보를 함께 보존하여 여러 페이지에 인쇄되지 못하도록 하는 것입니다. Informix-4GL을 사용하면 다음과 같이 need문으로 이러한 작업을 수행할 수 있습니다.

```
after group of acct_sub

   need 3 lines
   print "Account Name: ",  accts.acct_name
   print "Account Group Total: ", sum_amt[2]
   print "Sub Account Total: ", sum_amt[1]
```

이 예에서는 세 라인의 정보를 보고서에서 동일한 위치에 함께 인쇄하는 것이 바람직합니다. 이 정보를 여러 페이지에 분산시키면 읽기 어려워집니다. need문은 이들 라인을 함께 보존하여 필요할 경우 강제로 페이지 구분을 수행하는 데 사용됩니다.

보고서의 텍스트 필드 관리

Informix-4GL은 보고서 내에서 BLOB 자료형을 관리할 수 있도록 몇 가지 유용한 명령문을 제공합니다. 일반적으로 BLOB 자료형이 Text 컬럼인 경우 보고서 작성기는 자료형을 관리할 수 있는 기능을 제공합니다. BLOB 자료형이 Binary인 경우 이 자료형을 관리하기 위한 기능은 제공되지 않습니다. Binary BLOB을 처리하려면 함수(대부분 C 함수임)를 호출해야 할 것입니다.

Text 자료형은 크기가 2GB로 제한된 가변 길이 필드를 제공하는 BLOB 자료형의 서브세트입니다. 이 필드의 내용은 인쇄할 수 있는 ASCII 문자 세트와 제어 문자 ^ L(라인 바꿈) 및 ^ J(용지 넘김)로 제한됩니다. Informix-4GL 버전 4.1의 인쇄 명령문에 추가된 확장 기능을 사용하여 이 자료형을 포맷할 수 있습니다. 이들 확장 기능은 다음과 같습니다.

```
format
    on every row
    print "Description : ", description_field
                        wordwrap compress right margin 80
```

이 예에서 description_field는 BLOB Text 자료형이며, 오른쪽 여백에서 줄을 바꾸는 wordwrap절을 사용하여 인쇄됩니다. compress 옵션은 단어 사이에 하나의 공백만을 남기고 필드에서 여분으로 남는 영역을 압축하는 데 사용됩니다. 오른쪽 여백 기본값은 132byte이지만 right margin절을 사용하여 강제로 80컬럼으로 인쇄하도록 하므로 이 BLOB 필드에서는 무시됩니다. (현재 print 섹션의 컬럼 위치는 wordwrap 옵션이 사용될 때 무시된다는 점을 유의하십시오.)

페이지 구분

Informix-4GL로 페이지 구분을 강제 실행하는 기본적인 방법은 페이지의 길이까지 공백 라인을 인쇄하는 것입니다. 이 방법은 속도가 느릴 뿐만 아니라 오류가 발생하기 쉽고 페이지 길이가 정확하지 않으면 연속 페이지가 인쇄될 때 페이지 헤더가 페이지의 아래로 밀려날 수 있습니다. 프린터를 다음 페이지로 이동시키는 데 좋은 방법은 페이지 넘김 문자인 Control-L을 사용하는 것입니다. 이는 보

고서의 output 섹션에서 다음과 같은 구문으로 지정할 수 있습니다.

```
output
top of page "^L"
```

이 명령문은 Informix-4GL 보고서 작성기로 하여금 Control-L 문자를 사용하여 다음 페이지로 이동하도록 지시합니다. 실제로 모든 프린터는 이 제어 문자를 사용하여 다음 페이지로 이동하도록 지원합니다.

페이지 구분은 보고서 블록의 page header 섹션을 트리거하며, 페이지 헤더를 생성하는 코드가 여기에 들어 있습니다. 보고서의 포맷을 제어하고 일관성을 보장하려는 목적으로, 이 블록에서 사용된 print문을 포함하는 if문은 if/then/else문의 두 부분에서 모두 동일한 라인 수를 인쇄합니다. 다음의 예가 이러한 동작을 보여줍니다.

```
page header

    if rpt_rec.acct_group = 3 then
        print "Report Group Three being printed"
else
      let counter = count+1
end if
```

이 명령문은 허용되지 않습니다. Informix-4GL 보고서 작성기는 페이지 헤더가 항상 일관된 크기일 것으로 예상합니다. 이 명령문이 허용될 경우 보고서가 균형이 맞지 않는 포맷팅 결과를 작성하게 될 것입니다. 이 실행을 위해서는 다음과 같은 코드가 필요합니다.

```
...
page header
    if rpt_rec.acct_group = 3  then
        print "Report Group Three being printed"
    else
      skip 1 line
      let counter=count+1
    end if
```

이 예에서는 if/then문의 else절에서 한 라인을 인쇄(스킵)하여 보고서 출력의 균형을 유지합니다. 그러면 이 절 위의 then절에서 인쇄되는 한 라인을 효과적으로 보충할 수 있습니다.

한 페이지에 보고서 정보를 함께 보존해야 하거나 정보를 인쇄하는 데 전체 페이지가 필요한 경우 skip to top of page문을 사용할 수 있습니다. 그러면 프린터가 다음 페이지로 강제로 이동됩니다. 다음의 예는 이 명령문의 사용을 보여줍니다.

```
after group of acct_group

    - print accounts rolled into account group
    if num_accounts > 40 then – print on a separate page
        skip to top of page
    end if
    for n = 1 to num_accts
    - print one line for each account

....
```

이 예에서는 부속 계정에 대한 제어 중단 이후에 루프가 실행되어 해당 계정 그룹에 포함된 계정의 목록을 인쇄합니다. if/then문을 사용하여 인쇄할 계정의 수를 구합니다. 계정 수가 40을 초과하면 보고서는 skip to top of page문을 사용하여 페이지 구분을 강제로 실행합니다.

계정 세부 정보를 인쇄하는 목적이 모든 계정 정보를 한 페이지에 함께 보존하기 위한 것일 경우 위의 방법은 소용이 없습니다. 페이지에서 현재 위치는 고려되지 않습니다. 이 정보를 구하는 보다 나은 방법은 다음과 같습니다.

```
report sub_acct_rpt( page_length, rec )

    define page_length integer,
        rec record like …

…

after group of sub_accounts
- determine whether or not there is room
- print the list of accounts
```

```
    if ( page_length - lineno - num_accounts ) <= 0
    then
    skip to top of page
end if

- print the information
for n = 1 to num_accts
 ...
```

이 예에서 보고서의 page_length는 매개변수로 제공됩니다. 또한 이 매개변수 이외에 페이지의 현재 라인 번호(lineno)와 인쇄할 계정의 수를 함께 구합니다. 등식(page_length - lineno - num_accounts)으로 결정된 모든 계정을 모두 인쇄할 여유 공간이 없는 경우 skip to top of page문을 실행하여 다음 페이지로 이동합니다.

모듈식 동적 보고서

런타임 시 제공된 매개변수에 기초하여 보고서의 포맷을 제어하는 모듈식 접근법은 모든 관련 매개변수를 레코드에 넣어 이 레코드를 보고서에 전달하도록 수행할 수 있습니다. 이 방법을 나타내는 예는 다음과 같습니다.

```
database sales

main
define rpt_control record
                out_file char(10),
                page_length  integer,
                right_margin  integer,
                left_margin  integer
        end record

define sales_rec record like sales_person.*

let rpt_control.out_file     = "SCREEN"
let rpt_control.page_length  = 22
let rpt_control.right_margin = 80
let rpt_control.left_margin  = 3
```

```
start report sales_rpt

declare c1 cursor for
      select *
      from   sales_person
foreach c1 into sales_rec.*

    output to report sales_rpt( rpt_control.*, sales_rec.* )

end foreach

finish report sales_rpt

end main
_____________________________
report sales_rpt( rc, sr )

define rc record
                out_file char(10),
                page_length  integer,
                right_margin  integer,
                left_margin  integer
        end record
define sr record like sales_person.*
output
    page length  55
    right margin  0
    left margin   0
    top margin    0
    bottom margin 0

format

  on every row
    print "Sales Person: ", sr.first_name, " ", sr.mi," ",
                            sr.last_name

    if lineno = rpt_control.page_length then
```

```
    - automatically handle output to the screen

    if rpt_control.out_file = "SCREEN"  then
       pause "Press Return to Continue"
    end if

       skip to top of page

 end if
  ...
```

이 예에서는 보고서를 호출하기 전에 보고서 매개변수가 레코드에 저장됩니다. 이 레코드에는 페이지 길이, 출력 파일 이름, 오른쪽 여백 및 왼쪽 여백에 대한 항목들이 있습니다. 보고서의 output 섹션에서 페이지 길이는 시행될 실제 페이지 길이보다 큰 수로 설정됩니다.

페이지 길이 매개변수는 데이터가 인쇄될 때 평가됩니다. 내장 변수 lineno에 들어 있는 페이지의 현재 라인 번호가 지정된 페이지 길이보다 큰 경우, 보고서는 skip to top of page문을 사용하여 다음 페이지의 처음으로 건너뜁니다. 그리고 제어 레코드에 입력된 page_length 매개변수가 보고서 output 섹션에 있는 페이지 길이 설정보다 크다고 가정하고, Informix-4GL 보고서 작성기의 제어에서 페이지 구분을 제거합니다.

보고서 제어 레코드에서 출력 장치 설정도 조사합니다. 출력 장치가 "SCREEN"으로 설정되어 있으면 pause문을 실행하여 사용자가 다음 화면이 표시되는 것을 볼 준비가 되었을 때, Enter를 누를 수 있도록 합니다. 이로 인해 보고서는 화면 출력과 프린터 장치를 동적으로 지원할 수 있습니다. (이 출력이 화면에 나타나지 않는 경우 pause는 아무 효과가 없으므로 이 조건절이 필요하지 않습니다. 그러나 화면 출력에 고유한 다른 코드를 여기에 삽입할 수 있습니다.)

보고서 문제

보고서의 포맷이 끊임없이 변화하는 것은 드문 일이 아닙니다. 사용자들은 종종 매주 다른 보고서 포맷을 요청합니다. 약간의 전망과 계획만 있으면 보고서를 쉽게 수정할 수 있으며 이는 다음 예로 확인할 수 있습니다.

보고서의 오른쪽 여백과 페이지 길이는 출력 데이터와 출력 장치에 따라 매우 다를 수 있습니다. 오른쪽 여백이 변경될 때 이 변경에 맞게 조정하는 것은 보고서 코드에 달려 있습니다. 다음의 예는 이러한 포맷 매개변수의 변경을 관리하는 방법을 보여줍니다.

```
 ...
first page header

    let header1 = "Accounts Report "
    let pos = right_margin - length( header1 clipped)  / 2
    let header1 = spaces pos, header1 clipped
    let header1[1,16] = "Date: ", today
    let header1[right_margin - 12, 12 ] = "Time: ", time

    let header2= "Group Breakdown"
    let pos = right_margin - length( header2 clipped ) / 2
    let header2 = spaces pos, header2 clipped
    let header2[ right_margin - 9, 9] = pageno using "Page:
<<<"
page header
    let header2[right_margin - 9, 9] = pageno using "Page:
<<<"
    print column col1, header1 clipped
    print column col1, header2 clipped
```

이 예에서는 오른쪽 여백이 변할 수 있으므로 보고서에 대한 페이지 헤더를 준비할 때 매개변수로 처리됩니다. 현재 오른쪽 여백으로 제목을 문자열의 가운데에 맞추는 등식을 사용하여 보고서 제목을 문자열의 가운데에 놓습니다. 이 등식은 다음과 같습니다.

```
pos = ( right_margin - length_of_string ) / 2
```

이 등식은 오른쪽 여백에서 가운데가 되는 시작 컬럼 위치를 식별하고, 원하는 위치까지 공백으로 채워서 이 위치에 놓을 문자열의 길이를 고려합니다. 다음 Informix-4GL문은 가운데에 맞출 문자열을 이 위치에 놓습니다.

```
let header1 = spaces pos, header1 clipped
```

이 명령문은 내장 공백 함수를 사용하여 가운데에 맞출 문자열의 시작 위치까지 문자열을 공백으로 채웁니다. 그런 다음, 가운데에 맞출 문자열(이 경우, 문자열 변수 header1의 내용)이 이 문자열에 연결됩니다. 페이지 헤더와 시간도 Informix-4GL의 내장 문자열 처리 연산자를 사용하여 이 문자열에 삽입됩니다.

두 헤더 문자열이 포맷된 다음에는 변수 오른쪽 여백을 고려하여 보고서의 page header 섹션에 인쇄됩니다. 헤더 문자열은 보고서가 시작될 때 실행된 first page header 섹션에서 포맷되었기 때문에 페이지 헤더가 인쇄될 때마다 다시 포맷할 필요가 없습니다. 그러나 페이지 번호는 변경되므로, 각 page header 블록이 반복될 때마다 이를 문자열에 삽입해야 합니다.

헤더 외에도 보고서의 행별 출력에서는 오른쪽 여백의 변경을 고려해야 합니다. 다음 코드의 예가 이를 나타냅니다.

```
on every row

    let row1_data = "Account Name: ", Acct_name …

    if right_margin > 80  then
     let row1_data = row1_data clipped, "Additional info: ",
         acct_info.group, column col3,
         acct_info.acct_status
         let row2_data = NULL
else
      let row2_data = "Additional Information: ",
                      acct_info.group, column col3,
        acct_info.acct_status
end if

    print column col1, row1_data
    if row2_data IS NOT NULL then
       print column col1, row2_data
    end if
```

이 예에서는 오른쪽 여백의 길이와 관계없이 인쇄될 데이터를 문자열에 넣습니다. right_margin 변수를 평가하여 길이를 결정합니다. 길이가 추가 데이터 인쇄를 지원하기에 충분하면 데이터는 row1_data 문자열에 추가되고 row2_data 문자열은 NULL로 설정됩니다. 길이가

충분하지 않으면 추가 데이터는 row2_data 문자열에 놓입니다. 마지막으로 row1_data 문자열이 인쇄되고, NULL이 아닐 경우 row2_data 문자열이 인쇄됩니다.

보고서의 모듈성

보고서의 모듈성을 향상시키기 위해 몇 가지 일반적인 규칙을 적용할 수 있습니다. 가능한 한, 보고서의 본문 내에는 로컬 변수를 사용해야 합니다. 로컬 보고서 변수는 정적 로컬 범위 변수이기 때문에 Informix-4GL에서 전역 변수에 대한 가장 일반적인 요구 중 하나가 제거됩니다. 보고서 함수로 전달해야 하는 값의 수를 줄이기 위해 전역 변수를 사용하고 싶은 유혹이 종종 있지만 이러한 습관을 줄이면 보고서의 모듈성이 향상되고 보고서가 보다 읽기 쉬워집니다.

앞에서 언급한 바와 같이 보고서의 first page header 섹션은 보고서에서 사용될 변수를 초기화하는 데 유용합니다. 대부분의 Informix-4GL 프로그래머들은 이 섹션에서 초기화가 수행될 수 있다는 것을 알고 있으므로 보고서의 다른 곳보다는 이 블록에서 초기화를 수행하는 것이 바람직합니다. 로컬 변수와 모듈 범위 또는 전역 변수를 이 블록에서 초기화할 수 있습니다. 보고서 전체에서 사용될 문자열이나 변수를 이 섹션에서 초기화하거나 포맷하는 것도 권장합니다. 예를 들어, 복잡한 페이지 헤더를 보고서 전체에서 일관되게 인쇄할 경우 일련의 문자열에 first page header 섹션에 있는 페이지 헤더의 값을 지정한 다음 보고서의 각 페이지에 간단하게 인쇄할 수 있습니다.

보고서는 점점 더 복잡해지기 때문에 보고서 내에서 모듈 구조를 유지하기가 더욱 어려워집니다. print문은 보고서의 본문 내에서만 인식되기 때문에 특정 제어 블록에서 필요한 정보를 인쇄하기 위한 함수를 호출할 수 없습니다. (함수 내의 print문을 실행하려고 시도하면 컴파일러 구문 오류가 발생합니다.) 따라서 복잡한 보고서의 본문이 모듈 포맷의 권장량을 훨씬 초과하여 수천 라인 이상으로 증가하는 일이 다반사입니다. 보고서 본문 내의 코드 라인 수를 줄이고 모듈성을 향상시키기 위해 보고서에 인쇄되는 문자열을 로드하는 함수를 호출할 수 있습니다. 이 기법은 print문에서 사용되었던 동일한 using절을 사용하여 Informix-4GL let문이 포맷되도록 합니다. 다음의 코드 샘플이 이 방법을 보여줍니다.

```
report accts( jrec, acctrec )
…
define line1_string char(132)

after group of acctrec.sub_accts
      let gsum = group sum(acctrec.balance)
      call sub_accts( gsum, jrec, acctrec )
               returning line1_string

      print column 1, line1_string
…

function sub_accts( gsum, jrec, acctrec )

define ret_str1
define jrec like journals.*
define acctrec like accts_master.*
define gsum, totals decimal

select jour_name, jour_type, acct_type
 into  jrec.jour_name, jrec.jour_type, acctrec.acct_type_name
from journals, accts_crx_ref, accts_master
where journals.journal_number =
 accts_crx_ref.journal_number  and
 accts_crx_ref.acct_number =
 accts_master.acct_number

let ret_str1 = gsum using "Sub-Accts total: <<<<",
               " Journal Activity ",
jrec.jour_name," ", jrec.total, jrec.jour_type

return ret_str1

end function
```

이 예에서는 여러 레코드가 보고서로 전달됩니다. 두 개의 문자열이 Informix-4GL 보고서의 기본 폭으로 정의됩니다. 이들 문자열의 내용은 함수 sub_accts에 대한 호출을 통해 로드됩니다. 이 함수는 일련의 select문을 실행하여 데이터를 검색하고 문자열에 이 데

이터를 로드한 다음 이들 문자열을 호출 함수로 리턴합니다.

보고서의 본문에 직접 select문을 코딩하지 않고 함수를 호출하여 보고서의 모듈성을 더욱 유지하고 sub_accts 함수의 세부 정보를 보고서 프로그램 블록과 격리시킵니다.

보고서 동기화

동일한 데이터 스트림을 사용하여 두 개의 다른 보고서를 작성할 수 있는 기능은 실제로 매우 필요한 기능입니다. Informix-4GL은 이러한 보고서 동기화를 지원합니다. 즉, 다음의 코드에서와 같이 두 개의 다른 보고서로 동시에 출력을 보낼 수 있습니다.

```
main

define sr record like sales_person.*

declare c1 cursor for select *
    from sales_person

start report sales1
start report sales2
foreach c1 into sr.*

    output to sales1( sr.* )
    output to sales2( sr.* )

end foreach

finish report sales1
finish report sales2

end main
```

이 예에서 두 개의 다른 보고서인 sales1과 sales2가 시작됩니다. foreach 루프 내에서 sales_person 테이블로부터 데이터를 검색한 다음 두 개의 보고서로 출력합니다. foreach 루프가 끝나면 두 보고서는 모두 finish report문으로 완료됩니다.

construct 명령문 중심의 동적 보고서

많은 보고서들은 동적인 성격을 지니고 있습니다. 즉, 요구사항이 끊임없이 변경됩니다. 많은 MIS 조직들은 이러한 동적 보고서에 대한 보고서 작성과 수정에 상당히 많은 프로그래밍 자원을 투입합니다. 그러나 동적 보고서로 간주되는 것은 실제로 동적 선택 기준을 가진 보고서입니다. 보고서의 포맷은 실제로 동일하게 유지되며 선택 기준만 변경됩니다. 이러한 보고서를 작동시키는 데 필요한 다양한 선택 기준을 처리하기 위한 매우 유용한 기능은 construct문(그림 9-5 참조)이 제공하는 Query-By-Example(QBE) 기능입니다. 다음의 예가 이러한 유형의 보고서를 보여줍니다.

```
Enter Report Criteria. Press <Escape> when finished.

      Account Number    [                ]
      Accounting Month  [            ]
      Accounting Year   [            ]
      Total             [                    ]
```

그림 9-5 보고서 기준에 대한 construct문

많은 조직들은 상당한 수의 장부 계정을 사용합니다. 이들 계정에 대한 보고서를 작성해야 하는 경우 작은 서브세트의 계정만 조사하는 경향이 있습니다. 예를 들어, 경영진이 지출을 결정하려는 경우 지출 계정에 대해서만 계정 보고서를 실행하게 됩니다. 또는 경영진이 1월 동안의 작은 지출 계정 세트에 대해서만 지출을 조사하도록 검토 범위를 더욱 좁힐 수 있습니다. 가능한 각 선택 기준 세트에 대해 개별 보고서를 작성하기보다는 간단하게 단일 보고서를 작성하고 다른 선택 기준을 사용하여 보고서를 작동시키는 것이 더욱 합리적일 수 있습니다. 다음의 예처럼 construct문을 사용하여 보고서를 작동시키면 쉽게 작업할 수 있습니다.

이 예의 기본 프로그램 블록은 단지 두 개의 함수를 호출합니다. 화면 폼을 표시하고 construct문을 실행하는 qry_acct_totals 함수와 construct문의 결과를 받고 커서를 연 다음 보고서로 데이터를

출력하는 rpt_drv 함수가 이에 해당합니다.

```
database accounting

main
define where_clause char(500)

call qry_acct_totals() returning where_clause
call rpt_drv( where_clause, "acct_totals.out" )

end main
```

qry_acct 함수

qry_acct 함수는 보고서에 대한 선택 기준을 검색하는 데 사용됩니다. 먼저 윈도우를 연 다음 화면 폼을 표시하고 기준을 입력하는 방법을 사용자에게 설명하는 메시지를 표시합니다. 그런 다음, construct문이 실행됩니다. construct문이 완료되면 보고서를 작동시키는 데 사용될 select문의 where절에 대한 기준이 where_clause 문자열 변수에 놓입니다. 그런 다음 윈도우가 닫히고 where_clause 변수의 내용이 호출 함수로 리턴됩니다.

```
function qry_acct_totals()
define where_clause char(500)
open window w1 at 5,5
  with 12 rows, 60 columns
  attribute( border )

open form f1 from "acct_tot"
display form f1
message "Enter report criteria. Press <Escape> when finished."
attribute ( reverse )

construct by name where_clause on acct_totals.*

close window w1
return where_clause clipped

end function
```

rpt_drv 함수

rpt_drv 함수는 이전 함수에 의해 준비된 where절 기준과 보고서에 의해 작성될 출력 파일의 이름을 받습니다. 보고서를 작동시키는데 사용될 쿼리, 데이터베이스에서 데이터를 받을 레코드 및 처리되는 계정의 이름을 보유하기 위한 로컬 문자열 변수를 정의합니다.

이 함수에서 수행되는 첫 번째 단계는 준비될 쿼리 문자열을 빌드하는 것입니다. 함수에 문자열로 전달된 where절 기준은 쿼리 명령문의 select절의 문자열 상수에 추가됩니다. 호출될 보고서에는 고정 레코드 구조를 예상하는 정적 보고서 포맷 명령문이 들어 있기 때문에 이 select절은 상수가 되어야 합니다.

쿼리 명령문이 작성되었으면 기본 오류 검사가 해제되고 쿼리 명령문이 준비됩니다. prepare문 다음에는 sqlca.sqlcode가 검사됩니다. 오류가 발생하면 오류 메시지가 표시되고 함수는 리턴됩니다. prepare문이 성공적으로 실행되었으면 준비된 명령문에 대해 커서가 선언됩니다.

또한 커서가 성공적으로 선언되면 보고서가 시작됩니다. 보고서는 함수로 전달된 파일 이름으로 출력됩니다. 그런 다음 데이터베이스에서 데이터를 받아서 데이터를 보고서로 출력할 foreach문이 시작됩니다. 오류가 발생하면 오류 메시지가 표시되고 함수는 리턴됩니다.

```
function rpt_drv( where_clause, file_name )
define where_clause char(500),
       file_name     char(50),
       qstr        char(1000)
define at_rec record like acct_totals.*
define acct_name like accounts.acct_name

- build the query
let qstr = "select acct_totals.*, accounts.acct_name from ",
            " acct_totals, accounts where ",
             where_clause clipped,
            " and accounts.acct_num = acct_totals.acct_num ",
            " order by acct_totals.acct_num,<=
                     acct_totals.acct_year, ",
            " acct_totals.acct_month"
```

```
whenever error continue
prepare s1 from qstr
if sqlca.sqlcode < 0 then
   call err_print( sqlca.sqlcode )
   call err_msq( "Error on prepare." )
   return
end if

declare rpt_curs cursor for s1
if sqlca.sqlcode < 0 then
   call err_print( sqlca.sqlcode )
   call err_msg( "Error on declare." )
   return
end if

start report acct_totals_rpt to file_name

- main output loop
foreach rpt_curs into at_rec.*, acct_name

   if sqlca.sqlcode < 0 then
      call err_print( sqlca.sqlcode )
      call err_msg( "Error on fetch." )
      return
   end if

   output to report acct_totals_rpt( at_rec.*, acct_name )

end foreach

finish report acct_totals_rpt

end function
```

acct_totals 보고서

acct_totals 보고서는 계정 총계 레코드와 처리 중인 현재 계정의 이름을 수신합니다. 보고서 레코드가 출력될 컬럼의 컬럼 위치를

보유하기 위해 변수가 선언됩니다. 보고서 헤더 문자열을 보유하기 위한 변수도 선언됩니다.

보고서의 output 섹션에서 왼쪽 여백은 영(0)으로 설정됩니다. 그러면 Informix-4GL 보고서 작성 기능에서 여백 제어가 제거됩니다. 여백 제어는 보고서 데이터에 대한 컬럼 위치를 보유하는 로컬 변수에 의해 유지보수됩니다.

first page header 블록에서 페이지 헤더를 인쇄하는 데 사용되는 변수를 초기화합니다. all 함수는 일련의 동일한 문자로 구성된 문자열을 작성하는 데 사용됩니다. 이 경우, 페이지 헤더에 라인을 인쇄하는 데 사용될 문자열에 일련의 "=" 문자가 저장됩니다. 이러한 문자열은 first page header 블록에 로드되므로 페이지 헤더가 인쇄될 때마다 헤더 문자열을 포맷하는 오버헤드가 일어나지 않습니다.

컬럼 위치는 첫 번째 페이지 헤더 블록에도 로드됩니다. 이러한 방식으로 컬럼 위치를 로드하면 응용프로그램 개발 및 유지보수 프로세스 중에 쉽게 수정할 수 있게 됩니다. 모든 컬럼은 이전 컬럼(현재 컬럼의 왼쪽)과 관련되어 선언된다는 점을 유의하십시오. 따라서 두 번째 컬럼을 변경하면 보고서 라인에서 순서대로 뒤에 나타나는 모든 컬럼에 영향을 줍니다.

페이지 헤더 문자열이 모두 로드되면 인쇄됩니다. first page header 섹션이 보고서에 나타나는 경우 Informix-4GL은 보고서의 page header 섹션을 실행하지 않으므로 페이지 헤더 문자열이 여기에 인쇄되어야 합니다. 이 문자열은 일련의 인쇄 명령문을 사용하여 col1 변수에 정의된 컬럼 위치에 인쇄될 것입니다. 이러한 동일한 인쇄 명령문은 page header 섹션에서 다시 실행됩니다.

그런 다음, 컬럼 헤더가 들어 있는 문자열이 인쇄됩니다. 컬럼 위치 문자열을 사용하면 이들 헤더를 정확한 위치에 보다 쉽게 인쇄할 수 있습니다. 다음 예에서는 이 헤더를 해당 컬럼에 인쇄될 10진수의 포맷팅과 정렬시키기 위해 총계 라인이 오른쪽으로 여덟 칸 옮겨졌습니다.

```
report acct_totals_rpt( acct_rec, acct_name )
define acct_rec record like acct_totals.*
define acct_name like accounts.acct_name
define col1, col2, col3, col4  smallint
define header1, header2, header3 char(80)

output
```

```
 left margin 0

format

first page header
   let header1 = all("=", 80)
   let header2 = center( "Account Totals Report", 80 )
   let header3 = all("=", 80)

   - set column positions
   let col1 = 4
   let col2 = col1 + 15
   let col3 = col2 + 15
   let col4 = col3 + 10

   - print page headers
   print column col1, header1
   print column col1, header2
   print column col1, header3

   - print column headers
   print column col1, "Acct Number",
         column col2, "Month",
         column col3, "Year",
         column (col4 + 8 ), "Total"

page header
   print header1
   print header2
   print header3
   print column col1, "Acct Number",
         column col2, "Month",
         column col3, "Year",
         column (col4 + 8), "Total"
```

보고서의 on every row 섹션에서는 acct_rec 레코드 요소가 보고서의 first page header 섹션에 설정된 정수 변수에 지정된 컬럼 위치에 인쇄됩니다.

acct_rec.acct_num에 대한 before group of 블록에서는 인쇄할 계정의 이름이 출력됩니다. acc_rec.acct_num 레코드 요소에 대한 after group of 블록에서는 방금 처리된 계정의 총계가 인쇄됩니다.

```
on every row
   print  column col1, acct_rec.acct_num using "<<<<<",
          column col2, acct_rec.acct_month using "<<",
          column col3, acct_rec.acct_year  using "<<<<",
          column col4, acct_rec.acct_total using <=
   "###,###,###.##"

before group of acct_rec.acct_num
   skip 1 line
   print column 1, "Account Name: ", acct_name
   skip 1 line

after group of acct_rec.acct_num

   print column col4, "================"
   print column 1, "Account Total ",
         column col4, group sum( acct_rec.acct_total )
                                      using  "###,###,###.##"
   skip 1 line

end report
```

예 9-1 계정 총계 보고서에 대한 보고서 출력

```
==============================================================
                    Account Totals Report
==============================================================
Acct Number         Month            Year           Total

Account Name: Electricity Exp

1                   1                1993           100.00
1                   2                1993           100.00
1                   4                1993           100.00
1                   5                1993           1,100.00
```

```
1                   6                1993           1,011.00
1                   7                1993           553.00
1                   8                1993           133.00
1                   9                1993           1,133.00
1                   10               1993           133.30
1                   11               1993           556.00
1                   12               1993           44.30
==============================================================
Account             Total                           4,963.60

Account Name: Miscellaneous E
11                  1                1993           1,333.30
11                  2                1993           334.00
11                  3                1993           1,334.00
11                  4                1993           133.30
11                  5                1993           33.40
11                  6                1993           333.40
11                  7                1993           33.44
11                  8                1993           300.00
11                  8                1993           300.00
11                  9                1993           500.00
11                  10               1993           600.00
11                  11               1993           234.00
11                  12               1993           560.00
==============================================================
Account             Total                           6,028.84

Account Name: Water
2                   1                1993           400.00
2                   3                1993           500.00
2                   4                1993           600.00
2                   5                1993           550.60
2                   6                1993           1,133.00
2                   7                1993           554.00
2                   8                1993           443.00
2                   9                1993           334.00
2                   10               1993           1,100.10
2                   11               1993           1,193.00
```

```
2                   12              1993          1,110.00
============================================================
Account             Total                         7,917.70
```

테이블 중심의 동적 보고서

보고서 포맷의 가능성은 사실상 제한이 없지만 반복되기 쉬운 경향이 있습니다. 이러한 보고서 포맷 중 하나는 일련의 숫자가 보고서 페이지에 인쇄되는 보고서입니다. 또한 보고서에 인쇄되는 각 숫자는 개별 select문의 결과로, 거의 집계 총액을 검색하는 것입니다. 보고서에 월별 총액을 인쇄하고 각 행에 계정 그룹을 인쇄하는 계정 지출 보고서가 이러한 보고서의 한 예이며 예 9-2에 나와 있습니다. 다른 예로는 보고서의 각 라인이 일련의 계정에 대한 총액을 나타내는 계정 대차 대조표가 있습니다.

이러한 각 보고 요구에 대해 구별된 사용자 정의 보고서를 작성하는 것은 지루한 작업이 될 수 있습니다. 상당한 양의 코드를 재사용할 수 있는 보다 모듈식의 접근법을 선호하게 됩니다. 그림 9-6의 예가 이러한 접근법을 나타냅니다.

이러한 동적 보고서는 보고서 포맷과 보고서의 각 컬럼 및 행에 대해 검색할 데이터를 지정하는 정보를 저장하기 위해 테이블을 필요로 합니다. 이러한 정보는 궁극적으로 보고서를 작동시키는 정보입니다. 런타임 시 이 테이블을 읽습니다. 이 보고서와 일치하는 모든 행이 선택됩니다. 각 행에 대해 쿼리 명령문이 준비되고 실행됩니다. 쿼리 명령문의 실행 결과는 제어 레코드에 들어 있는 다른 포맷팅 정보와 함께 보고서에 출력됩니다. 그림 9-6이 이 프로세스를 보여줍니다.

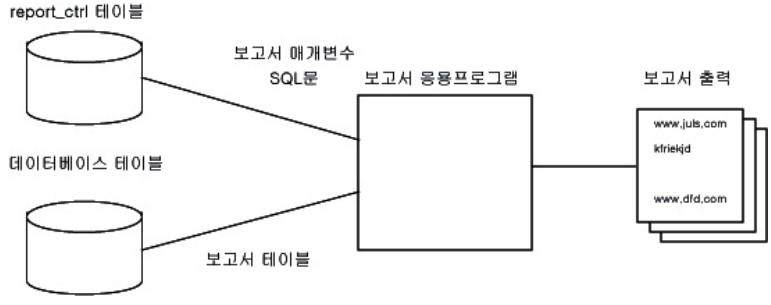


그림 9-6 테이블 중심의 동적 보고서

예 9-2 동적으로 생성된 지출 보고서의 보고서 출력

```
============================================================
3/19/1994          Expense Report                 15:05:01
=====================================================Page: 1
             Jan                  Feb             Mar

Water        100.00               100.00
Phone        400.00               500.00
Labor        600.00               100.00          123.00
Subtotals    1100.00              200.00          623.00
Cleanup      2274.40              1133.00         123.33
Misc.        1333.30              334.00          1334.00
Subtotals    3607.70              1467.00         1457.33
Grand Total  4707.70              1667.00         2080.33
```

이 테이블의 각 행은 보고서 이름, 보고서에서 데이터가 인쇄될 위치에 관한 정보, 데이터를 검색하는 데 사용되는 SQL문을 지정합니다. 표 9-5는 이 보고서를 작동시키는 데 사용되는 테이블의 내용을 나타냅니다.

표 9-5 보고서 제어 테이블

컬럼	용도
rept_name	보고서의 이름
line_no	보고서의 라인 번호. 보고서의 행과 관련이 있습니다.
header	line_no = 0인 경우 보고서의 헤더이거나 보고서 제목입니다. line_no가 0이 아닌 경우 행의 헤더이거나 제목입니다.
col_pos	검색할 데이터가 인쇄될 컬럼 위치
col_no	이 행의 컬럼 번호 또는 행 번호. 이 번호는 이 컬럼의 데이터를 식별하는 데 사용됩니다. 소계와 총계를 저장하고 report_ctrl 테이블에서 검색한 데이터를 정렬하는 데 사용됩니다.
sign	이 행과 컬럼에 있는 데이터의 부호(+/-)를 저장합니다. 인쇄할 이 컬럼의 데이터가 음수이면, 음수 1이 저장되고, 인쇄할 데이터가 양수이면, 양수 1이 저장됩니다.
action	이 행에 대한 1byte 플래그. 수행할 조치를 나타냅니다. 사용 가능한 설정은 아래에 나열되어 있습니다.
q_stmt	이 행과 컬럼에 대한 데이터를 검색하는 데 사용될 쿼리 명령문을 저장합니다.

```
create table report_ctrl
(
    rept_name char(10),
    line_no smallint,
    header char(20),
    col_pos smallint,
    col_no smallint,
    sign smallint,
    action char(1),
    q_stmt char(500)
);
```

main 프로그램 블록과 rpt_drv 함수

main 프로그램 블록은 다음과 같이 보고서의 이름, 보고서의 컬럼 폭 및 보고서가 출력될 파일로 rpt_drv 함수를 간단하게 호출합니다.

rpt_drv 함수는 보고서를 설명하는 report_ctrl 테이블에서 행을 읽은 다음, 각 행의 내용을 이 행을 처리할 함수에 전달합니다. 먼저, 실행되고 있는 보고서 이름과 일치하는 모든 행을 report_ctrl 테이블에서 읽도록 커서를 선언합니다. 커서가 선언되면 보고서가 시작되고 함수에 전달된 파일 이름으로 출력됩니다.

그런 다음, foreach 루프가 실행되고 report_ctrl 테이블의 모든 요소가 report_ctrl 레코드로 검색됩니다. report_ctrl 레코드의 action 요소는 처리되고 있는 행의 유형을 나타내는 플래그입니다. 작업 플래그에 사용 가능한 설정은 표 9-6에 나와 있습니다.

표 9-6 작업 플래그 설정

플래그 설정	용도
N	이 행과 컬럼에 있는 데이터가 숫자 데이터임을 나타냅니다. 이는 데이터를 처리하기 위해 proc_line 함수를 호출해야 함을 의미합니다.
S	이 행과 컬럼이 소계를 트리거하는 데 사용될 것임을 나타냅니다.
G	이 행과 컬럼이 총계를 트리거하는 데 사용될 것임을 나타냅니다.
H	헤더 행임을 나타냅니다. 헤더 컬럼의 내용이 보고서 헤더로 인쇄됩니다.

작업 플래그가 "N"으로 설정된 경우, report_ctrl 레코드는 proc_line 함수로 전달되며 이 함수는 레코드에 포함된 쿼리를 처리하고 금액을 10진수 변수로 리턴합니다. 그런 다음, 금액 변수, 보고서 제어 레코드 및 보고서의 컬럼 폭이 보고서에 출력됩니다. foreach 루프가 완료되면 보고서는 finish report문에 의해 닫힙니다.

```
database testit
main

call rpt_drv( "Accounting", 80, "acctg.out" )

end main

function rpt_drv( rept_name, col_width, file_name )
define rept_name like report_ctrl.rept_name
define col_width smallint,
       file_name char(50)
define rc record like report_ctrl.*
define amt decimal
declare c1 cursor for
  select *
  from  report_ctrl ct
  where  ct.rept_name = rept_name
  order by line_no

start report sql_rpt to file_name

foreach c1 into rc.*

    if rc.action = "N" then
       call proc_line( rc.* ) returning amt
    end if
    output to report sql_rpt( rc.*, amt, col_width )

end foreach
finish report sql_rpt

end function
```

proc_line 함수

proc_line 함수는 보고서 제어 레코드를 매개변수로 사용합니다. 10진수는 리턴 값으로 사용하도록 정의됩니다. 기본 오류 검사 모드는 해제되며 보고서 제어 레코드의 q_stmt 요소에 들어 있는 명령문이 준비됩니다. 오류가 발생하면 실패한 쿼리 명령문과 함께 오류 메시지가 표시됩니다. 그런 다음, 준비된 쿼리 명령문에 대해 커서가 선언됩니다. 커서가 열리고 데이터가 페치되어 호출 함수로 리턴됩니다.

커서에서 추가 행을 읽으려는 시도가 없다는 것을 유의하십시오. 쿼리가 한 행의 숫자 데이터 즉, 일련의 숫자 값에 대한 합계만 검색할 것입니다.

```
function proc_line( rc )
define rc record like report_ctrl.*
define retval decimal

whenever error continue
prepare rpt_stmt from rc.q_stmt
if sqlca.sqlcode < 0 then
  error "Error on prepare.", sqlca.sqlcode
  initialize retval.* to NULL
  return retval.*
end if

declare rpt_curs cursor for rpt_stmt

open rpt_curs

fetch rpt_curs into retval

return retval * rc.sign

end function
```

sql_rpt 보고서

sql_rpt 보고서는 보고서 제어 레코드, 행에 대한 금액의 10진수 값 및 보고서의 컬럼 폭을 매개변수로 받습니다. 보고서의 헤더와 보고서의 현재 라인을 보유하기 위해 변수가 선언됩니다. 이 보고서에서 데이터가 처리되는 방식 때문에 소계와 총계를 보고서 집계(그룹 합계, 그룹 소계 등)로 처리할 수 없습니다. (이는 보고서 집계가 보고서에 전달된 매개변수에 대해서만 수행되고 보고서에 대한 여러 컬럼 출력을 인식하지 않으므로 적용되지 않습니다.) 따라서 보고서의 소계와 총계를 보유하기 위해 각각 50 차원의 두 개의 배열이 선언됩니다. 보고서 내에서 카운터 변수와 플래그로 사용될 변수도 선언됩니다.

보고서의 output 섹션에서 왼쪽 여백을 영(0)으로 설정하여 보고서가 보고서 컬럼의 배치를 완전히 제어할 수 있도록 합니다. first page header 섹션에서 초기화가 수행됩니다. 소계 및 총계 플래그가 false로 설정되며 소계 및 총계 배열이 영(0)으로 초기화됩니다. 보고서의 논리는 이 수행을 요구합니다.

보고서 헤더는 다른 행이 보고서에 출력되기 전에 빌드되어야 하기 때문에, 제어 테이블의 보고서 헤더 행이 첫 번째로 보고서로 전달되도록 하는 방식으로 보고서 제어 데이터를 입력해야 합니다. 이 예에서는 이를 위해 헤더 레코드에 대한 컬럼 번호(report_ctrl 테이블의 col_no 컬럼)를 영(0)으로 설정하였습니다. 보고서 제어 레코드는 올림차순 컬럼 번호로 정렬되고 다른 모든 비헤더 행은 영(0)보다 큰 컬럼 번호를 가지므로 보고서 제어 헤더 레코드가 가장 먼저 보고서로 전달됩니다. first page header 섹션의 코드는 rc 레코드의 action 요소가 “H”로 설정되었는지 검증할 때 특히 더 주의합니다. 사실 보고서에 전달되는 첫 번째 행이 헤더 레코드인 경우에는 이 값을 검사할 필요가 없습니다.

first page header 제어 블록 내에서는 보고서 헤더가 일련의 세 개 문자열에 작성되어 저장됩니다. all 함수를 사용하여 컬럼 총계를 저장하는 데 사용될 문자열을 초기화합니다. 이러한 문자열은 현재 라인 문자열 변수의 서브스트링으로 삽입될 것이기 때문에 명시적으로 지정되지 않은 문자열 부분에 불명확한 값이 남아 있지 않도록 전체 문자열을 공백으로 설정해야 합니다.

보고서의 제목은 보고서 제어 레코드의 header 요소에 들어 있습니다. 또한 컬럼 헤더는 보고서 제어 레코드의 q_stmt 요소에 있습니다. 컬럼 헤더는 colheader 문자열에만 지정됩니다. 날짜, 시간 및 페이지 번호도 보고서 헤더 문자열에 배치됩니다. 모든 헤더 정보는

함수로 전달된 컬럼 폭 매개변수와 관련하여 배치됩니다.

일단 헤더 문자열과 컬럼 헤더 문자열이 작성되면 이 문자열들은 출력됩니다. 세 번째 페이지 헤더 문자열에서 변경되는 정보는 단지 페이지 번호 뿐입니다. 이것은 각 페이지의 header3 문자열에 다시 삽입됩니다.

```
report sql_rpt( rc, amt, col_width )
define rc record like report_ctrl.*
define amt decimal,
      col_width  smallint
define header1, header2, header3, colheader char(132)
define tstr char(50)
define curr_line char(132)
define n, x, colno smallint
define sub_totals array[50] of decimal
define grand_totals array[50] of decimal
define sub_total_flag, grand_total_flag smallint

output
left margin 0
format
first page header

- initialize
let sub_total_flag   = FALSE
let grand_total_flag = FALSE

for n = 1 to 50
  let sub_totals[n]   = 0
  let grand_totals[n] = 0
end for

- these need to be passed in 1st to force initializations
if rc.action = "H"  then - build the header

  let curr_line = all(" ", col_width )
  let colheader = all(" ", col_width )
  let header1  = all("=", col_width )
```

```
  let header2  = all("=", col_width )
  let header2 = center( rc.header, col_width)

  let colheader = rc.q_stmt      this field contains column<=
  headers
  let tstr     = today
  let n        = length(tstr clipped)
  let header2[1,n+1] = tstr clipped
  let tstr = time
  let n = length( tstr clipped)
  let header2[ col_width - n, col_width ] = tstr clipped
  let header3 = all("=", col_width )

end if

  let tstr = pageno using "Page: <<<"
  let n = length( tstr clipped )
  let header3[ col_width - n , col_width ] = tstr

  print header1  clipped
  print header2  clipped
  print header3  clipped
  print colheader clipped
  skip 1 line


Page header

  let tstr = pageno using "Page: <<<"
  let n = length( tstr clipped )
  let header3[ col_width - n , col_width ] = tstr

  print header1 clipped
  print header2 clipped
  print header3 clipped
  print colheader clipped
  skip 1 line
```

on every row 보고서 블록

보고서의 on every row 섹션은 상당한 양의 처리를 수행합니다. 이 보고서에 사용된 보고서 플래그는 숫자 출력의 경우 “N”이고, 소계의 경우 “S”이며, 총계의 경우 “G”입니다. 행에 이러한 플래그가 설정되어 있으면 제공된 금액은 현재 라인(curr_line 변수) 문자열의 올바른 오프셋에 저장됩니다. 현재 구현된 코드는 금액 컬럼에 대해 10자를 허용합니다. 현재 라인 문자열에 금액을 로드하기 위한 오프셋을 계산하여 로컬 변수에 저장합니다. 때때로 서브스트링 참조의 표현식에 문제를 나타내는 Informix-4GL 구문 분석기의 특성으로 인해 이러한 작업이 매우 필요합니다. 따라서 금액은 현재 라인 문자열에 삽입되고 다시 보고서 제어 레코드에 전달된 시작 컬럼 위치에 10byte를 더한 서브스트링에 삽입됩니다.

먼저 보고서 제어 레코드 요소의 조치 플래그 설정에 기초하여 현재 라인 서브스트링에 금액이 저장됩니다. 조치 플래그가 “S”인 경우 처리 중인 컬럼 번호의 현재 라인에 소계를 삽입해야 합니다. 보고서에서 컬럼은 왼쪽에서 오른쪽 방향으로, 첫 번째 컬럼은 1이고 최대 컬럼은 50이 되는 방식으로 번호가 지정됩니다. 컬럼 번호는 소계 및 총계 배열에 대한 배열 인덱스로 사용되며, 이들 배열은 50 요소 배열로 정의되기 때문에 1과 50 사이의 요소 값만 승인합니다.

action 레코드 요소가 “S”, “G” 또는 “N”으로 설정된 경우 적합한 배열의 금액 값이 현재 라인 문자열에서 해당 컬럼 위치(보고서 제어 col_pos 레코드 요소) 오프셋에 배치됩니다. 일련의 if/then 명령문을 사용하여 사용할 변수를 결정할 수 있습니다. 조치 플래그가 “S” 또는 “G”인 경우 소계 플래그나 총계 플래그는 TRUE로 설정됩니다. 이 플래그는 프로그램에서 나중에 다른 조치를 트리거하는 데 사용됩니다. 행 헤더도 첫 번째 컬럼 위치부터 시작하여 보고서 제어 레코드의 헤더 문자열 요소 길이만큼 확장하여 서브스트링의 현재 라인에 로드됩니다.

그런 다음, 금액 값을 테스트합니다. 금액이 NULL인 경우 영(0)으로 설정되어 프로그램에서 나중에 올바른 수학적 처리를 할 수 있도록 합니다. 소계 플래그를 평가하여 소계 플래그를 지워야 하는지 여부를 결정합니다. 소계 플래그가 설정되고 처리 중인 현재 행이 소계 행이 아닌 경우(rc.actions !=S) 소계를 지울 수 있습니다. (총계는 한 번만 인쇄되기 때문에 지울 이유가 없습니다.) 그런 다음, 소계 플래그는 FALSE로 설정되어 이 코드를 불필요하게 계속 반복 실행하도록 합니다.

현재 행의 조치 플래그가 "N"으로 설정된 경우 이 행은 숫자 라인이므로 해당 컬럼에 대한 소계 계산과 총계 계산에 추가해야 합니다.

현재 라인이 보고서 제어 테이블의 일련의 행들로 구성되어 있기 때문에 각 행에는 동일한 보고서 라인 번호가 있습니다. 보고서 제어 테이블의 각 행은 보고서로 전달될 때 출력되지 말아야 합니다. 현재 라인(현재 라인 문자열에 포함된)이 보고서 출력 파일로 출력되어야 할 때는 단지 현재 보고서 라인 번호의 보고서 제어 행이 모두 보고서로 전달되었을 때뿐입니다. 라인이 출력된 다음 현재 라인 문자열은 다시 초기화됩니다. 보고서의 마지막 행에서 총계 플래그가 설정되면 총계가 보고서 파일로 출력됩니다.

```
on every row

if rc.action = "N" or rc.action = "S" or rc.action = "G" then
   let n = rc.col_pos
   let x = n + 10
   let colno = rc.col_no

   - load the amount
   if rc.action = "S" then  - subtotal
   let curr_line[n,x] = sub_totals[colno]
   let sub_total_flag = TRUE
end if

if rc.action = "G" then – grand total
   let curr_line[n,x] = grand_totals[colno]
   let grand_total_flag = TRUE
end if

if rc.action = "N" then
   let curr_line[n, x] = amt
end if

   - load the row header
   let n = 1
   let x = n + length( rc.header clipped )
   let curr_line[n,x] = rc.header clipped
```

```
   - process the amount or subtotal
   if rc.action = "N" then
      if amt IS NULL then
         let amt = 0
      end if
   end if

end if

   - clear subtotals if necessary
if sub_total_flag and rc.action != "S"  then – clear the <=
        subtotals

       for n = 1 to 50
           let sub_totals[n] = 0
       end for

       skip 1 line

       let sub_total_flag = FALSE

end if

   if rc.action = "N"  then
      let sub_totals[colno]   = sub_totals[colno] + amt  - <=
       accumulate the total
      let grand_totals[colno] = grand_totals[colno] + amt - <=
       accumulate the grand total
   end if

   after group of rc.line_no
      print curr_line clipped
      let  curr_line = all( " ", col_width)

   on last row
   if grand_total_flag then
      skip 1 line
   end if
   print curr_line clipped

end report
```

요약

이 장에서는 Informix-4GL 언어를 사용하여 입력과 출력을 수행하는 프로세스에 대해 설명했습니다. 화면 폼을 사용하여 사용자 인터페이스를 설계하고 제공하는 프로세스를, 데이터를 검색하고 표시하기 위해 각각 화면 폼과 상호 작용하는 `input` 및 `display` 명령문의 사용과 연결하여 설명하였습니다.

Informix-4GL 보고서 구문의 기능도 제공하였습니다. Informix-4GL 보고서 블록을 사용하면 보고서를 빠르고 쉽게 개발할 수 있습니다. 프로시저문은 SQL의 비프로시저문 및 Informix-4GL 보고서 블록과 쉽게 혼합될 수 있으므로 이 언어를 사용하여 복잡한 보고서를 작성할 수 있습니다.

다음 장 미리보기

다음 장에서는 Informix-4GL의 세 가지 구성 요소인 함수, 오류 처리 및 도움말 기능의 사용에 대해 보다 자세한 내용을 설명합니다. 함수의 적절한 사용은 Informix-4GL과 같은 언어를 사용하는 모듈식 프로그래밍의 중요한 부분입니다. Informix-4GL은 여러 입력 매개변수, 여러 리턴 매개변수 및 반복을 허용하는 함수 호출의 유연한 기능을 제공하며, 이는 모두 다음 장에서 예시됩니다.

Informix-4GL은 IDS 오류를 트랩하여 의미 있는 오류 메시지(오류 번호만이 아님)를 표시하는 기능을 제공합니다. 다음 장에서는 이 오류 기능을 사용하는 프로세스를 Informix-4GL 도움말 화면 구문의 사용에 대한 설명과 함께 제공합니다.

10장

Informix-4GL의 함수 및 오류 처리

Informix-4GL의 가장 중요한 기능 중 하나는 함수 생성 기능입니다. Informix-4GL은 PASCAL, ALGOL, C와 같은 제 3 세대 언어의 전통적인 모듈 및 블록 지향 방식으로 함수를 생성할 수 있도록 합니다.

4GL 함수는 function function_name문으로 지정된 시작 블록과 end function문으로 지정된 종료 블록을 가집니다. 이 블록 내에서 함수의 변수 선언 섹션에 선언된 모든 변수는 로컬 범위를 가집니다. 이들 변수는 해당 함수의 블록 범위 내에서만 인식됩니다. (로컬 변수의 이름 선언은 모듈 또는 전역 범위로 동일한 이름으로 선언된 모든 변수의 선언을 마스크한다는 점을 유의하십시오.)

Informix-4GL에서 이러한 변수 이름 범위의 처리 방식은 C 또는 PASCAL의 처리 방식을 그대로 반영하는 것이며 매우 유용한 언어 설계 기능입니다. 프로그래머들은 프로그램의 다른 곳에서 사용 중인 다른 배열 인덱스 변수의 변경을 염려하지 않고 자유롭게 모든 배열 인덱스 변수의 이름을 지정할 수 있습니다. 뿐만 아니라, 로컬 변수에 대한 영역은 함수로 들어갈 때 할당되고 함수에서 리턴될 때 할당해제되기 때문에 로컬 변수를 사용하면 메모리를 더욱 효율적으로 사용할 수 있습니다.

로컬 변수의 사용은, 변수가 단일 명령문 블록으로 선언된 다음 프로그램 전체에서 사용되는 COBOL과 같은 언어에 더욱 익숙한 프로그래머들에게 쉽게 이해되지 않을 수도 있습니다. 모듈 방식의 프로그래밍과는 대조되는 프로그래밍 스타일이기 때문입니다.

Informix-4GL은 응용프로그램을 강제로 모듈 방식으로 작성되도록 하지 않습니다. 그러나 특히 함수의 로컬 변수를 사용하여 변수 범위를 제한하는 경우 그 이점은 상당히 큽니다. Informix-4GL에서 모듈 방식 프로그래밍의 이점을 과소 평가해서는 안 됩니다.

함수 매개변수

Informix-4GL에서 함수는 배열을 제외한 모든 자료형의 매개변수를 받을 수 있으며, 배열을 제외한 모든 자료형의 매개변수를 리턴할 수 있습니다. 또한 다른 여러 언어들과 달리 Informix-4GL은 여러 개의 매개변수를 리턴할 수 있습니다.

Informix-4GL의 함수는 값에 의한 호출(call-by-value)입니다. 함수로 전달되는 매개변수는 값으로 전달되며 이 값은 함수에 의해 수정될 수 없습니다.

들어오는 매개변수의 자료형은 함수 본문의 변수 선언 섹션 내에서 정의되어야 합니다. 이들 자료형은 함수 호출에 사용되는 매개변수의 자료형과 순서 위치가 일치해야 하지만, 자료형이 필수적인 것은 아닙니다. 그 이유는 Informix-4GL이 가능하면 함수로 전달된 자료형을 해당 매개변수에 대한 적절한 자료형으로 변환하기 때문입니다. 들어오는 함수 매개변수의 자료형을 정의하지 못하면 컴파일러 오류가 발생합니다.

구현

Informix-4GL에서의 함수 구현은 다른 모듈식 언어의 구현과 유사합니다. 함수 매개변수는 왼쪽에서 오른쪽의 순서로 Informix-4GL 스택에 푸시됩니다. 이들이 스택으로 푸시될 때 변수의 자료형이 스택에 기록됩니다. Informix-4GL이 함수를 호출하면 이 함수에는 스택으로 푸시된 매개변수 수가 송신됩니다. 스택은 그림 10-1에 나와 있습니다.

함수는 스택에서 해당 매개변수를 검색하기 전에 먼저 전달된 매개변수의 수를 검색합니다. 이 수가 올바른 매개변수 수가 아닌 경우, 함수는 치명적 오류 신호를 보내고 프로그램이 중단됩니다.

올바른 매개변수 수가 전달되면, 매개변수 수는 스택에 위치된 순서와 반대의 순서로(예를 들어, 오른쪽에서 왼쪽으로) 스택에서 검색됩니다. 매개변수가 스택에서 검색되면 매개변수에 필요한 자료형

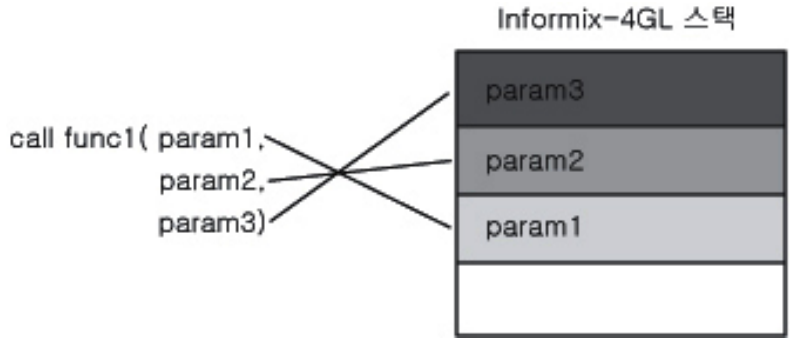


그림 10-1 Informix-4GL 스택

이 검색 중인 매개변수의 자료형과 비교됩니다. 매개변수가 변환 가능하면 이를 변환하고, 그렇지 않으면 치명적 오류를 리턴하고 프로그램 실행을 종료합니다. Informix-4GL은 매개변수의 변수 수를 함수로 전달하는 기능을 명시적으로 지원하지 않습니다. 매개변수 수는 실제로 고정되어 있기 때문입니다.

일단 모든 매개변수가 검색되었으면 함수 본문이 실행됩니다. 함수 선언을 제외한 모든 유효한 Informix-4GL문은 함수 본문 내에 위치될 수 있습니다. 함수 블록의 실행을 완료하면 Informix-4GL은 리턴 매개변수를 스택에 넣은 후, 스택에 있는 매개변수 수를 리턴하거나 송신합니다. 리턴 매개변수는 함수 매개변수와 같은 방식으로 처리됩니다. 즉, 이들을 반대 순서로 검색하여 해당되는 자료형 변환을 수행합니다. 함수에서 리턴될 때 로컬 변수의 스토리지에 사용되었던 모든 공간이 할당해제됩니다. 변수는 더 이상 존재하지 않습니다.

모든 매개변수 유형 변환과 매개변수 수의 테스트는 런타임 때까지 수행되지 않습니다. Informix-4GL 컴파일러는 함수 호출의 유효성을 검사하지 않습니다.

표현식으로서의 함수

Informix-4GL에서 함수 호출은 표현식으로도 처리될 수 있습니다. 유효한 명령문은 다음과 같습니다.

```
-the function isvalid_state() returns integer 0 for
-false and 1 for true
```

```
let x = isvalid_state("CA")

define x integer
if x then
     call do_something()
end if
```

위의 코드에서, 지정 명령문 let x = isvalid_state("CA")는 isvalid_state의 리턴 값을 변수 x에 자동으로 지정합니다. 이는 isvalid_state 함수가 단지 한 개의 매개변수를 리턴한다고 가정합니다. (그렇지 않으면 런타임 오류가 발생합니다.)

Informix-4GL 부울 표현식 또한 함수를 표현식으로 허용합니다. 유효한 코드는 다음과 같습니다.

```
if isvalid_state("CA") then
   call do_something()
end if
```

이 기능을 사용하면 유효성 확인에 필요한 코드 라인 수를 줄이고 모듈 방식의 읽을 수 있는 코드를 더 많이 작성할 수 있다는 것에 유의하십시오. 더욱 복잡한 중첩 함수 표현식도 다음과 같이 작성할 수 있습니다.

```
let state = "ca"
let country = "us"

if isvalid_state(upshift (state)) and
        isvalid_country(upshift (country)) and
        isvalid_zip("08833") then
        call do_something()
end if
```

함수 내에서 변수 사용

앞서 언급한 바와 같이, 함수 내에 선언된 Informix-4GL 변수는 로컬 범위를 가집니다. 이들 변수는 해당 함수 내에서만 볼 수 있습니다. 이들 변수의 공간은 함수로 들어갈 때 할당되고 함수에서 나갈

때 할당해제됩니다. 이는 연속적인 함수 호출 사이에서 변수가 값을 보존하지 않는다는 것을 의미합니다. 대부분의 경우, 이것은 문제가 되지 않습니다. 그러나 함수가 이전 호출의 정보를 보존해야 하는 경우, 해당 값을 보존할 수 있는 변수가 필요합니다. 모듈식 코딩에서는 전역 변수의 사용을 권장하지 않기 때문에 전역 변수를 사용하지 않는 것이 좋습니다.

솔루션은 모듈 범위로 변수를 선언하는 것입니다. 이는 변수가 선언된 Informix-4GL 소스 코드 모듈(소스 코드 파일)에만 보여지는 변수입니다. 이들 변수는 다음과 같이 함수 블록 외부 및 전역 명령문 블록 외부에서 간단하게 선언됩니다.

```
-
- declare module scope variables
-
define first_pass_flag smallint
define pass_counter  integer

-
- function declarations start here
-

function get_data(rec_key)
#
# Purpose: to retrieve data from the tab1 table
#
define rec_key  like tab1.key_col
define rec_var  record like tab1.*
- assert first_pass variable is initialized to TRUE in main
- We only want to execute the following code once, on the
- first pass through the function
-
if first_pass then

             declare c1 cursor for
                     select *
                     from tab1
                     where key_col = rec_key
```

```
                    open c1

                    let first_pass = false

end if

fetch c1 into rec_var.*

-
- assert pass_counter was initialized to 0 in main
let pass_counter = pass_counter + 1
call display_counter( pass_counter )

return rec_var.*

end function
```

위의 예에서는 두 개의 모듈 범위 변수인 first_pass와 pass_counter 변수의 선언을 통해 함수 get_data가 이전에 호출되었는지 여부를 기억하도록 할 수 있습니다. 이전에 호출되었으면 커서를 선언하고 커서를 열어야 하는(비용이 많이 드는 데이터베이스 작업임) 오버헤드를 방지할 것입니다. 또한 전달 계수를 변수 pass_counter에 보존하며, 이 변수는 지금까지 리턴된 레코드 수의 계수를 표시할 함수로 전달됩니다. first_pass와 pass_counter는 해당 소스 코드 모듈의 다른 함수에는 표시되고 수정될 수 있지만, 다른 소스 코드 모듈에서는 표시되거나 수정될 수 없습니다.

매개변수의 변수 수를 함수로 전달

Informix-4GL은 매개변수의 변수 수가 함수로 전달되도록 명시적으로 허용하지 않습니다. 그러나 Informix-4GL의 몇 가지 간단한 문자열 조작 기능을 사용함으로써 Informix-4GL 함수에 매개변수의 변수 수를 전달하고 리턴받을 수 있습니다.

이 기능의 핵심은 구분된 함수 매개변수를 문자열에 넣어 이 문자열을 함수 매개변수로 전달하는 것입니다. 이 방법은 다음과 같습니다.

```
let table_list = "table1, table2, table3"
let column_list = "col1, col2, col3"
```

```
call get_data (table_list, column_list)
```

함수는 구분된 문자열을 가져와 매개변수를 분석하여 각 매개변수를 변수에 넣습니다. 함수는 이들 변수를 함수 매개변수로 사용합니다. 이를 수행하는 데 필요한 문자열 조작은 다음과 같습니다.

```
function get_data( tables, cols )
define tables char(100)
define cols   char(100)
define x,y    smallint
define len    smallint
define table_parms array[10] of char(20)
define col_parms   array[10] of char(20)

-
- initialize the parameter strings.
-
for n = 1 to 10
   for y = 1 to 20
         let table_parms[n] [y] = " " — a blank character
   end for
end for
-
- move through tables string, getting parameters
-
let len = length( tables )
let n = 1
let y = 1

for x = 1 to len
  -
  - 'copy' to parameter array
  -

  let table_parms[y] [n] = tables[x]
  let n = n + 1

  -
```

```
  - test to see if we are on a delimiter. If so,
  - jump to the next parameter
  -
  if tables[x] = ","  then – terminate the parameter

  let x = x + 1       – move past the delimiter
  let y = y + 1       – next parameter
  let n = 1           – start at the beginning
  end if

end for
-
- do the same for the columns list passed
- into the function
-
...
```

이 예에서 입력 매개변수는 for 루프 내에서 분석되는데, 이 루프는 모든 매개변수를 안전 한계(내장 length 함수에 의해 제공됨)까지 전달하여 이들을 문자 변수의 배열에 저장하도록 합니다. 단, 문자 배열을 사용하려면 먼저 안전 값 세트로 초기화해야 합니다. 이는 두 가지 이유 때문에 필요합니다. 첫째, 앞에서 언급한 대로 Informix-4GL에서 선언된 변수는 항상 결정되지 않은 값을 가집니다. (p-코드 버전 실행 프로그램은 이를 다른 방법으로 보이도록 만들지만, 프로그래머는 항상 값이 결정되지 않은 것처럼 코드합니다.) 둘째, Informix-4GL에서 문자열은 항상 정의된 길이만큼 채워진다고 가정합니다. 전체 문자열을 참조하고 있다면, 전체 문자열을 참조하는 Informix-4GL 지정 명령문은 문자열을 공백으로 채웁니다. 그러나 문자열의 각 요소가 지정되는 경우, 문자열은 공백으로 채워지지 않습니다.

이러한 변수를 초기화하는 데는 보통 initialize문이 사용되지만, 문자 변수를 초기화하는 경우 간단히 첫 번째 문자를 NULL 문자로 설정합니다. (이 동작은 20byte 문자열의 20byte를 수정해야 하는 대신 단지 한 개의 바이트만 수정하면 되므로 성능을 향상시킵니다.)

이 책에 나와 있는 코드 예에서는, 지정 명령문을 사용하여 문자열의 요소를 직접 지정하고 있기 때문에 이 동작은 작동하지 않을 것입니다. 문자열의 모든 요소를 공백으로 초기화하지 않을 경우,

남은 문자들은 명시적으로 지정한 마지막 바이트 뒤에 있는 문자열 끝에 따라 들어올 것입니다.

예에서처럼 일단 문자열 배열을 초기화한 다음에는 for 루프를 사용하여 함수로 전달된 문자열에서 매개변수를 분석합니다. 그러면 인덱스 지원에 사용된 여러 정수가 적절한 값으로 초기화됩니다.

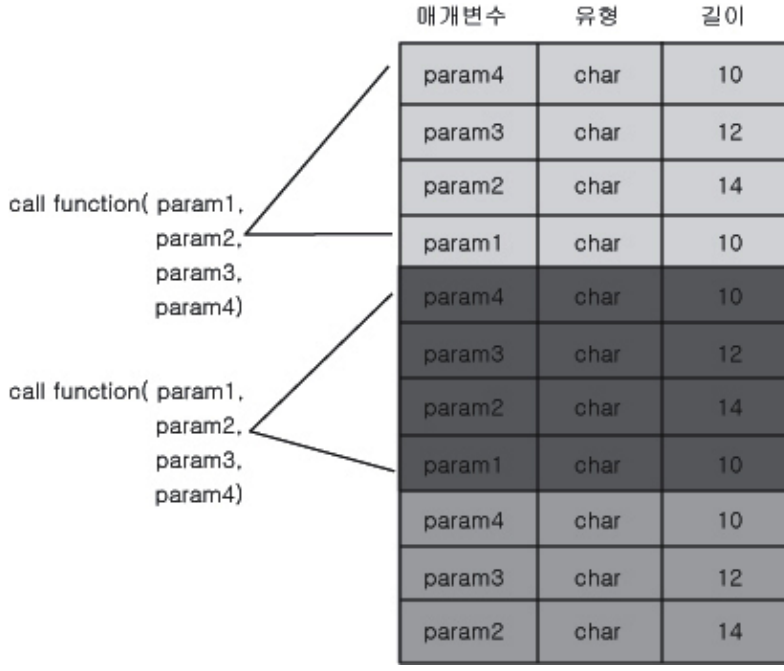


그림 10-2 Informix-4GL 스택

그런 다음, 매개변수가 들어 있는 문자열의 전체 길이에 대하여 for 루프를 시작합니다. for 루프가 반복될 때마다 매개변수가 들어 있는 문자열의 내용이 문자열 매개변수의 배열로 바이트 단위로 복사됩니다. 구분 문자가 있으면(이 경우에는 콤마임) 현재의 매개변수 배열 요소를 추적하는 인덱스 변수가 점증되고 재설정됩니다. 그러면 for 루프가 끝날 때 tables_parms 배열에는 함수로 전달된 모든 테이블 매개변수가 들어가게 됩니다.

이 예에서는 10개가 넘는 매개변수가 함수로 전달되는 인스턴스를 처리하도록 코드되지 않았다는 것을 유의하십시오. 일반적인 유효성 확인 함수를 작성하기 위해서 함수 매개변수를 전달하는 방법의 보다

더 완전한 예에 대해서는 이 장의 뒷부분에서 설명하겠습니다.

Informix-4GL의 반복 기능

Informix-4GL에서는 더욱 강력한 함수 관련 언어 기능인 반복 기능을 사용할 수 있습니다. 반복 기능은 일반적으로 모듈 방식의 제 3 세대 언어에서 필수 요소로 여겨집니다. Informix-4GL은 약간의 제한사항만 가지고 비프로시저 코드와 프로시저 코드 둘 다에서 반복 기능을 제공합니다. Informix-4GL은 허용된 반복 레벨에 대하여 제한을 두지 않습니다. 반복은 연관된 스택 작업을 지원하는데 사용할 수 있는 메모리가 있는 한 계속 수행할 수 있습니다.

반복은 기본적으로 함수 자체 호출을 허용합니다. Informix-4GL이 스택을 사용하며 푸시다운 방식으로 인수를 스택에 넣으므로, 함수 매개변수는 반복 때문에 혼란스럽게 되지 않습니다. 단, 반복 기능에는 비용이 듭니다. 함수 호출은 스택 작업을 필요로 하고 스택 작업은 자원 사용 면에서 비용이 많이 들 수 있습니다. 비록 반복 솔루션을 사용할 수 있다 하더라도 종종 반복적인 프로그래밍 솔루션을 선택하여 반복의 오버헤드를 방지하기도 합니다.

반복은 반복적 알고리즘으로 표현하기 어려운 문제에 대해 상대적으로 간단하고 명쾌한 솔루션을 제공합니다. 다음 코드는 Informix-4GL에서의 반복 기능의 예입니다.

```
-
-define module scope variable to hold counter value
-
define levels smallint - indicates the depth of recursion

main

let levels = 0
call recurse()

end main

function recurse ()
define col smallint

-
```

```
- increment level counter
-
let levels = levels + 1

-
- stop at a depth of 20
-
if levels = 21 then
   display "Finished run." at 8,10
   sleep 2
   return
end if

-
- display message
-
display levels using "Recurse level <<"
               at 10,10
let col = 9 + levels
display "*" at 12, col

-
- go even deeper
-
call recurse()

end function
```

이 예에서는 호출 간의 값을 보유할 카운터 변수를 제공하기 위해 모듈 범위 변수를 선언했습니다. 호출될 때마다 카운터가 점증되고, 동일한 정보를 표시하는 텍스트 그래프와 함께 반복 레벨 계수가 표시됩니다. 반복이 20 레벨에 도달하면(21번째 반복 시), 프로그램이 종료됩니다.

반복의 예 – 부분 분해 문제

반복의 사용은 난해한 문제로 제한되지 않습니다. 반복 기능은 일부 공통적인 관계형 데이터베이스 문제에 대한 솔루션을 제공합니다.

이러한 문제들 중 가장 주목할 만한 것은 자체 참조 테이블 문제입니다. 실제적인 예는 부분 분해 문제로 나타납니다.

다음에 설명되는 테이블 정의는 자동차 부품 데이터베이스에 관한 것입니다. 테이블에 저장된 부품들은 자체적으로 부품이 되는 구성 요소 부품으로 구성되어 있습니다. 정의에 따르면 자체 참조 레벨은 제한이 없습니다. 반복 기능을 잘 구현하면 단지 사용 가능한 메모리에 의해서만 제한되는 정도까지 반복을 지원하게 됩니다. (이는 Informix-4GL의 경우입니다.)

아래에 표시된 Informix-SQL 스크립트는 부품 데이터베이스를 작성하고, 부품 테이블 및 인덱스를 작성한 후, 데이터를 부품 데이터베이스에 로드합니다.

```
{ create the parts database }
create database partsdb;

{ create the parts table and indexes }
create table parts( pname char(20),
                    pnum integer,
                    component_of_pnum integer );
create index pidx1 on parts( pnum );
create index pidx2 on parts( component_of_pnum );

{ load the data for the parts table }
insert into parts values( "alternator-p1", 1, 0 );
insert into parts values( "gear-ax1z", 2, 1 );
insert into parts values( "bearing", 3, 1 );
insert into parts values( "stabilizer-rod", 4, 2 );
insert into parts values( "coil-assembly", 5, 1 );
insert into parts values( "wire-assembly", 6, 5 );
insert into parts values( "wire-frame", 7, 5 );
insert into parts values( "coil-assembly-gasket", 8, 5 );
insert into parts values( "clips-a2", 9, 6 );
insert into parts values( "washer-w2", 10, 5 );
insert into parts values( "clip-w2", 11, 7 );
```

다음은 부분 분해 문제에 대한 Informix-4GL의 프로그램적인 솔루션입니다. 이 코드는 이전 데이터를 사용하고 있으며, 3 레벨까지 반복하여 모든 자동차 부품과 해당 구성 요소 부품, 그리고 이들 구성

요소 부품을 이루는 구성 요소 부품의 목록을 제공합니다.

```
database partsdb

-
- define module scope variables to use as program
- counters and program constants
-
define levels   smallint
define MAX_LEVELS   smallint

main
define pr record like parts.*

declare c1 cursor for
select  *
from   parts

-


- Initialization
-
let MAX_LEVELS = 5
let levels     = 0

foreach c1 into pr.*

   display pr.pnum using " Part number <<<< Name: ",
                           pr.pname

   call get_parts(pr.pnum)

end foreach

end main
```

```
function create_cursor( plevel, lpnum )
-
- DECLARE and OPEN the cursor needed for
- this level number
- hardcoded limit of five levels.
-
define plevel smallint
define lpnum like parts.pnum
define qstr char(100)

let qstr = " select * from parts where ",
            " parts.component_of_pnum = ?"

case plevel
    when 1 prepare stmt1 from qstr
           declare cl1 cursor for stmt1
           open cl1 using lpnum
    when 2 prepare stmt2 from qstr
           declare cl2 cursor for stmt2
           open cl2 using lpnum
    when 3 prepare stmt3 from qstr
           declare cl3 cursor for stmt3
           open cl3 using lpnum
    when 4 prepare stmt4 from qstr
           declare cl4 cursor for stmt4
           open cl4 using lpnum
    when 5 prepare stmt5 from qstr
           declare cl5 cursor for stmt5
           open cl5 using lpnum

end case

end function

_____________________
function fetch_cursor( plevel )
define plevel smallint
define pr record like parts.*
```

```
          case plevel
             when 1
                  fetch cl1 into pr.*
             when 2
                  fetch cl2 into pr.*
             when 3
                  fetch cl3 into pr.*
             when 4
                  fetch cl4 into pr.*
             when 5
                  fetch cl5 into pr.*
          end case

return pr.*

end function
```

```
function get_parts( lpnum )
-
- recursively retrieve component parts
-
define lpnum like parts. pnum    – part master
define cpr record like parts. *    - components of master

let levels = levels + 1

if levels > MAX_LEVELS then
                display "Too many levels"
                exit program
end if
call create_cursor( levels, lpnum )
call fetch_cursor( levels ) returning cpr.*

-
- Continue to loop as long as data is found
- at this level
-
while cpr.pnum IS NOT NULL and sqlca.sqlcode = 0
```

```
display cpr.pnum using
        "     Component part number: <<<< Name:" ,
          cpr.pname clipped, lpnum using
                   " Component of <<<<",
                   levels using " Recurse level: <<"
          -
          - Now get its component parts, if any
          -
          call get_parts( cpr.pnum )

          call fetch_cursor( levels ) returning cpr.*
end while
let levels = levels - 1  - going back one level

end function
```

이 프로그램은 커서를 선언하면서 시작하여 parts 테이블에서 모든 행을 읽습니다. 그런 다음, foreach문에 의해 커서가 내재적으로 열리며, 이 foreach문이 행을 더 이상 발견할 수 없을 때까지 커서로부터 계속 데이터를 읽습니다.

읽은 각 행에 대해 get_parts 함수가 호출됩니다. get_parts 함수는 이 부품에 대해 구성 요소 부품이 존재하는지 여부를 결정하고, 존재하는 경우 해당 구성 요소 부품에 대한 정보를 표시합니다.

Informix-4GL은 반복 기능을 완전하게 지원하지만 생성된 커서는 그렇지 않습니다. Informix-4GL에서 생성된 커서는 로컬 범위가 아니라 모듈 범위를 가집니다. 해당 데이터 영역은 힙에서 할당되고 연속적인 함수 호출 사이의 값을 보존합니다. (이는 계획된 설계로, 커서는 열려 있는 한 해당 값을 보존할 것입니다.)

이러한 동작은 반복적인 함수에서 문제를 일으킵니다. get_parts 함수에서 하나의 커서가 열리고 foreach 루프에서 get_parts가 반복적으로 호출되는 경우, get_parts 함수는 한 레벨 위의 get_parts 함수로 리턴하기 전에 foreach 루프를 종료합니다. get_parts 함수로 리턴할 때, 프로그램은 foreach 루프 내에서 실행을 시작하지만 foreach 루프를 제어하는 커서는 get_parts로의 이전 반복 호출에 의해 닫히게 됩니다. 따라서 더 이상 열리지 않습니다.

이 문제에 대한 솔루션은 잠재적으로 사용될 일련의 커서를 생성하도록 코드를 작성하는 것입니다. 이는 부품 번호 매개변수와 레벨 매개변수가 전달된 create_cursor 함수에서 수행됩니다. 반복 레벨

을 기준으로 하여 부품 번호에 대한 모든 구성 요소 부품을 선택하는 커서를 생성합니다. 이 커서는 또한 이 함수 내에서 열립니다.

foreach 루프 대신, 구성 요소 부품의 행을 검색하기 위한 제어 구조로서 while 루프를 사용합니다. 반복 레벨을 매개변수로 사용하여 fetch_rows 함수를 호출합니다. 반복 레벨은 커서를 생성할 때 인덱스 값으로 사용되기 때문에, 동일 커서에서 행을 페치하기 위해 이를 사용할 수도 있습니다.

각 행을 검색하면, 부품 번호, 부품 이름, 마스터 부품 번호(이 부품이 구성 요소가 되는 부품 번호) 및 반복 레벨이 표시됩니다. while 루프가 종료되기 전에 fetch_cursor 함수를 다시 호출하여 추가 행을 검색합니다. 행을 더 이상 찾을 수 없으면 while 루프가 종료됩니다. while 루프에서 종료 시, 프로그램 실행이 이전 레벨로 리턴하므로 levels 변수는 점차 감소됩니다.

이러한 부분 분해 문제의 구현은 레벨 5의 반복으로 제한됩니다. 그러나 이 코드의 모듈적 특성 때문에 create_cursor와 fetch_cursor 함수에 레벨을 쉽게 추가하고 이에 따라 MAX_LEVELS 값을 조정할 수 있습니다.

Informix-4GL의 오류 처리 및 도움말 메시지

Informix-4GL은 오류 메시지 및 도움말 화면을 표시하기 위한 몇 가지 다양한 방법을 제공합니다. 일부 기능의 목적은 응용프로그램 개발의 디버깅 단계 동안 프로그래머에게 도움을 주기 위한 것이며, 또 다른 기능들은 런타임 시 사용자에게 유용하고 정보를 제공하는 오류 메시지를 제공하기 위한 것입니다.

Informix-4GL은 SQL 오류, Informix-4GL 언어 구문 오류 및 화면 폼 오류의 세 가지 유형의 오류 메시지를 제공합니다. Informix-4GL 언어 구문 오류는 컴파일러에 의해 트랩됩니다. SQL prepare 문을 사용하여 준비된 SQL문외의 SQL 구문에 관한 오류는 컴파일 시점에서 검사됩니다. 이들 명령문은 컴파일러에 표시되지 보이지 않습니다. 이들 명령문에 대한 구문은 런타임 때까지 검사되지 않다가 Informix-4GL의 프론트엔드가 아니라 엔진에 의해 검사됩니다.

폼 오류는 다양한 표현식에서 발생하는 런타임 오류뿐 아니라 화면 폼 상호 작용 중에 발생하는 오류도 포함합니다. 여기에는 영(0)으로 나누기, 10진수 오버플로 및 날짜/날짜 시간 작업과 같은 오류도 포함됩니다. 폼 오류는 간단한 whenever error문에 의해 트랩되지

않으므로 whenever any error문을 사용해야 합니다.

whenever error 명령문

Informix-4GL은 매우 훌륭한 기본 오류 보고 기능을 제공합니다. 이 오류 보고서는 발생된 SQL 오류 번호, 발생했다면 오류가 발생한 소스 코드 모듈 및 라인 번호, 그리고 오류에 대한 간단한 설명을 포함합니다. 기본적으로 Informix-4GL 런타임 구성 요소는 이 오류 메시지를 표시한 후, 프로그램을 종료합니다(그림 10-3 참조). 이 오류 트랩 방법은 생산 환경에서 잘 사용되지 않을 수 있지만 디버깅 프로세스에서는 매우 유용합니다.

```
Program stopped at "gen_main/4gl", line number 43.
SQL statement error number -529.
Cannot attach to shared memory.
SYSTEM error number -123.
ISAM error: no shared memory.
$
$ 
```

그림 10-3 기본 Informix-4GL 오류 보고서

Informix-4GL 언어는 기본 오류 검사를 무시하기 위해 whenever error문을 제공하여 오류를 트랩하고 이들 오류를 근거로 하여 여러 이벤트를 트리거합니다.

이 메시지의 구문은 다음과 같습니다.

```
whenever | an | error | stop | continue | goto label | call <=
function
```

whenever error문은 모듈 범위를 가집니다. 이것은 그 영향이 컴파일러에 의해 제어되기 때문에 컴파일러 지시로 여겨질 수 있습니다. 이 명령문이 발생하면, 컴파일러는 소스 코드 모듈에서 whenever error문이 발생한 지점으로부터 선택된 오류 처리 방식을 연속적으로 실행하기 위하여 적절한 코드를 생성하기 시작합니다. 따라서 wh

enever error문의 영향은 이 명령문이 발생한 지점부터 x문의 다음 발생 또는 소스 코드 파일의 끝까지입니다(그림 10-4 참조).

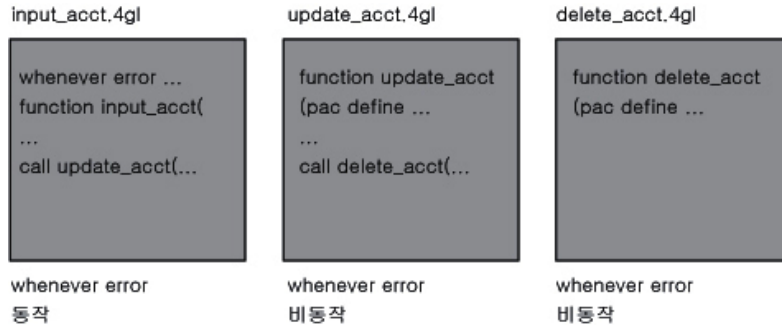


그림 10-4 ***whenever error문의 범위***

whenever error continue 방식의 오류 트래핑은 명령문이 의미하는 것처럼 오류가 발생될 때까지 계속됩니다. 이러한 오류 처리 방법을 사용하는 경우, 오류가 발생하는 라인 다음에서 전역 상태 변수가 발생된 오류의 오류 번호로 설정됩니다. 프로그래머는 이 변수를 검토하고 적절히 대응해야 합니다. 일반적으로 명령문 다음의 상태 변수를 계속 검사할 필요가 있으므로, whenever error continue문의 범위를 제한하고 생산 응용프로그램에서 몇 가지 다른 일관된 방법의 오류 트래핑을 제공할 것을 권장합니다. 다음의 샘플 코드는 whenever error continue문의 일반적인 사용을 보여줍니다.

```
function get_data( rec1 )
define rec1 like table1.*

whenever error continue
select *
into rec1.*
from table1
where table1.col1 = rec1.col1

whenever error stop

if sqlca.sqlcode < 0 then
     error "Error", sqlca.sqlcode, " on select"
```

```
    initialize rec1.* to NULL
end if

return rec1.*

end function
```

이 예에서 데이터베이스에서 데이터를 검색하기 위하여 함수가 호출됩니다. whenever error continue문은 SQL문이 실행되기 전에 실행되도록 코드되었습니다. SQL문 바로 다음에 whenever error stop문을 사용하여 기본 오류 검사를 재시작합니다. SQL문을 실행하면 SQL문이 성공했는지 여부를 판별하기 위하여 sqlca.sqlcode를 검토합니다. 성공하지 않은 경우, 오류 명령문을 사용하여 터미널 화면에 오류 메시지를 표시하고, 데이터를 검색하지 않았음을 호출 함수에 나타내기 위하여 리턴 코드를 NULL로 설정합니다. 함수의 마지막 라인은 이 레코드를 호출 함수로 리턴합니다.

Informix-4GL을 사용하여 오류 로깅

Informix-4GL은 오류 메시지를 ASCII 파일로 로그되도록 하는 함수 세트를 제공합니다. 이 오류 로깅 방법은 생산 응용프로그램의 실행 중 발생할 수 있는 오류를 저장할 수 있는 장점이 있습니다. 일반 사용자는 터미널 화면에 표시된 오류 메시지를 인식하지 못하거나 잘못 읽을 가능성이 높기 때문에 오류 로깅을 통해 오류 메시지를 저장하면 기술 지원 담당자가 나중에 이를 검색하고 검토할 수 있습니다. 또한 유용한 디버깅 도구를 제공하여 오류 로깅이 발생된 오류의 실행 기록 목록을 보존합니다.

오류 로그는 startlog 함수를 사용하여 시작합니다. 일단 시작되면 프로그램 중에 발생하는 연속적인 오류는 이 파일에 엔트리를 생성합니다. 엔트리에는 날짜, 시간, 소스 모듈 이름, 라인 번호, 오류 번호, 그리고 오류 메시지가 포함되어 있습니다. 오류는 항상 오류 로그 파일에 첨부되며 프로그래머는 때때로 이 파일을 정리해야 합니다.

또한 errorlog 함수를 사용하여 엔트리를 오류 로그 파일에 구체적으로 작성할 수 있습니다. 다음 코드는 Informix-4GL 오류 로깅 기능의 사용을 보여줍니다.

```
main

....

call startlog( "prog1.err" )

call get_data( key1, key2 )
returning rec1.*
. . .

end main
```

```
function get_data( key1, key2)
define rec1 like table1.*

- allow errors to be logged; continue program execution on
error
whenever error continue

select col1, col2
into     rec1.*
from  table1
where col1 = key1 and
      col2 = key2

whenever error stop

- write entry to error log indicating function call that
failed
- display error message to user just telling user the data <=
is not there
if sqlca.sqlcode < 0 then
    errorlog( "The get_data function call failed " )
    error "Data not available."
             - display to the terminal screen
    initialize rec1.* to NULL
end if
```

```
return rec1.*

end function
```

이 프로그램은 현재 디렉토리에 오류 로그를 생성하기 위하여 우선 startlog 함수를 호출합니다. (전체 경로 이름을 startlog 함수로 전달하면 특정 디렉토리에 오류 로그를 생성합니다.) 또한 get_data 함수를 사용하여 데이터베이스에서 데이터를 검색합니다. get_data 함수를 호출하면 우선 기본 오류 검사를 해제하고 오류가 발생해도 프로그램이 계속 수행되도록 지시됩니다. 여기서는 오류 로깅이 이미 설정되어 있으므로 프로그램은 오류 정보를 오류 로그에 작성하고 계속 실행합니다.

이 응용프로그램에서 select문 실행 중 오류가 발생하면, 오류는 오류 로그에 로그됩니다. select문 다음에 sqlca.sqlcode를 평가합니다. 이때 영(0)보다 적으면, 오류가 발생하고 추가 설명 정보를 오류 로그에 작성합니다. 이는 오류 로그를 기술적으로 검토하는 사람에게 추가 정보가 됩니다.

whenever any error를 사용하여 폼 및 표현식 오류

화면 폼 입/출력 오류 및 표현식 오류를 모두 포함하는 폼 오류는 간단한 whenever error문으로 트랩되지 않습니다. whenever any error문에서처럼 any라는 키워드를 사용해야 합니다. 다음 코드는 이 명령문의 사용을 보여줍니다.

```
function fix_data(str1, int1, int2)
define str1 char(10)            - should be a date
define int1, int2 integer       - divide and return the result
define dte date
define result decimal

whenever any error continue

- create the date

let dte = date( str1 clipped )
if status < 0 then
```

```
   error "Error in date conversion. Date string is", str1
   let dte = NULL
end if

- perform the division
let result = int1 / int2
if status < 0 then
   error "Error in division. Operands:" ,int1, " / ", int2
   let result = NULL
end if

whenever any error stop

return dte, result

end function
```

이 예에서 fix_data 함수는 문자열을 날짜로 변환하고 두 개의 정수를 나누어 그 결과를 리턴해야 합니다. whenever any error 트래핑은 표현식에서의 오류에 대해 편리한 트래핑 방법을 제공합니다. 이 경우, whenever any error문은 continue절을 사용하여 프로그램이 표현식 오류에 대해서도 계속 실행되도록 하고 표현식 다음에 오는 코드가 오류를 검사하여 적절히 응답하도록 합니다.

날짜 변환에 대한 문자열의 경우, date 함수에 대한 함수 호출의 폼으로 표현식을 평가합니다. 함수 호출 후, status 변수를 음수 값에 대해 검사합니다. 값이 음수인 경우, 날짜 변환 오류로 추정하고 날짜 변환에서 오류가 발생했음을 나타내면서 위반된 날짜 문자열을 표시하는 오류 메시지가 표시됩니다. 그런 다음, date 변수는 NULL로 설정됩니다.

수학적 표현식으로 두 개의 정수를 나누고 그 결과를 결과 변수에 저장합니다. 이 표현식에서 가능한 오류는 영(0)으로 나누기와 같은 표현식 오류입니다. 표현식 오류는 나누기 연산 이후에 상태 코드를 검토함으로써 트랩됩니다. 예를 들어 영(0)보다 적은 경우, 나누기 오류가 발생한 것으로 추정하고 위반된 연산자와 함께 오류 메시지를 표시합니다. 그리고 결과 변수는 NULL로 설정됩니다.

이들 예에서는 특정 오류 코드를 검사하지 않았다는 것을 유의하십시오. 오류가 발생했으면, 오류는 상태 코드를 검토하는 위치보다 위에 있는 라인에서 발생했음이 확실합니다. 이는 이전 예에서도 매우 확실한 추정일 수 있지만 항상 그런 것은 아닙니다. whenever

any error문을 표현식 오류가 발생할 수 있는 위치와 가깝게 배치하면, 잘못된 오류가 트랩될 수 있는 가능성이 적어질 수 있습니다.

특정 오류 번호가 트랩되는 경우, 몇 가지 오류 번호가 있는 case 문을 코드에 넣어야 합니다. 많은 경우, 구체적으로 어떤 오류 번호를 사용할 것인지를 지정하기 어려울 수 있습니다. Informix 문서는 실패한 라이브러리 함수 호출 또는 표현식이 어떤 오류 번호를 리턴하는지를 항상 지정하지 않습니다. 그러므로 앞에서 설명한 오류 트래핑 방법이 가장 사용하기 쉬운 방법입니다.

오류 메시지

오류 메시지는 여러 방법을 사용하여 표시할 수 있습니다. 가장 일반적인 것은 error문을 사용하는 것입니다. error문은 단일 문자열 인수를 가져와 터미널 벨을 울리고, 오류 라인(options문에 정의됨)상에 이 문자열을 역상으로 표시합니다. error문의 현재 버전은 표현식을 인수로 승인하지 않고 단지 단순 자료형으로 승인합니다.

오류 메시지를 표시하는 데 사용할 수 있는 오류 처리와 관련하여 error문뿐만 아니라 여러 함수가 제공됩니다. 표 10-1은 이러한 함수의 목록입니다.

표 10-1 Informix-4GL 오류 처리 함수

함수	목적
err_print	오류 메시지를 오류 라인에 역상으로 인쇄합니다.
err_get	오류 번호를 정수 인수로 가져와 이 오류 번호에 해당하는 문자열 오류 메시지를 리턴합니다.
err_quit	오류 번호를 정수 인수로 가져와 오류 메시지를 오류 라인에 인쇄한 후, 프로그램을 종료합니다.
errorlog	문자열을 인수로 가져와 이를 메시지가 인쇄된 날짜 및 시간과 함께 오류 로그로 인쇄합니다. 이 함수를 사용하기 위해서는 startlog 함수를 미리 호출해야 합니다.
startlog	오류 로그 파일의 이름 및/또는 전체 경로에 해당하는 문자열 인수를 가져옵니다. 전체 경로가 지정되지 않았을 경우, 현재 디렉토리에서 오류 로그를 엽니다. 전체 경로 이름이 지정되었으면 전체 경로에 지정된 디렉토리에서 오류 로그를 엽니다.

일반화된 오류 처리 루틴

모듈 방식의 응용프로그램 개발 환경에서 작업하는 프로그래머들이 일반화된 오류 처리 루틴을 사용할 수 있도록 하는 것은 바람직한 일입니다. 필요한 경우, 응용프로그램 코드는 whenever error continue문을 사용하여 오류 핸들러를 해제한 다음, 다시 whenever error문을 사용하여 오류 핸들러를 설정할 수 있습니다. 다음 코드는 이러한 방법을 보여줍니다.

```
function err_handler()

whenever error call err_handler
. . .

function err_handler()

call errorlog( err_get )
call err_print( sqlca.sqlcode )

end function
```

```
function get_count( key1, key2 )
define key1 like table1.col1
define key2 like table1.col2
define cnt integer

- turn off the default error handler
whenever error continue
select count(*)
into cnt
from table1
where col1 = key1 and
      col2 = key2

if sqlca.sqlcode < 0 then
    let cnt = 0         - rather than null
```

```
    call err_handler()  - now call this routine to display <=
         the error message
end if
whenever any error call err_handler

return cnt

end function
```

이 예에서는 whenever error문을 사용하여 err_handler 함수에 대한 모든 오류를 트랩하고 있습니다. 모든 오류가 오류 로그 파일에 작성되고 오류 메시지는 터미널의 화면에 인쇄됩니다. 이 프로세스는 err_get 함수의 리턴 값을 인수로 갖는 errorlog를 사용하여 수행합니다. err_get 함수는 현재 오류의 오류 번호를 사용하여 호출되므로 현재 오류에 해당하는 오류 메시지를 리턴합니다. 이는 오류 로그에 작성되는 오류 메시지입니다. err_handler 함수의 마지막 라인은 현재 오류의 오류 번호를 사용하여 err_print 함수를 호출합니다. 이 함수는 현재 오류에 대한 오류 메시지를 터미널 화면에 인쇄합니다.

오류 프롬프트 함수

오류 처리 기능의 한 가지 작은 단점은 사용자가 메시지를 읽었는지를 확인하지 못한다는 것입니다. 메시지는 단지 화면에 표시되거나 오류 로그에 작성됩니다. 사용자가 오류 메시지를 읽었는지 확인할 방법은 없습니다.

사용자가 작업을 계속하려면 키를 누르도록 요구하는 대화 상자는 사용자가 적어도 오류 메시지를 보았다는 것을 확인합니다. (대부분의 경우 사용자는 메시지를 적어 두지 않습니다.) 다음 코드는 창을 열고 발생한 오류에 대한 정보를 표시한 후, 프로그램을 계속 수행하기 전에 사용자가 리턴 키를 누르기를 기다리는 함수의 사용을 보여줍니다.

```
function err_msg(msg)
define msg char(80)
define tmpmsg char(60)
define c1 char(1)
```

```
let msg = center(msg, 50)

open window _ew1 at 10, 5 with 4 rows, 50 columns
           attribute(border, prompt line last)

display msg clipped at 2,2 attribute(bold)

prompt "Press return to continue " for char c1

close window _ew1

end function
```

이 함수는 단일 매개변수인 메시지 문자열을 승인합니다. 그런 다음, 창을 열고 굵은체 속성으로 오류 메시지를 표시한 다음, 사용자에게 리턴 키를 눌러 프로그램을 계속 수행하도록 프롬프트를 표시합니다. 사용자가 리턴 키를 누르면 열려진 창이 닫히고 함수가 리턴됩니다. 여기서 프로그래머는 프로그램 종료를 선택할 수 있지만 이 함수는 사용자로부터 적어도 어떤 상호 작용을 필요로 하기 때문에 강제로 사용자가 이 문제에 응답하도록 합니다.

Informix-4GL의 도움말 메시지

Informix-4GL은 런타임 시 일반 사용자에게 도움말을 제공하기 위하여 완전한 도움말 기능 세트를 제공합니다. 올바른 포맷을 사용하여 도움말 파일을 생성한 경우, 이는 특정 Informix-4GL문을 사용하여 자동으로 또는 showhelp 함수를 사용하여 명시적으로 화면에 표시될 수 있습니다. 도움말 메시지는 mkmessage라는 유틸리티를 사용하여 구분된 텍스트 파일로부터 작성됩니다. 이 텍스트 파일의 내용은 다음과 같이 입력됩니다.

```
.100
This is the error message 100. When the showhelp function is
called with and integer parameter 100, this message will be d
isplayed to the screen.

.200
This is the error message 200. When the showhelp function is
```

```
called with and integer parameter 200, this message will be d
isplayed to the screen.
```

도움말 파일은 텍스트 편집기를 사용하여 작성되고 단순한 ASCII 문자로 구성되어야 합니다. 도움말 파일은 다음과 같이 인수를 사용하는 mkmessage 유틸리티를 사용하여 컴파일됩니다.

```
mkmessage input_file output_file
```

이 유틸리티는 출력 파일에 대해 이름 지정 규칙을 가정하거나 강제하지 않습니다. 이 유틸리티는 입력 파일을 가져와 이를 처리하여 출력 파일을 생성합니다. 주의할 점은 출력 파일에 입력 파일과 동일한 이름을 지정할 경우, 출력 파일이 입력 파일로부터 생성되고 출력은 오류 없이 자동으로 원래의 입력 파일을 겹쳐 씁니다.

도움말 메시지는 메뉴 명령문 또는 프롬프트 명령문과 같은 특정 명령문과 함께 사용할 수 있습니다. 이 명령문을 사용하여 온라인 도움말을 트리거하는 기본 키는 Control-W입니다. 그러나 이 키는 options문을 사용하여 재지정할 수 있습니다. 다음 코드는 Informix-4GL에서의 도움말 사용을 보여줍니다.

```
options help file "menu.hlp"

menu "Input"

     command "Insert"
                help 101

     command "Other"
                call showhelp (201)

...
```

이 코드 조각에서 menu문은 help절과 함께 사용되었습니다. 도움말 파일을 식별하기 위해 options문을 사용한 점을 유의하십시오. 도움말 파일은 참조되기 전에 식별되어야 합니다. 그렇지 않으면 런타임 오류가 발생합니다. help절은 해당 메뉴 옵션에 대한 도움말 번호에 해당하는 정수 매개변수를 가져옵니다. 메뉴가 표시되고 커서가 해당 옵션에 있을 때 사용자가 도움말 키를 누르면, 도움말 키

101에 해당하는 도움말 메시지가 표시됩니다. 이것은 내재적인 도움말의 예입니다.

'Other' 메뉴 옵션에 대한 menu 명령에는 명시적 도움말 메시지의 예가 있습니다. 이 옵션을 선택하면, showhelp 함수를 호출하여 전달된 정수 도움말 번호에 대한 도움말 메시지를 표시합니다.

이러한 폼의 명시적인 도움말 메시지는 필드 레벨 도움말에 input 문을 제공하는 데 자주 사용됩니다. input문은 help절을 제공하지만, 이는 input문의 구체적 필드에 대한 것이 아니라 일반적 명령문에 대한 것입니다. 필드 레벨 도움말을 제공하기 위하여 종종 다음과 같은 방법을 사용합니다.

```
options help file "input.hlp",
               help key Control-T
...
input rec1.* from scr-rec.*
     on key (Control-W, F1)
                case
                     when infield(col1)
                              call showhelp(101)
                     when infield(col2)
                              call showhelp(102)
                     when infield(col3)
                              call showhelp(103)

....
```

이 예에서 options문은 먼저 도움말 파일의 이름을 식별하고 표준 도움말 키를 다른 키로 재정의하는 데 사용됩니다. 그런 다음, 입력 명령문을 시작합니다. Control-W 키를 누르면 case문이 실행됩니다. case문 내에서, 현재 필드인지를 판별하기 위하여 infield 함수를 사용하여 각 필드를 테스트합니다. 현재 필드에 따라 showhelp 함수를 호출하여 해당 필드에 대한 도움말 메시지를 표시합니다.

sqlca 레코드

Informix-4GL은 sqlca(SQL 통신 영역) 레코드를 통해 오류를 추적합니다. 이 레코드를 사용하여 오류와 경고를 모두 기록합니다.

오류는 일반적으로 처리 중에 발생하는 예기치 못한 이벤트이며 응용프로그램의 작업에 부정적인 영향을 줄 수 있습니다. 경고는 프로그램 작업 중 허용 가능하다고 생각되는 예기치 못한 이벤트입니다. 오류와 경고 사이의 구분은 때때로 분명하지 않습니다. 프로그래머는 이러한 차이를 알고 적절하게 응답해야 합니다.

오류는 프론트엔드 또는 백엔드에 의해 리턴될 수 있습니다. 백엔드(또는 엔진) 오류는 SQL 처리 중에 발생하여 런타임 시 엔진에 의해 리턴됩니다. SQL 오류가 발생하면 상태 변수는 SQL 오류 코드 값으로 설정되고, sqlca 레코드는 이 오류에 대한 추가 정보로 채워집니다. 예 10-1은 이 sqlca 레코드의 요소를 보여줍니다. 표 10-2는 SQL 오류 코드의 값 범위에 대한 목록입니다.

예 10-1 ***sqlca*** *레코드 구조*

```
define sqlca record
        sqlcode integer,
        sqlerrm char(71),
        sqlerrp char(8),
        sqlerrd array(6) of integer,
        sqlwarn char(8)
end record
```

표 10-2 SQL 오류 코드 값 범위

오류 코드	의미
1-99	운영 체제 오류.
1-200	C-ISAM/RSAM 오류. 이 오류는 하위 레벨 오류입니다. 일반적으로 sqlca.sqlerrd[1]은 ISAM 오류 코드가 있는 경우 이에 대해 설정됩니다. 이는 작업이 실패한 이유에 관한 중요한 정보를 제공합니다.
0	이 코드는 SQL 작업이 성공한 경우 엔진에 의해 설정됩니다. 레코드의 다른 모든 요소는 실행 중인 SQL 작업에 따라 설정됩니다.

sqlerrd 배열

sqlca 레코드의 sqlca.sqlerrd 요소는 엔진이 실행한 최종 SQL 작업에 대한 정보가 들어 있는 6개 정수의 배열입니다. 요소에는 표 10-3에 표시된 정보가 들어 있습니다.

표 10-3 sqlerrd 배열 요소

sqlerrd 요소	목적
1	현 시점에서는 사용되지 않습니다.
2	SQL 작업이 성공적이고 시리얼 컬럼이 있는 테이블에 대한 삽입을 포함하는 경우, 시리얼 값이 리턴됩니다. SQL 오류가 발생한 경우, sqlca.sqlcode < 0이면, 이 요소는 발생한 ISAM 오류로 설정됩니다.
3	처리된 행 수입니다. 업데이트 명령문이 실행되고 데이터베이스가 ANSII 모드가 아닌 경우, 이 요소는 업데이트의 성공 여부를 표시합니다.
4	쿼리에 대한 추정 CPU 비용입니다.
5	처리된 SQL 쿼리 명령문으로의 SQL 오류 오프셋입니다.
6	선택된 마지막 행의 rowid입니다.

sqlwarn 문자열

sqlca 레코드의 sqlwarn 요소는 최종 SQL 작업에서 생성된 경고를 표시하는 단일 문자들로 이루어진 문자열입니다. 이 문자열에는 열린 데이터베이스 유형, 데이터베이스에서 로깅이 사용되고 있는지 여부 및 마지막 SQL문이 Informix ANSII의 확장인지 여부에 관한 유용한 정보가 들어 있습니다. 이 문자열의 요소는 표 10-4와 같이 설정됩니다.

표 10-4 sqlwarn 요소

문자열 요소	목적
1	다른 배열 요소들 중 한 요소가 설정된 경우 W로 설정됩니다. 이 요소를 효과적으로 신호하면 경고가 발생했는지에 대한 정보를 제공합니다.
2	데이터베이스 열기 명령문 실행 시, 데이터베이스가 트랜잭션 로깅을 사용하고 있으면 이 요소는 W로 설정됩니다. select문이 실행되고 문자열이 Informix-4GL 문자 변수에 맞도록 절단되어야 하는 경우, 이 요소는 W로 설정됩니다. (이 요소는 어떤 변수를 절단했는지 표시하지 않습니다. Informix-4GL은 현재 이를 수행하는 방법을 제공하지 않습니다.)

<계속>

표 10-4 sqlwarn 요소

문자열 요소	목적
3	데이터베이스 열기 명령문 실행 시, 열려진 데이터베이스가 ANSI 모드 데이터베이스이면 이 요소는 W로 설정됩니다. select문이 실행되고 select문 내의 집계 함수에 NULL이 발생하면, 이 요소는 W로 설정됩니다.
4	데이터베이스 열기 명령문 실행 시, 열려진 데이터베이스가 Informix-On-line 데이터베이스이면 이 요소는 W로 설정됩니다. 이 요소는 select문이 실행되고, select 목록의 항목 수가 선택된 데이터를 받기 위해 제공된 Informix-4GL 변수 수와 일치하지 않는 경우 설정됩니다.
5	10진수 변환에 대해 float를 사용하는 경우, 이 요소를 W로 설정합니다.
6	마지막 명령문이 Informix ANSI 확장으로 실행되고 DBANSIWARN 환경 변수가 설정된 경우, 이 요소는 W로 설정됩니다.
7	사용하지 않습니다.

프론트엔드 오류는 영(0)으로 나누기 또는 숫자 오버플로와 같이 컴파일 시점에서 트랩될 수 없는 표현식 오류입니다. 응용프로그램에서 이들을 트랩하려면 whenever any error문을 오류를 트랩해야 하는 코드 앞에 배치해야 합니다.

요약

프로그래머는 Informix-4GL 함수를 사용하여 필요한 추가 작업의 수행을 위해 효과적으로 언어를 확장할 수 있습니다. 모듈 방식의 프로그래밍 노력의 일환으로 함수 라이브러리를 작성하여 프로그래머 그룹이 공유할 수 있습니다.

함수가 다른 함수를 작성하는 데 함께 사용될 때, 프로그래머들은 코드를 효과적으로 재사용하고 응용프로그램의 개발에 필요한 코드 라인 수를 줄입니다. 이 방법은 프로그램 작업이 분리되고 이해할 수 있는 모듈 함수 라이브러리 세트로 그룹화되므로 응용프로그램 개발 프로세스를 단축시키고 응용프로그램을 더욱 쉽게 유지보수할 수 있도록 합니다.

Informix-4GL은 유용한 도구 세트를 제공하여 프로그래머가 응용프로그램을 오류에 대해 적절하게 응답하도록 합니다. Informix-4GL의 기본 오류 검사 기능은 생산 환경에서 선호하는 오류 검사 방법이 아니므로 이 기능의 구현은 프로그래머의 결정에 달려있습니다.

집중적이고 일반화된 오류 함수를 사용하면 응용프로그램을 더욱 일관성 있게 만들고 응용프로그램의 개발에 필요한 코딩의 양을 줄일 수 있습니다. 이러한 모듈 방식의 접근은 응용프로그램 개발 직원들에게 단기 및 장기적인 장점을 제공합니다. 단기적인 측면에서, 응용프로그램은 더욱 신속하게 개발될 것입니다. 또한 장기적인 측면에서는 거의 모든 오류 검사가 한 위치에서 유지보수되기 때문에 응용프로그램의 수정이 더욱 용이해질 것입니다.

색인

가

가공된 파일, 112
가상 메모리, 44, 50
    내장 프로시저 캐시, 52
    사전, 52
    세션 데이터, 51
    스레드 데이터, 51
    전역 풀, 52
    정렬 풀, 52
    big buffer, 51
가상 프로세서(VP), 16-19
값 단위 호출, 350
검토 단계, 해시 조인, 200-201
공유 로크, 44
공유 메모리, 66
    가상 메모리, 44, 50
    논리 로그 버퍼, 45-46
    데이터 복제 버퍼, 48
    로크 테이블, 45
    물리 로그 버퍼, 47-48
    버퍼 테이블, 44
    버퍼 풀, 46
    사용자 테이블, 46
    상주 메모리, 44
공유 메모리, 상주 메모리 및 가상 메모리도 참조
    청크 테이블, 45
    통신 부분, 43, 52
    트랜잭션 테이블, 46
    페이지 클리너 테이블, 45
    DBSPACE 테이블, 45
    LRU 관리, 48-49
    TABLESPACE 테이블, 45
공유 메모리 매개변수 변경, 222
    onmode -a, 222-223
    onmode -r, 223-224
공유 메모리 버퍼, 15-16
공유 메모리 버퍼 캐시, 160
공유 메모리의 통신 부분, 53
공유 메모리의 통신 부분, 43, 52
관계형 데이터베이스, Informix 내장 프로시저 언어(SPL) 및 구조화된 쿼리 언어(SQL) 참조
관계형 데이터베이스:
    개념, 246-248

관계, 246-247
다대다, 247
일대다, 247
일대일, 247
성공 사례, 245
속성, 246-247
엔티티, 246-247
역사, 246
정규화, 247-248
정의, 246
제 3 정규형, 248
컬럼 무결성 및 참조 제한 조건, 264-268
확장 SQL문, 249
Informix-SQL, 245-336
SQL DDL문, 249, 250-268
SQL DML문, 249, 268-286
구조화된 쿼리 언어(SQL), 3, 245
복사 작업, 286-287
임시 테이블, 사용, 288-289
확장 SQL문, 249
DDL문, 249, 250-268
DML문, 249, 268-286
IDS, 248-249
구조화된 쿼리 언어(SQL), Informix 내장 프로시저 언어(SPL), SQL DDL문, SQL DML문 참조
load 명령문, 287-288
unload 명령문, 287-288
급속 복구, 20
기본 서버, 33
기본 키, 266
기본 DBSPACE, 33
긴 트랜잭션, 23

나

내장 프로시저 캐시, 52
논리 디스크, 41
레이아웃, 정의, 113-114
논리 로그, 22-23
모니터링, 233-235
상태 플래그, 234
논리 로그 버퍼, 46-47

다

다대다 관계, 247
다중 스레드, 16-19
가상 프로세서(VP), 16-19
스레드 작동, 18-19
대기 작업 모드, 214
대기 큐, 19
대소문자 변환 함수, 327
대칭 멀티프로세서 플랫폼, sar 출력, 159
더티 페이지, 170
데이터 복제, 33-34
동기 복제, 33-34
비동기 복제, 34
데이터 복제 버퍼, 48
데이터 분산, 133-141
분산 정보를 사용하여 분할화 방법 결정, 138-141
시간에 따라 다양, 140-141
데이터 분산 통계, IDS 옵티마이저, 205
데이터 사전 관리자, Extended Parallel Server (XPS), 7
데이터베이스 레벨 사용 권한, 227-228
connect 권한, 227
dba 권한, 228
resource 권한, 228

데이터베이스 로깅, 20-23
논리 로그, 22-23
물리 로그, 20-22
체크포인트, 23
데이터베이스 로깅 상태, 변경, 242-243
데이터베이스 메모리, 42-55
메모리 부여 관리자(MGM), 42
데이터베이스 바인딩:, 화면 폼에서, 392-393
INFORMIX-4GL, 351
데이터베이스 보안, 226-232
데이터베이스 레벨 사용 권한, 227-228
connect 권한, 227
dba 권한, 228
resource 권한, 228
사용자 역할, 229
취소 명령, 사용 권한 제거, 232
테이블 사용 권한, 230-231
데이터베이스 테이블:, 로크 단위, 124-125
페이지 레벨 로크, 124-125
행 레벨 로크, 124-125
분할화, 125-127
방법 개발, 127-133
보관, 128
성능, 128-132
작성, 119-133
데이터베이스 트리거, 311-323
데이터베이스 connect 권한, 227
데이터베이스 dba 권한, 228
데이터베이스 resource 권한, 228
동기 복제, 33-34
동적 해시 조인, 194
디스크 문제, 해결, 166-167
디스크 사용도, 모니터링, 141-146
디스크 스트라이핑 방법, 129
디스크 활동, 모니터링, 164-167

라

라운드 로빈 분할 방법, 129
래치, 43
레벨 0 보관, 239-240
레벨 1 보관, 239-240
레벨 2 보관, 239-240
레코드, 350
로그 성능 문제, 해결, 168
로컬 범위, 변수, 306
로크 테이블, 45
루프 조인, 200
링크, 문제점, 120

마

메모리 누출, 340
메모리 문제, 해결, 159-160
메모리 사용, 모니터링, 160-163
메모리 상주 테이블 및 인덱스, 작성, 331
명령문, 339
모니터링 도구, 8-9
모듈, 387-388
화면 폼 상호 작용, 395-396
INFORMIX-4GL 보고서, 435-437
모듈 프로그래밍, INFORMIX-4GL, 342-343
모듈식 동적 보고서, 430-432
물리 디스크, 41
레이아웃, 정의, 112-113
물리 로그, 20-22
용어 사용, 20-22
물리 로그 버퍼, 47-48
뮤텍스, 19, 43

미러 DBSPACE, 33
미러링, 32-33
미리 읽기 옵션, 198-199

바

반복적 쿼리, 277-279
배열, 350
배치 처리, INFORMIX-4GL 보고서, 423-426
배타적 로크, 44
백엔드 프로세스, 4, 340
버퍼 로크, 44
버퍼 미지정 로깅, 22, 34
버퍼 테이블, 44
버퍼 풀, 공유 메모리, 46
별칭, 테이블, 274-275
병렬 데이터 쿼리(PDQ), 58-59
병렬 데이터베이스 작업, 12-13
보고서 동기화, 437
보고서 블록, 339
보관, 분할화, 128
보관 및 복구, 239-243
    보관 레벨, 239−240
        레벨 0 보관, 239−240
        레벨 1 보관, 239−240
        레벨 2 보관, 239−240
    보관 수행, 240−241
    복구 수행, 240−241
        데이터베이스 로깅 상태 변경, 242−243
        선택 복구, 242
        전체 복구, 242
보조 DBSPACE, 33
복구, 241-242
    데이터베이스 로깅 상태, 변경, 242−243
    선택 복구, 242
    수행, 241−243
    전체 복구, 242
복제 버퍼, 34
부분 분해 문제, 469−475
분리된 인덱스, 253
분할화, 125−127, 253−55
    동일 디스크로 분할화, 190−191
    방법 개발, 127−133
    병렬 액세스, 125
    병렬 처리, 42
    보관, 128
        방법, 132−133
    성능, 128−132
        DSS 분할화 방법, 130−132
        OLTP 분할화 전략, 128−130
    수평, 42
    정적, 206
    조각 제거, 126
        필터, 191−192
    쿼리 튜닝, 206−210
        분할화 스키마, 206−210
        inv_date별 분할, 207−208
        inv_flag별 분할, 208−209
        ship_to_code별 분할, 209−210
    테이블, 24
    표현식 기준, 253
    IDS 분할화 정의, 252−255
    RAID, 210
        방법, 210
블록, 339
블록 특수 장치, 113
비동기 복제, 34
비어 있는 버퍼, 44
비용 기반의 옵티마이저, 56, 201−202

비프로시저문, 337
빌드 단계, 해시 조인, 201

사

사용자 세션, 67
사용자 역할, 229
    작성 및 사용, 229
사용자 정의 함수, Extended Parallel Server (XPS), 7
사용자 테이블, 46
사전 캐시, 52
사전 확장 테이블, 121
삭제 트리거, 314-316
삽입 커서, 383
삽입 트리거, 312-314
상속, Informix Dynamic Server9(IDS9), 8
상주 메모리, 43-46
    공유 메모리 버퍼 풀, 46
    논리 로그 버퍼, 46−47
    데이터 복제 버퍼, 48
    물리 로그 버퍼, 47−48
    LRU 관리, 48−49
서버, 4
서버 아키텍처, 40-55
    구성 요소, 40−41
    논리 디스크, 41
    데이터베이스 메모리, 44−54
    디스크 입출력, 41−42
    물리 디스크, 41
서브쿼리, 사용, 275-277
선택 복구, 242
세션, 180
세션 데이터, 51
소스 코드 모듈, 343
소프트 링크, 118-119
속성, 246−247
수신 버퍼, 34
수정되지 않은 버퍼, 44
수정된 버퍼, 44
수평 분할화, 42
스레드, 53−54
스레드 데이터, 51
스캔 스레드, 126
스케줄러, 7
스크롤 커서, 378−385
    문제, 380−382
스택, 51
스팀 파이프 통신, 55
시리얼 커서, 377
시스템 예약 페이지, 34−36
    PAGE_ARCH, 36
    PAGE_CKPT, 35−36
    PAGE_CONFIG, 35
    PAGE_DBSP, 36
    PAGE_MCHUNK, 36
    PAGE_PCHUNK, 36
    PAGE_PZERO, 35
시스템 자원 문제, 해결, 186−188
시스템 카탈로그, 37−38, 52

아

액세스 계획, 200
액세스 목록, 버퍼 테이블, 44
언어 도구, 9
업데이트 커서, 382
업데이트 트리거, 316−318
엔티티, 246−247
연결 연산자, INFORMIX−4GL, 362−363
연결 코서버, 60
연산자, 339

연속 읽기, 인덱스 읽기 대, 197-198
영역 키, 127
오프라인 작업 모드, 214
온라인 모드, 214
옵티마이저 지시문, 332-334
외부 조인, 279-280
외부 테이블, 200
운영 체제 자원 모니터, 사용, 159
원시 디스크, 112
응용프로그램, 튜닝, 179-186
이중 프로세스 아키텍처, 4
익스텐트, 24, 121
인덱스, 분할화, 128-130
인덱스 읽기, 200
    연속 읽기 대, 197-198
인터넷 개발 도구, 9-10
일대다 관계, 247
일대일 관계, 247
임시 테이블, 사용, 288-289

자

자동 인덱스, 200
자료형, 25-34, 340, 343-353
    레코드, 350
    배열, 350
    콤플렉스 자료형, 350
    특수 자료형, 31-32
    Character, 31, 349
    Date, 28-29, 347
    Datetime, 29, 347-348
    Decimal, 26-27, 345-346
    Float, 28, 347
    Integer, 26, 345
    Interval, 30-31, 348-349
    Money, 28, 346-347
    NULL 데이터 값, 352-353
    Smallfloat, 26, 345
    Smallint, 25, 344
    Varchar, 31
자료형 확장성, Informix Dynamic Server9(IDS 9), 8
자원 경쟁, 161
전역 범위 변수, 307-308
    SPL에서 사용, 307-308
전역 풀, 52
전체 복구, 242
전체 테이블 스캔, 식별 및 문제 해결, 202-205
정규화, 247-248
정렬 병합 조인, 56
정렬 풀, 52
정적 분할, 206
제 3 정규형, 247-248
조각 제거, 126
    필터, 191-192
조건식, INFORMIX-4GL, 361-362
조인 계획, 200
조인 기준, 270
준비 큐, 19
중첩 루프 조인, 200
집계 함수, 273, 422

차

청크, 24-25, 111
    삭제, 118
    지정하는 데 소프트 링크 사용, 118-119
    DBSPACE에 추가, 116-117
청크 사용 가능 목록, 24, 113
청크 테이블, 45
체크포인트, 23
    평가 및 튜닝, 178-179

추적, 309
취소 명령, 사용 권한 제거, 232

카

캐시 적중률, 161
커서 닫기, 384-385
컨텍스트 전환, 16
콤플렉스 자료형, 350
쿼리 계획, 199-202
    비용 기반 옵티마이저, 201-202
    액세스 계획, 200
    조인 계획, 200
    평가 순서, 201
    Set optimization 명령, 201-202
쿼리 세션, onstat -g ses를 사용하여 추적, 180-183
쿼리 옵티마이저, XPS, 7
쿼리 튜닝, 179-186, 188-199
    동일 디스크로 분할화, 190-191
    전체 테이블 스캔, 202-205
    테이블 분할화 사용, 206-11
    필터 및 조각 제거, 191-192
    PDQPRIORITY, 192-197
    set explain 명령, 188-190
쿼리:, 결과, 처리, 57
    병렬로 실행,
    엔진을 통해 추적, 55-57
    최적화, 55
    쿼리 계획, 60
    튜닝, 쿼리 튜닝 참조, 188-199
큐 처리, 19
클라이언트/서버 컴퓨팅, 5

타

테이블 별칭 지정, 274-275
테이블 분할화, 24
테이블 사용 권한,
    데이터베이스 보안, 230-232
    with grant option문 사용, 231
테이블 스캔, 200
테이블 중심의 동적 보고서, 446-456
    main 프로그램 블록, 448-449
    on every row 보고서 블록, 454-456
    proc_line 함수, 450
    rpt_drv 함수, 448-449
    sql_rpt 보고서, 451-453
튜닝 방법, IDS, 149-150
트랜잭션 롤백, 20
트랜잭션 완료, 20
트랜잭션 테이블, 46
특수 자료형, 31-32

파

파일 링크, 119
페이지, 20, 113
    시스템 예약 페이지, 34-36
    청크 사용 가능 목록, 24, 113
페이지 구분, INFORMIX-4GL 보고서, 427-430
페이지 레벨 로크, 124-125
페이지 클리너 테이블, 45
페이지 플러셔 스레드, 170
평가 순서, 201
표현식 기준 분할화, 253
프로그램 블록, 339
프로그램 스레드, 16
프로세서 친화도, 18

프로세스 간 통신(IPC), 4
프로시저문, 337
프론트엔드 프로세스, 4, 340
플러시, 170
필드, 화면, 392
필터, 조각 제거, 191-192
필터 기준, 270

하

하드 링크, 119
하드웨어 미러링, 33
함수, 339
함수 블록, 339
해시 조인, 56, 200
    검토 단계, 200-201
    빌드 단계, 201
해시 테이블, 44
행 레벨의 로크, 124
현재 필드, 강조 표시, 408-409
협력적 다중 작업 환경, 18
화면 폼, 391-457
    다중 윈도우, 관리, 396-401
    데이터베이스 바인딩, 392-393
    모듈성 및 화면 폼 상호 작용, 395-396
    입력 필드, 통과, 406-408
    현재 필드, 강조표시, 408-409
    화면 크기, 관리, 391-393
    화면 폼 식별자, 395
    화면 폼 프로그래밍, 392
    current window 명령문, 396-401
    display 명령문, 401-405
        화면 폼 상호 작용, 403-405
    formonly 화면, 393-394
    INFORMIX-4GL로 처리, 394-395
    input 명령문, 405-409
    Interrupt key, 412-413
    termcap, 391, 409-413
        기능 키 재정의, 410
        Insert key 및 delete key 해제, 410
        termcap 파일에서 기능 키 추가, 410-411
화면 폼 식별자, 395
화면 필드, 392
확장 SQL문, 249
휴면 큐, 19
힙, 51

A

acct_totals 보고서, 441-446
AFF_NPROC 매개변수, 93
after field 명령문, 406-408
ALARMPROGRAM 매개변수, 88
alter 사용 권한, 230
alter fragment문, 140-141, 166
alter table문, 123-125, 249, 260-262
attributes 섹션, 화면 폼, 392

B

b-트리 인덱스, 128-130
BAR_ACT_LOG 매개변수, 101
BAR_BSALIB_PATH 매개변수, 103
BAR_NB_XPORT_COUNT 매개변수, 102
BAR_RETRY 매개변수, 102
BAR_XFER_BUF_SIZE 매개변수, 102
before field 명령문, 406-408

before group of/after group of 절, 422-423
begin/end 명령문 블록, 299-300
Big buffer,
BLOB, 32, 46
BUFFERS 매개변수, 94

C

call 명령문, 295-296
case 명령문, 328-329
CDR_DSLOCKWAIT 매개변수, 101
CDR_EVALTHREADS 매개변수, 101
CDR_LOGBUFFERS 매개변수, 101
CDR_QUEMEM 매개변수, 101
Character, 31, 349
C-ISAM, 3
CKPTINTVL 매개변수, 97
CLEANERS 매개변수, 95
Codd, E.F, 246
CONSOLE 매개변수, 87-88
construct 명령문, 438-446
qry_acct 함수, 439
rpt_drv 함수, 440−441
continue 명령문, 300-301
cpio 포맷 파일, 67
CPU 자원 문제, 해결, 159-160
CPU VP, 추가, 187-188
create database 명령문, 121, 249, 250-251
자료형, 251−252
컬럼 무결성 및 참조 제한 조건, 264−268
고유 컬럼 작성, 266−267
제한 조건 이름 지정, 267−268
테이블 레벨의 기본키와 외부키 정의, 265−266
컬럼 이름, 251−252
테이블 이름, 251−252
IDS 분할화 정의, 252−255
create trigger 명령문, 312−314
create view 명령문, 249, 255−258
뷰를 사용하여 보안 시행, 256−257
뷰에 키워드 사용, 257−258
전체 쿼리 참조, 258
current window 명령문, 396−401

D

Data Director 제품, 8
database 섹션, 화면 폼, 392
DATASKIP 매개변수, 106−107
Date, 28−29, 347
Datetime, 29, 347−348
DBANSIWARN 환경 변수, 72
DBCENTURY 환경 변수, 72−73
DBDELIMTER 환경 변수, 73
DBEDIT 환경 변수, 73
DBFLTMASK 환경 변수, 74
dbinfo 함수, 309−311
DBLANG 환경 변수, 73−74
DBMONEY 환경 변수, 74
DBPATH 환경 변수, 74−75
DBPRINT 환경 변수, 75
dbschema 명령, 134
DBSERVERALIASES 매개변수, 90
DBSERVERNAME 매개변수, 90
DBSPACE, 24−25, 111−119
번호, 143
소유자, 142, 145
유지보수, 114
이름 지정, 142−145

작성 및 삭제, 115-116
청크 추가, 116-117
플래그, 142
onspaces 명령, 114-118
root, 37
DBSPACE 테이블, 45
DBSPACETEMP 매개변수, 103, 289
DBSPACETEMP 환경변수, 76
DBSPACE 미러링, 117-118
DBTIME 환경 변수, 76
DEADLOCK_TIMEOUT 매개변수, 91-92
Decimal, 26-27, 345-346
decode 함수, 329-330
defer interrupt 명령문, 412-413
define 명령문, 306
delete 명령문, 285-286
delete 사용 권한, 230
display 명령문, 395-99
화면 폼 상호 작용, 403-405
display array 명령문, 411
DRAUTO 매개변수, 99-100
DRINTERVAL 매개변수, 100
DRLOSTANDFOUND 매개변수, 100-101
drop database 명령문, 249, 260
drop table 명령문, 249, 259
drop view 명령문, 249, 259
DS_MAX_QUERIES 매개변수, 105-106
DS_MAX_SCANS 매개변수, 106
DS_TOTAL_MEMORY 매개변수, 106
DUMPCNT 매개변수, 104
DUMPCORE 매개변수, 104
DUMPDIR 매개변수, 103
DUMPGCORE 매개변수, 104
DUMPSHMEM 매개변수, 104

E

end function문, 339
err_get 함수, 482
err_handler 함수, 484
err_print 함수, 482
err_quit 함수, 482
errorlog 함수, 482
ESQL/C(Informix Embedded SQl for C), 9
EXISTS 연산자, SPL 조건식, 291-292
exit 명령문, 308-309
exit program문, 339
Extended Parallel Server(XPS), 7-8
extent size절, 122

F

FET_BUF_SIZE 환경 변수, 76-77
FILLFACTOR 매개변수, 105
finish report 명령문, 414
first page header 섹션, 보고서, 419-421
Float, 28, 347
FLRU 큐, 49
for 명령문, 297-299
foreach 명령문, 294-295
보고서 생성, 416-417
form4gl, 391
format 섹션, 419
before group of/after group of 절, 422-423
formonly 화면, 393-394
from절, select 명령문, 269

G

group sum 명령문, 426

H

HTML(Hyper Text Markup Language), 9

I

IDS
가상 프로세서(VP), 16-19
공유 메모리 버퍼, 15-16
구조화된 쿼리 언어(SQL), 248-250
기본 개념, 15-23
다중 스레드, 16-19
스레드 작동, 18-19
대기 큐, 19
데이터 복제, 33-34
데이터 분산, 133-141
데이터 스토리지, 24-25
데이터베이스 관리, 213-246
데이터베이스 로깅, 20-23
데이터베이스 보안, 226-232
데이터베이스 트리거, 311-323
디스크, 111-147
디스크 문제, 해결, 166-167
디스크 사용도, 141-146
디스크 활동, 모니터링, 164-167
로그, 관리, 233-239
로그 성능 문제, 해결, 168
메모리 문제, 해결, 159-160
메모리 사용, 모니터링, 160-163
문자열 처리 함수, 323-327
물리 로그 위치, 169
물리 로그 크기, 169-170
물리 로그 활동, 평가 및 튜닝, 168-170
뮤텍스, 19
미러링, 32-33
버전 7.3 기능, 327-334
메모리 상주 테이블 및 인덱스, 331
옵티마이저 지시문, 332-334
case 명령문, 328-329
decode 함수, 329-330
nvl 함수, 330
병렬 데이터 쿼리(PDQ), 58-59
병렬 처리 및 분할화, 42
서브디렉토리, 67
설치 및 구성, 65-109
소프트웨어 디렉토리 구조, 67
스레드(프로세스), 53-54
시스템 예약 페이지, 34-36
시스템 자원 문제, 해결, 186-188
시스템 카탈로그, 37-38
아키텍처, 40-55
구성 요소, 40-41
논리 디스크, 41
데이터베이스 메모리, 42-55
디스크 입출력, 41-42
물리 디스크, 41
옵션 및 확장, 59-62
XPS 서버, 59-62
자료형, 25-34
준비 큐, 19
청크, 24-25
체크포인트, 평가 및 튜닝, 178-179
쿼리, 튜닝, 188-199
쿼리 및 응용프로그램, 튜닝, 179-186
통신, 54-55
공유 메모리, 54
스팀 파이프, 55
TCP/IP와 IPX/SPX, 55
튜닝 방법, 149-150

환경 변수, 68-79
휴면 큐, 19
CPU 자원 문제, 해결, 159-160
CPU VP, 추가, 187-188
IDS 엔진의 로그 활동, 167-168
LRU 성능, 평가, 171-174
onconfig 파일 매개변수, 80-108
onstat 유틸리티, 151-159
root DBSPACE, 37
sysmaster 데이터베이스, 38-39
IDS 구성, 65-68
설치, 67-68
설치 준비, 66
설치 프로그램, 실행, 68
IDS 데이터베이스 트리거, 311-323
내장 프로시저 실행 시 사용, 320-323
사용자 정의 복제에 내장 프로시저 사용, 320-323
삭제 트리거, 314-316
삽입 트리거, 312-314
업데이트 트리거, 316-318
유용성, 318-319
create trigger 명령문, 312-314
when절, 사용 , 319-320
IDS 디스크 관리, 111-147
관리, 233-239
논리 로그:, 모니터링, 233-235
삭제, 238
추가, 237-238
물리 로그:, 크기 및 위치 변경, 235-237
onconfig 파일 변경, 237
onparams 명령, 237
IDS 로그, 백업, 234-235
IDS 문자열 처리 함수, 323-327
대소문자 변환 함수, 327
lpad 함수, 326
replace 함수, 325-326
rpad 함수, 326
substr 함수, 325
substring 함수, 324
trim 함수, 324
IDS 분할화 정의, 252-255
IDS 엔진의 로그 활동, 167-168
IDS 옵션, 59-62
상속, 62
아키텍처, 61
IDS 환경 변수, 68-79
DBANSIWARN, 72
DBCENTURY, 72-73
DBDELIMTER, 73
DBDIT, 73
DBFLTMASK, 74
DBLANG, 73-74
DBMONEY, 74
DBPATH, 74-75
DBPRINT, 75
DBSPACETEMP, 76
DBTIME, 76
FET_BUF_SIZE, 76-77
INFORMIXSERVER, 71
INFORMIXSHMBASE, 77
INFORMIXSQLHOSTS, 77
INFORMIXSTACKSIZE, 78
INFORMIXTERM, 78
ONCONFIG, 70-71
PATH, 71-72
PDQPRIORITY, 78-79
PSORT_DBTEMP, 79
PSORT_NPROCS, 79

TERM, 79
TERMCAP, 79
TERMINFO, 71
IDS9, 사용자 정의 함수, 7
상속, 7
자료형 확장성, 6
IDS의 데이터베이스 관리, 213-246
데이터베이스 시작 및 중지, 213-215
보관 및 복구, 239-243
보관 및 복구 참조
온라인 엔진 수정, 216-226
공유 메모리 매개변수 변경, 222-224
다중 사용자 모드로 엔진 가져오기, 219
단일 사용자 모드로 엔진 가져오기, 218-219
오프라인 모드로 엔진 가져오기, 217-218
CPU 자원 추가 및 제거, 220
DSS 매개변수 변경, 221-222
IECC(Informix Enterprise Command Center), 8
if 명령문, 291-293
IN 연산자, SPL 조건식, 291-292
index 사용 권한, 230
INFORMIIXSQLHOSTS 환경 변수, 77
Informix 내장 프로시저 언어(SPL), 289-311
내장 프로시저, 작성, 289-291
전역 범위 변수, 사용, 307-308
begin/end 명령문 블록, 299-300
call 명령문, 295-296
continue 명령문, 300-301
dbinfo 함수, 309-311
define 명령문, 306
exit 명령문, 308-309
for 명령문, 297-299
foreach 명령문, 294-295
IDS 데이터베이스 트리거, 311-323
if 명령문, 291-293
let 명령문, 302-303
on exception 명령문, 305-306
raise exception 명령문, 301-302
return 명령문과 returning 절, 293-294
system 명령문, 304
trace 명령문, 디버그 결과 생성, 309
while 루프, 296-297
Informix 도구, 8
모니터링 도구, 8-9
언어 도구, 9
인터넷 개발 도구, 9-10
Informix 서버 아키텍처, 40-55
Informix 웹 통합 옵션, 8
Informix 자료형, 25-34, 340, 343-353
Informix 제품 아키텍처, 10-12
디스크 입출력, 10-11
메모리(버퍼), 10-11
병렬 데이터베이스 작업, 12-13
스레드(프로세스), 12
통신, 12
Informix 제품군, 1-14
관계형 데이터베이스와 Informix-RDS, 3
도구, 8-10
아키텍처, 10-12
역사, 1-10
API(application programmer interfaces), 2
C-ISAM, 2-3
Informix Dynamic Server7(IDS7), 5-6
Informix Dynamic Server9(IDS9), 6
Informix ESQL/C(Embedded SQI for C), 9
Informix SQL Editor, 8

INFORMIX-4GL, 9
Informix-SQL Engine(SE), 3-5
Informix-Turbo, 5-6
XPS, 7-8
Informix Dynamic Server9(IDS9), 6
Informix SQL Editor, 8
INFORMIX-4GL, 337-389
4GL 응용프로그램, 340-342
모듈, 342-343
백엔드, 340
사용자 상호 작용, 342
프론트엔드, 340
화면 폼, 342
데이터 변환, 353-358
문자 변환, 356-357
산술 연산자와 숫자 변환 규칙, 357-358
초기화, 358
표현식에서 지정 및 변환, 356
데이터 입력, 342
데이터베이스 바인딩, 366-369
데이터베이스 상호작용, 340
도움말 메시지, 485-487
메모리 제어, 339-340
명령문, 339
모듈 방식, 387-388
모듈 프로그래밍, 342-343
반복, 468-475
보고 및 데이터 필터링, 342
보고서, 413-456
뷰, 385-387
언어 구문 오류, 475
언어 상수, 358-359
언어 세대, 역사, 338
언어 연산자, 339
연결 연산자, 362-363
연산자, 339
오류 로깅, 478-480
오류 메시지, 475
오류 처리, 475-790
조건식, 361-362
커서, 374, 376-377
닫기, 384-385
삽입 커서, 383
스크롤 커서, 378-379
업데이트 커서, 382
유형, 377
정의, 376-377
프로그램 블록, 339
프론트엔드 대 백엔드 활동, 370-371
함수, 339
구현, 460-461
매개변수의 변수 수 전달, 464-468
변수 사용, 462-464
표현식, 461-462
함수 매개변수, 460
화면 폼, 391-457
화면 폼 오류, 475
흐름 제어문, 359-360
declare문, 최적화, 371-374
for 명령문, 360
foreach문, 375
Informix 자료형, 343-353
데이터베이스 바인딩, 351
레코드, 350
배열, 350
콤플렉스 자료형, 350
Character, 349
Date, 347
Datetime, 347-348
Decimal, 345
Float, 347

Integer, 344
Interval, 348-349
Money, 346-347
NULL 데이터 값, 352-353
Smallfloat, 345
Smallint, 344
prepare 명령문, 369-370
SQL 오류, 475
SQL문을 사용하여 유형 변환, 365-366
SQL문의 데이터 변수, 375-376
sqlca 레코드, 487-488
sqlerrd 배열, 488-489
sqlwarn 문자열, 489-490
INFORMIX-4GL 보고서, 413-456
기본 보고서, 생성, 416-417
동적 컬럼 위치, 작성, 421-422
모듈성, 435-437
모듈식 동적 보고서, 430-432
배치 처리, 423-426
보고서 동기화, 437
보고서 명령문, 414-415
보고서 문제, 432-435
선언, 415-416
테이블 중심의 동적 보고서, 446-456
main 프로그램 블록, 448-449
on every row 보고서 블록, 454-456
proc_line 함수, 450
rpt_drv 함수, 448-449
sql_rpt 보고서, 451-453
텍스트 필드, 관리, 427
페이지 구분, 427-430
acct_totals 보고서, 441-446
construct 명령문, 438-44
qry_acct 함수, 439
rpt_drv 함수, 440-441
first page header 섹션, 419-421
format 섹션, 419
before group of/after group of절, 422-423
order by 섹션, 419
output 섹션, 418-419
print 명령문, 426
INFORMIX-4GL의 도움말 메시지, 485-487
INFORMIX-4GL의 반복, 468-475
반복, 정의, 468-469
부분 분해 문제, 469-475
INFORMIX-4GL의 오류 처리, 475-490
오류 로깅, 478-480
오류 메시지, 482
오류 프롬프트 함수, 484-485
일반화된 오류 처리 루틴, 483-484
whenever any error 명령문, 480-482
whenever error 명령문, 476-478
INFORMIX-4GL의 커서:
닫기, 384-385
삽입 커서, 383
스크롤 커서, 378-379
문제, 380-382
업데이트 커서, 382
INFORMIXDIR 환경 변수, 67
Informix-RDS, 3
INFORMIXSERVER 환경 변수, 71
INFORMIXSHMBASE 환경 변수, 77
Informix-SQL, 24, 119, 245
관계형 데이터베이스, 245-336
Informix-SQL, 구조화된 쿼리 언어(SQL) 참조
Informix-SQL Engine, 3-5
INFORMIXSTACKSIZE 환경 변수, 51, 78
INFORMIXTERM 환경 변수, 78

Informix-Turbo, 5-6
initialize 명령문, 466
input array 명령문, 410
insert 명령문, 283-284
insert 사용 권한, 230
Integer, 26, 345
Interval, 30-31, 348-349
IPX/SPX, 55

L

LBU_PRESERVE 매개변수, 108
let 명령문, 302-303
load 명령문, 287-288
LOCKS 매개변수, 94
LOGBUFF 매개변수, 168
LOGFILES 매개변수, 86
LOGSIZE 매개변수, 85, 169
LOGSMAX 매개변수, 95
lpad 함수, 326
LRU 관리, 48-49
LRU 버퍼, 170
    가중된 업데이트 로드, 매개변수 설정, 177
    경미한 업데이트 로드, 매개변수 설정, 176-177
    구성 매개변수, 174
    의사 결정 지원 시스템, 매개변수 설정, 177
    LRU 관리, 48-49
        튜닝, 174-178
    LRU 성능, 평가, 171-174
    LRU 튜닝, 반복 벤치마킹, 177-178
LRU 성능, 평가, 171-174
LRU_MAX_DIRTY 매개변수, 97, 172-177
LRU_MIN_DIRTY 매개변수, 97-98, 173-177
LRUS 매개변수, 97, 174-178
LTAPEBLK 매개변수, 89
LTAPEDEV 매개변수, 89
LTAPSIZE 매개변수, 89
LTXEHWM 매개변수, 98
LTXHWM 매개변수, 98

M

main 프로그램 블록, 테이블 중심의 동적 보고서, 448-449
MAX_PDQPRIORITY 매개변수, 105
MAXARR Integer 변수, 359
MAXRECS Integer 변수, 359
MIRROR 매개변수, 84-85
MIRROROFFSET 매개변수, 85
MIRRORPATH 매개변수, 85
mkmessage 유틸리티, 485-487
MLRU 큐, 49
Money, 28, 346-347
MSGPATH 매개변수, 86-87
MULTIPROCESSOR 매개변수, 92
nettype 매개변수, sqlhosts 파일, 67-68
NETTYPE 매개변수, 91

N

next field 명령문, 406-408
next size절, 122
NOAGE 매개변수, 93
NOT IN 연산자, SPL 조건식, 291-292
NULL 데이터 값, 352-353
NUMAIOVPS 매개변수, 94
NUMCPUVPS 매개변수, 92
nvl 함수, 330

O

OFF_RECVRY_THREADS 매개변수, 99
on every row 보고서 블록, 테이블 중심 동적 보고서, 454-456
on exception 명령문, 305-306
ON_RECVRY_THREADS 매개변수, 99
onarchive 유틸리티, 239
onbar 유틸리티, 239
oncheck -pe 명령, 123
onconfig 파일:,
매개변수, 80-108
AFF_NPROC, 93
ALARMPROGRAM, 88
BAR_ACT_LOG, 101
BAR_BSALIB_PATH, 103
BAR_NB_XPORT_COUNT, 102
BAR_RETRY, 102
BAR_XFER_BUF_SIZE, 102
BUFFERS, 94
CDR_DSLOCKWAIT, 101
CDR_EVALTHREADS, 101
CDR_LOGBUFFERS, 101
CDR_QUEMEM, 101
CKPTINTVL, 97
CLEANERS, 95
CONSOLE, 87-88
DATASKIP, 106-107
DBSERVERALIASES, 90
DBSERVERNAME, 90
DBSPACETEMP, 103
DEADLOCK_TIMEOUT, 91-92
DRAUTO, 99-100
DRINTERVAL, 100
DRLOSTANDFOUND, 100-101
DS_MAX_QUERIES, 105-106
DS_MAX_SCANS, 106
DS_TOTAL_MEMORY, 106
DUMPCNT, 104
DUMPCORE, 104
DUMPDIR, 103
DUMPGCORE, 104
DUMPSHMEM, 104
FILLFACTOR, 105
LBU_PRESERVE, 108
LOCKS, 94
LOGBUFF, 95
LOGFILES, 86
LOGSIZE, 86-87
LOGSMAX, 95
LRU_MAX_DIRTY, 97
LRU_MIN_DIRTY, 97-98
LRUS, 97
LTAPEBLK, 89, 90
LTAPEDEV, 89, 90
LTAPESIZE, 89
LTXEHWM, 98
LTXHWM, 98
MAX_PDQPRIORITY, 105
MIRROR, 84-85
MIRROROFFSET, 85
MIRRORPATH, 85
MSGPATH, 87
MULTIPROCESSOR, 92
NETTYPE, 91
NOAGE, 93
NUMAIOVPS, 94
NUMCPUVPS, 92
OFF_RECVRY_THREADS, 99
ON_RECVRY_THREADS, 99
ONDBSPACEDOWN, 107-108
OPCACHEMAX, 108

OPTCOMPIND, 107
PHYSBUFF, 94
PHYSDBS, 85
RA_PAGES, 103
RA_THRESHOLD, 103
RESIDENT, 92
ROOTNAME, 83-84
ROOTOFFSET, 84
ROOTPATH, 84
ROOTSIZE, 84
SERVERNUM, 90
SHMBASE, 95
SHMVIRTSIZE, 95-96
SINGLE_CPU_VP, 92
STACKSIZE, 99
STAGEBLOB, 90
TAPEBLK, 88
TAPEDEV, 88
TAPESIZE, 88-89
TXTIMEOUT, 98
USEOSTIME, 105
변경, 237
ONCONFIG 환경 변수, 70-71
ONDBSPACEDOWN 매개변수, 107-108
oninit 명령, 213-215
옵션, 214
-i 옵션, 215
onmode 명령, 160
기타 명령, 224-226
옵션, 217
-a 옵션, 222-223
-D 매개변수, 221
-d 옵션, 226
-F 옵션, 224
-k 옵션, 217-218
-l 옵션, 225
-M 매개변수, 221-222
-m 옵션, 219
-n 옵션, 224
-p 옵션, 220
-R 옵션, 226
-r 옵션, 223
-s 옵션, 218-219
-u 옵션, 218-219
-y 옵션, 217-218
-z <pid> 옵션, 225
-Z <transaction id> 옵션, 225
PDQ 설정, 197
onmonitor 유틸리티, 236
onparams 명령, 237
onspaces 명령, 114-118
onstat 유틸리티, 151-159
미리읽기 옵션, 198-199
실행, 154
옵션 및 매개변수, 151-152
자원 병목 현상 식별, 155-159
-d 매개변수, 154
-g 매개변수, 152-154
-g act 명령, 184-185
-g glo 명령, 155-158
-g iof 명령, 168
-g ses 명령, 쿼리 추적 세션, 180-183
-g sql 명령, 183-186
IDS 모니터링을 위해 반복, 162-163
-l 매개변수, 168, 233
onstat VP 출력, 해석, 158-159
-p 매개변수, 154, 161
-ut 명령, 154
비동기 입출력 모니터링, 185-186
ontape 명령, 239-241
-a 옵션, 235
-c 옵션, 235

Opaque 데이터 객체, 6
OPCACHEMAX 매개변수, 108
OPTCOMPIND 매개변수, 107
order by 섹션, 419
output 섹션, 418-419
output to report 명령문, 414-415

P

PAGE_ARCH, 36
PAGE_CKPT, 35-36
PAGE_CONFIG, 35
PAGE_DBSP, 36
PAGE_MCHUNK, 36
PAGE_PCHUNK, 36
PAGE_PZERO, 35
PATH 환경 변수, 71-72
PDQPRIORITY 환경 변수, 58-59, 69, 78-79, 192-197
    설정 및 제어, 195-197
PHYDBS 매개변수, 85, 169
PHYSBUFF 매개변수, 94
PHYSFILE 매개변수, 236
prepare 명령문, 369-370
    최적화, 370
print 명령문, 426
print file 명령문, 426
proc_line 함수, 테이블 중심 동적 보고서, 450
PSORT_DBTEMP 환경 변수, 79
PSORT_NPROCS 환경 변수, 79

Q

qry_acct 함수, 439

R

RA_PAGES 매개변수, 103, 198
RA_THRESHOLD 매개변수, 103, 198
RAID, 분할화, 210
raise exception 명령문, 301-302
rename column 명령문, 262-263
rename table 명령문, 263
replace 함수, 325-326
report 명령문, 415-416
report to 명령문, 418
RESIDENT 매개변수, 92
return 명령문과 returning 절, 293-294
right margin 명령문, 418
ROM(Relational Object Manager), 8
root DBSPACE, 37
ROOTNAME 매개변수, 83-84
ROOTOFFSET 매개변수, 84
ROOTPATH 매개변수, 84
ROOTSIZE 매개변수, 84
rowid, 46
rpad 함수, 326
rpt_drv 함수, 440-441

S

sar 명령, 159
select 명령문, 268-280
    레코드 그룹화, 273-274
    리턴되는 최대 행 수, 지정, 272
    반복적 쿼리, 277-279
    외부 조인, 279-280
    정렬 순서, 식별, 271-272
    테이블 별칭 지정, 274-275
    표시 레이블, 지정, 271
    from절, 269

select절, 269
union절, 280-283
select 사용 권한, 230
select절, 269
SERVERNUM 매개변수, 90
set debug file 명령문, 309
set explain 명령, 188-190
set optimization 명령, 201-202
SHMBASE 매개변수, 95
SHMVIRTSIZE 매개변수, 95-96
SINGLE_CPU_VP 매개변수, 92
Smallfloat, 26, 345
Smallint, 25, 344
SQL DDL 명령문, 249, 250-268
alter table 명령문, 260-262
create database 명령문, 249, 250-251
create table문, 121, 251
자료형, 251-252
컬럼 무결성/참조 제한 조건, 264-268
컬럼 이름, 251-252
테이블 이름, 251-252
IDS 분할화 정의, 252-255
create view 명령문, 249, 255-258
뷰를 사용하여 보안 시행, 256-257
뷰에 키워드 사용, 257-258
전체 쿼리 참조, 258
drop database 명령문, 249, 260
drop table 명령문, 249, 259
drop view 명령문, 249, 259
rename column 명령문, 262-263
rename table 명령문, 263
SQL DML 명령문, 249, 268-286
insert 명령문, 283-284
select 명령문, 268-280
레코드 그룹화, 273-274
리턴되는 최대 행 수 지정, 272
반복적 쿼리, 277-279
서브쿼리 사용, 275-277
외부 조인, 279
정렬 순서 식별, 271-272
테이블 별칭 지정, 274-275
표시 레이블 지정, 271
from절, 269
select절, 269
union절, 280-283
where절, 270-271
update 명령문, 22, 284-285
sql_rpt 보고서, 테이블 중심 동적 보고서, 451-453
sqlca 레코드, 487-488
sqlerrd 배열, 488-489
sqlhosts 파일, 67-68, 215
sqlwarn 문자열, 489-490
STACKSIZE 매개변수, 99
STACKSIZE 환경 변수, 51
STAGEBLOB 매개변수, 90
start report 명령문, 415-416
startlog 함수, 478
substr 함수, 325
substring 함수, 324
sysmaster 데이터베이스, 38-39
system 명령문, 304

T

TABLESPACE, 113-114
개념, 25
페이지, 24-25
TABLESPACE TABLESPACE, 25, 113-114
TAPEBLK 매개변수, 88

TAPEDEV 매개변수, 88, 240
TAPESIZE 매개변수, 88-89
tar 포맷 파일, 67
TABLESPACE 테이블, 45
TCP/IP(Transmission Control Protocol/Internet Protocol), 5
IPX/SPX, 55
TERM 환경 변수, 79
termcap, 391, 409-413
기능 키 재정의, 410
termcap 파일에 기능 키 추가, 410-411
Insert key 및 delete key 해제, 410
TERMCAP 환경 변수, 79
terminfo 정의 파일, 391
TERMINFO 환경 변수, 71
trace 명령문, 디버그 결과 작성에 사용, 309
trim 함수, 324
TXTIMEOUT 매개변수, 98, 225

U

union절, 280-283
값 세그먼트화에 사용, 281-283
unload 명령문, 287-288
update 명령문, 22, 284-285
update 사용 권한, 230
update statistics 명령, 134-135, 205, 245
USEOSTIME 매개변수, 105

V

Varchar, 31
vmstat 명령, 159-60

W

when절, 트리거에 사용, 319-320
whenever any error 명령문, 480-482
whenever error 명령문, 476-478
while 명령문, 296-297
with append 절, set debug file 명령문, 309
with grant option문, 231

X

XPS, 아키텍처, 59-62
연결 코서버, 60
쿼리 처리, 60-61
XPS 서버, 59-62

INFORMIX POWER REFERENCE

ART TAYLOR

모든 DBA와 개발자를 위한 완벽한 Informix Dynamic Server™ 참조서!

Informix 관리자와 개발자가 참고할 수 있는 데이터베이스 아키텍처, 개발, 관리, 모니터링 및 튜닝 등에 관한 권위 있는 정보가 이 책 한 권에 모두 담겨져 있습니다. 이 INFORMIX POWER REFERENCE에는 모든 Informix 데이터베이스 전문가들이 신뢰성과 성능을 극대화하고자 할 때 꼭 필요한 기술과 지식이 모두 모여있습니다.

이 책을 통해 공유 메모리, 내부 다중 스레드 및 기타 성능에 직접적인 영향을 미치는 아키텍처 요소를 이해할 수 있습니다. 정적 데이터베이스 분할화를 통해 관리를 단순화하고, 데이터베이스 액세스를 위해 Informix SQL을 최대한 활용하는 방법을 알 수 있으며, 또한 내장 프로시저와 데이터베이스 트리거 작업 방법과 제한 조건을 통한 데이터 무결성 향상 방법 등에 대해서도 알게 됩니다. 다음은 이 책에서 다루는 내용입니다.

- Informix 데이터베이스 엔진에 대한 포괄적 개요
- 고성능 Informix Dynamic Server(IDS) 설치 및 구성
- IDS 물리 및 논리 디스크 레이아웃 관리
- CPU, 메모리, 디스크, LRU 큐 등의 자원 병목 현상 식별
- 효율성 극대화를 위한 쿼리 작성과 튜닝

이 책은 온라인 엔진 수정, DSS 매개변수 변경, onmode 명령, 데이터베이스 보안, 로그 관리, 보관, 복구 등의 데이터베이스 관리에 대한 정보가 들어 있는 실용적인 종합 안내서입니다. 전문가가 제공하는 팁과 예제들을 통해 한층 뛰어난 화면 폼과 보고서를 작성할 수 있으며 Informix-4GL을 최대한으로 활용할 수 있습니다. 완벽하고 간편하며 최신의 정보가 들어 있는 INFORMIX POWER REFERENCE는 DBA와 개발자 모두에게 아주 귀중한 자료가 될 것입니다.

저자에 대하여

ART TAYLOR는 *Informix-NewEra: A Programmer's Guide*와 *Advanced Informix-4GL Programming* 등 Informix와 관련한 훌륭한 안내서를 여러 권 저술했습니다. 데이터베이스 업계에서 13년 넘게 종사했으며, 이 가운데 7년은 Informix Software 컨설팅 부서에서 컨설턴트로 근무하였습니다.

INFORMIX POWER REFERENCE