ORACLE 9i 물리설계(物理設計)

작성 및 번역: 김 도근(롯데 정보통신 LOTTE CARD IS팀, OTN ACE)

본 자료는 OTN JAPAN에서 1년에 걸쳐서 연재되어서 많은 호응을 받았던 자료로서 내용을 번역하고 재구성한 것입니다.

많은 분들이 참조하셔서 도움이 되셨음 합니다.

연재자:

Oracle Japan CROSS 인터스트리부

OracleDirect SC 그룹

中家 裕之

목차

제 1 부 : DB전체의 설계

◆ 제1장 블록 사이즈 해결 방안

블록 사이즈를 결정하는 기본적인 지침

블록 사이즈는 CREATE DATABASE 커멘드의 실행 시에 초기화 파라미터 DB_BLOCK_SIZE에 의해서 지정된다. 기본적으로 2,048 , 4,096 , 8,192 , 16,384 , 32,768 바이트 중 하나가 선택 되어진다. 단, 32.768 바이트는 64비트 계열의 일부 OS 버전용 ORACLE 밖에 지정되지 않는다. 디폴트의 경우 대부분 2,048 바이트가 지정된다.

어떤 블록 사이즈를 지정하는 것은 시스템의 특징에 의존적이다. 테이블의 블록 사이즈의 크기에 의한 DB의 영향

을 고려한 다음 종합적으로 판단 되어 져야 할 것이다.

(표) 블록 사이즈 크기에 따른 DB의 영향

	블록 사이즈가 작음	블록 사이즈가 큼
블록에 저장되는 레코드 수	작다	많다
블록 I/O 비용	낮다.	높다
풀 스캔 I/O 비용	DB_FILE_MULTIBLOCK_READ_COUNT에 의존 、같은 값이 라면 블록 사이즈가 큰 편이 COST 가 높다.	
캐시 히트율	높다	낮다
트랙잭션 경합 발생가능성	작다	크다
ROW CHANING 발생가능성	크다	작다
세그먼트 압축 효과	작다	크다
적합한 시스템 형태	OLTP계열	DSS/DWH계열

일반적으로、

- 동일한 블록에 대해 복수개의 세션으로부터 동일하게 갱신되는 트랙잭션이 발생할 가능성이 높다.
- 테이블 중의 일부 레코드에 대해 삭제가 빈번하게 발생할 수 있다.
- 캐시 히트 율을 높이고 싶다.

이런 경우에는 작은 블록 사이즈를 사용하는 것이 좋다。한편

- LOB 타입과 LONG/LONG RAW 타입의 데이터가 많이 이용되고 있다.
- 레코드의 길이가 긴 테이블이 많다.
- 테이블과 인덱스의 압축 기능을 이용하고 싶다

이럴 경우에는 큰 블록 사이즈를 사용하는 것이 좋다。
표에 있는 특징에 대해서 설계하려는 시스템이 복합적인 성격 혹은 모호할 경우 일반적으로 8,192 바이트를 사용하는 것을 적극 추천한다.

멀티 블록 사이즈에 대해서

오라클 9 버전부터 테이블 스페이스에 블록 사이즈를 복수 개로 선택하는 것이 가능해졌다. 테이블 스페이스 영역에 있어 블록 사이즈를 지정하는 것은、아래와 같이 CREATE TABLESPACE문으로 지정된다.

SQL> create tablespace tbs1 datafile ... *blocksize 16k*;

하지만, ALTER TABLESPACE문으로 블록 사이즈를 변경하는 것은 불가능하다. 초기화 파라미터 DB_BLOCK_SIZE에 지정한 블록사이즈는 CREATE TABLESPACE문으로 블록사이즈를 지정하지 않고 실행하면 디폴트로 이 값을 이용한다.

멀티 블록 사이즈를 이용하면, 버퍼·캐쉬가 블록 사이즈 마다 확보된다.

블록 사이즈가 한 종류의 경우는 초기화 파라미터 DB_BOCK_BUFFERS 혹은 DB_CACHE_SIZE로 버퍼·캐쉬의 크기로 지정하고 있다. 멀티 블록 사이즈를 사용했을 경우는 초기화 파라미터 DB_CACHE_SIZE는 디폴트의 블록 사이즈의 테이블 스페이스의 캐쉬가 되며, 그 외의 블록 사이즈의 테이블 스페이스 용의 캐쉬는 초기화 파라미터 DB_*n*K_CACHE_SIZE(*n*는2/4/8/16/32의 어느 쪽 이든으로, 블록 사이즈에 맞춘다)로 크기를 지정합니다.디폴트의 블록 사이즈용의 DB_*n*K_CACHE_SIZE는 지정해도 무시된다.

이 블록 사이즈가 다르면 버퍼·캐쉬가 따로 설정된다는 특징을 이용하여, 집중적으로 캐싱하고 싶은 세그먼트(segment)와 캐쉬의 필요성의 적은 세그먼트(segment)를 다른 블록 사이즈의 테이블 스페이스 영역에 배치해, 캐쉬를 이용하는 테크닉을 이용할 수 있다. 또, LOB 테이블과 같은 레코드 사이즈가 긴 테이블을 저장하는 테이블 스페이스 영역의 블록 사이즈만 크게 하여 행체인의 발생을 억제하는 방법을 사용하는 것도 팁이 될 수 있다.

◆ 제2장 캐릭터·셋의 결정 방법

Oracle9*i*으로 이용할 수 있는 캐릭터 셋

Oracle9*i*로 한글을 취급하는 경우, 테이블에 있는 캐릭터·셋(Oracle상의 문자 코드)을 선택할 수 있다.

(표) Oracle9*i*로 지정할 수 있는, 한글을 취급할 수 있는 주요 캐릭터·셋

캐릭터 셋	KO16KSC5601	KO16MSWIN949	UTF8	AL32UTF8
한글 지원상태	한글 2350자	KO16KSC5601 + 확장 8822자(총 11172자)	한글 11172 자	한글 11172 자
캐릭터셋/인코딩 버전	한글 완성형	완성형코드포함 확장된8822자는 MSWin 코드페이	8.1.6 이전 : Unicode 2.1 8.1.7 이후: Unicode 3.	9i Rel1: Unicode 3.0 9i Rel2 : Unicode 3.1 10g Rel1 : Unicode 3.2

		지949에따라배열		1/0g Rel2 : Unicode 4.0
한글바이트	2바이트	2바이트	3바이트	3바이트
지원버전	7.x	8.0.6 이상	8.0 이상	9i R1 이상
Database Characterset으로 설정 가능 여부	가능	가능	가능	가능
National Characterset으로 설정 가능 여부	불가능	불가능	가능	불가능

캐릭터·세트를 결정하는 관점

한국어 · 영어 이외의 문자를 동시에 취급할 필요가 있을까?

예를 들면 중국어나 포르투갈어 등을 한글과 동시에 취급할 필요가 있는 경우, KO16KSC5601이나 KO16MSWIN949에서는 중국어나 포르투갈어는 취급할 수 없다. 이러한 경우에는 Unicode를 취급할 수 있는 AL32UTF8을 이용하는 것이 좋다. Oracle8*i* 이전부터의 데이터로 하위 버전과의 호환성을 중시하는 경우에는 UTF8를 이용한다.

다국어 캐릭터·셋에 대해

다국어 캐릭터 셋(create database문의 national character set구로 지정하는 캐릭터 세트.NCHAR형이나 NVARCHAR2형, NCLOB형등에서 사용되는 문자 코드)는 디폴트로 AL16UTF16를 지정하는 것이 좋다. 구 버젼의 Oracle에서는 KO16KSC5601 등에 대응한 캐릭터 세트를 사용 할 수 있지만, Oracle9*i*이후에서는 다국어 캐릭터 셋는 Unicode가 전제가 된다.

◆ 제3장 제어 파일의 작성에 있어서의 고려사항

다중화의 권유

제어 파일은 1개가 깨지면 DB가 정지해 버린다. 복구를 위해서도 컨트롤 파일을 다중화하는 것이 좋다. 통상 Oracle에서는 3 파일 이상의 다중화를 추천 하고 있다. 또, 내부적인 장해에 대비하기 위해, 다중화한 제어 파일은 가능한 한 다른 디스크에 배치한다.

제어 파일의 용량 산정

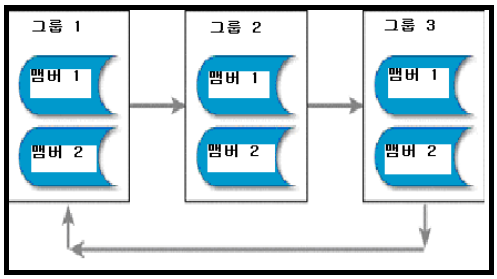
제어 파일의 용량에 관해서는 통상 수 MB 정도의 크기이므로, 크기에 대해서는 특히 고려할 필요는 없다.다만, 백업 툴의 Recovery Manager의 리커버리·카탈로그(백업에 관한 데이터)를 제어 파일 내에 작성하는 경우는, DB 규모나 백업 방침에 따라서는 수백 MB까지 사이즈가 부풀기도 한다

◆ 제4장 REDO 로그 파일의 작성에 있어서의 고려사항

그룹과 멤버의 차이

DB의 변경 정보를 기록하는 REDO 로그 파일에는 그룹과 멤버가 존재한다.어느 정도 DB관리의 경험이 있는 사람이라도 혼동 하고 있는 케이스가 많기 때문에, 그룹과 멤버에 대한 설명을 먼저 해 보자.

우선 그룹이란, 어떤 시점에 써 대상이 되는, 1개 내지 2개 이상의 REDO 로그 파일의 집합이다. 한편 멤버란, 어떤 A 그룹에 포함되는 개별의 REDO 로그 파일을 말한다. REDO 로그 파일에 변경 정보를 쓸 때는 최초의 그룹의 REDO 로그 파일로부터 차례로 변경 정보가 써져 있는 그룹의 REDO 로그 파일의 끝까지 변경 정보가 써지면, 다음의 그룹에 바뀌어(로그 스위치) 변경 정보가 써진다. 이 때 멤버가 다수개가 존재하면, 이전 멤버에 대해서 동시에 같은 내용의 기입한다. 따라서 같은 그룹내의 멤버는 항상 같은 내용이다.



그룹의 구성에 관한 고려사항

REDO 로그 파일의 초기 작성은 CREATE DATABASE 커멘드로 실시하지만, 이 때 최소 2개의 그룹을 지정할 필요가 있다. 아카이브(archive) 로그·모드로 DB를 운용하는 경우는, 아카이브(archive)중에 REDO 로그가 순환되어져서 REDO 로그 파일에 REWRITE되는 것을 방지하기 위한 시스템 HANGUP 현상을 방지 하기 위해서, 최소한

3개의 그룹을 작성하도록 한다.
아카이브(archive) 로그·모드로 DB를 운용하는 경우는, 각 그룹을 다른 물리 디스크에 배치하면, 아카이브(archive)와 WRITE I/O가 분산되어 퍼포먼스가 향상된다.

멤버의 구성에 관한 고려사항

멤버는 최소 하나에서도 가동한다. 하지만 두개 이상의 멤버를 작성하면, 어떤 A 멤버가 망가져도, 그룹 중 최소 하나의 멤버가 살아 남고 있다면 DB는 계속해서 가동이 된다. 그러므로 장해 대책으로서 멤버는 반드시 두개 이상 작성해 둔다..
같은 그룹에 소속하는 멤버는 동시에 WRITE를 한다. I/O를 분산시켜 퍼포먼스를 향상시키기 위해서, 각 멤버는 각기 다른 물리 디스크에 배치해 두도록 하자.

REDO 로그 파일의 용량 산정

REDO 로그 파일에 쓰여지는 데이터량은, 대상의 SQL나 갱신 데이터 양 등에 의해서 크게 바뀌기 위해, 사전에 정확하게 추측하는 것은 불가능하다. 그러므로 소규모 시스템이면 1~10 MB, 중 규모 시스템이면 10~100 MB, 대규모 시스템이면 100 MB이상의 크기로 우선 작성해, 실제의 REDO 발생량을 보고 운용의 과정에서 따로 조정하는 것이 일반적이다..

트랜잭션(transaction) 처리 중에 로그 스위치(LOG SWITCH)가 발생하면 그 트랜잭션(transaction)의 퍼포먼스가 떨어지기 때문에, 특히 OLTP계의 처리가 중심의 경우는, 처리의 피크타임 시에는 로그 스위치가 발생하지 않게 할 필요가 있다. 운용 REDO 로그 갱신량에 관한 데이터를 모아 피크 타임 시에 로그 스위치가 발생되지 않을 정도 크기로 조정하도록 하자. 또, 일반적으로 REDO 로그 파일의 크기가 큰 만큼 로그 스위치의 발생의 가능성이 적기 때문에, 퍼포먼스는 향상되게 된다. 그러나 한편 리커버리에 필요로 하는 시간은 길어지기 때문에, 이러한 트레이드 오프도 고려하도록 한다

◆ 제5장 OMF 구성의 소개

OMF란?

OMF(Oracle Managed Files)는 지정 디렉토리에 정리해 테이블 스페이스이나 제어 파일, REDO 로그 파일을 배치하는 것으로, DB관리의 번거로움을 크게 줄여주는 Oracle9*i*의 신기능입니다.
OMF를 이용하는 경우는 초기화 파라미터 DB_CREATE_FILE_DEST에 파일을 작성하고 싶은 디렉토리를 지정한

다. 이 설정을 하는 것으로, 테이블 스페이스나 제어 파일등을 작성할 때에 파일명을 지정할 필요가 없어진다.

테이블 스페이스를 작성하는 SQL문은 아래와 단순하게 작성 할 수 있다. 이때 파일명은 ORACLE이 자동으로 명명한다. 또 테이블 스페이스를 삭제하면 데이터파일도 자동으로 삭제가 된다.

```
SQL> create tablespace tbs1;
```

또, 파일 용량이나 확장 방법 등도 자동으로 이하와 같이 정해진다

- 파일 용량 : 100MB
- 데이터 파일의 확장 : 1 블록 사이즈씩 무제한하게 자동 확장
  (물론 OS나 디스크 용량에 의한 제한되어지는)
- extent 관리 : 로컬 관리(AUTOALLOCATE)

다만, CREATE/ALTER TABLESPACE문으로 디폴트를 바꾸어 작성·변경할 수 있다. 그러나 AUTOEXTENT은 권장하지 않는 옵션으로 적당하게 변경해서 생성하는 것을 권장한다. 이하는 파일 용량은 200 MB에, AUTOEXTENT 기능없이 변경해서 OMF를 이용해 테이블 스페이스를 작성하는 예이다.

```
SQL> create tablespace tbs1 datafile size 200m autoextend off;
```

OMF에 적절한 시스템

OMF는 아래와 같은 시스템에서 권장된다.

- **소규모 시스템**

  디스크가 1개 밖에 없는 듯한 소규모 시스템에서는 물리 배치를 생각하는 여지가 없기 때문에, 보다 간편하게 관리하기 위해서 권장되어진다.

- **패키지·소프트의 연구 최종 단계에 Oracle를 이용하는 경우**

  패키지·소프트의 경우 DB의 관리 요소를 별로 늘릴 수 없는 것이 많기 때문에, OMF를 이용하는 것으로 데이터 파일의 관리 요소를 줄일 수 있다

- **SAME 구성을 이용하는 경우**

  SAME(Stripe And Mirror Everything)란, 어떤 A 디스크 장치에 있는 디스크 전부를 정리해 스트라이핑 해서, 1개의 거대한 디스크로서 취급하는 방식이다. SAME 구성을 취하면, 모든 디스크에 대해서

I/O부하가 균등하게 분산되기 위해, 비교적 간단히 높은 I/O퍼포먼스를 유지시키는 것이 가능하게 된다. 또, 디스크가 1개가 되기 위해, 물리 배치에 관한 설계가 편해진다.OMF를 SAME 구성과 병용하면 보다 설계·관리의 부하를 줄일 수 있다.

제어 파일·REDO 로그 파일의 OMF 관리

앞에서 서명한 제어 파일, REDO 로그 파일도 OMF에 의한 관리가 가능하다. 그러나, 제어 파일이나 REDO 로그 파일을 다중화해도 다른 디스크에 분산 배치할 수 없게 된다.이 문제에 대한 대책으로서 초기화 파라미터 DB_CREATE_ONLINE_LOG_DEST_*n*가 준비되어 있다. 이 파라미터에 제어 파일·REDO 로그 파일 파일을 배치하고 싶은 디렉토리를 지정한다.
n(은)는 1~5까지의 숫자가 들어간다. 예를 들면 DB_CREATE_ONLINE_LOG_DEST_1, DB_CREATE_ONLINE_LOG_DEST_ 2, DB_CREATE_ONLINE_LOG_DEST_3의 3개의 파라미터를 지정해, 각각 다른 디렉토리를 지정하면, 제어 파일·REDO 로그 파일이 3개에 다중화가 된다. 이 파라미터를 지정하지 않으면 초기화 파라미터 DB_CREATE_FILE_DEST로 지정한 디렉토리에 작성이 된다.

제 2 부 : 테이블 영역의 설계

◆ 제1장 테이블 스페이스 분할·배치의 관점

테이블 스페이스 분할· 배치의 관점

데이터베이스는 다수의 테이블 스페이스로 구성된다. 테이블 스페이스의 분할을 검토할 때 , 주로 3개의 관점이 있습니다.

- 관리성
- 내장해성(耐障害性)
- 퍼포먼스

관리성의 관점

데이타베이스 관리를 용이하게 하기 위해서, 이하 열거하는 목표점에 유의한다.

- 단편화 등 퍼포먼스의 문제가 발생하기 않게 한다
- 디스크의 추가 등 테이블 스페이스 재편성에 수반하는 작업 수를 줄인다

- 관리자에게 있어서 알기 쉽다.

등과 같은 메리트를 얻을 수 있다.

(1) 시스템 테이블 스페이스 , UNDO 테이블 스페이스 , TEMPOARAY 스페이스 영역은 분리되어서 작성하자

시스템 테이블 스페이스 , UNDO 테이블 스페이스 , TEMPOARAY 스페이스 독립된 테이블 스페이스로 작성한다. 이러한 테이블 스페이스 영역에는 일반 유저의 테이블이나 인덱스는 생성하지 않게 주의한다.

(2) OMF(Oracle Managed Files)를 이용하자

OMF는 테이블 스페이스 영역의 자동 관리를 목적으로 하는 기능이다

(3) 읽기 전용 세그먼트(segment)를 별도의 공간에 분리한다.

읽기 전용의 테이블이나 인덱스는 읽기 전용의 테이블 스페이스 영역에 저장한다. 읽기 전용 테이블 스페이스에 저장하는 것으로 인해 읽기 전용인 것을 명시적으로 알 수 있게 한다 .또, 한 번 백업을 하면, 갱신이 없기 때문에 정기적으로 백업을 할 필요가 없어진다. 테이블이나 인덱스를 파티션 화 해서 파티션마다 테이블 스페이스를 나누면, 갱신이 발생하지 않는 파티션 만 읽기 전용으로 할 수도 있다.

(4) 저장 오브젝트로 나누자.

예를 들면 테이블과 인덱스는 다른 테이블 스페이스 영역에 생성한다, 혹은 같은 테이블에서도 통상의 테이블과 LOB 테이블은 다른 테이블 스페이스 영역에 생성한다, 이와 같이 , 오브젝트 마다 테이블 스페이스 영역을 나누면 DB규모가 커지는 만큼, 나중에 관리가 용이해진다.

(5) 유저(schema)로 분리해서 나누자

예를 들면 SCOTT 유저 소유의 오브젝트와 SMITH 유저 소유의 오브젝트를 다른 테이블 스페이스에 생성한다, 이와 같이 , 오브젝트 소유 유저 마다 테이블 스페이스 영역을 나눈다. DB규모가 큰 만큼 관리가 용이해진다.

(6) 업무용도로 나누자

예를 들면 경리 시스템으로 사용하는 오브젝트와 인사 시스템으로 사용하는 오브젝트, 양쪽 모두로 사용하는 오브젝트를 다른 테이블 스페이스 영역에 생성한다. 즉, 업무 마다 테이블 영역 따로 나눈다. DB규모가 커짐에 따라 관리가 용이해진다.

내장해성(耐障害性)의 관점

장해가 발생되지 않도록, 혹은 장해가 발생해도 영향 범위가 줄어들도록, 아래 사항 참조로 유의한다.

(1) 디스크의 분할

대용량의 디스크를 소량 준비하는 것보다도, 소용량의 디스크를 대량으로 준비해, 최대한으로 데이터를 분산시키는 것으로, 장해의 영향을 작게 한다고 하는 의미로 내장해성을 확보할 수 있다. 무엇보다, 디스크의 대용량화에 의해서 이 대책을 세우는 것은 어려울지도 모른다.

(2) 동일 디스크내 있을 정도의 파일을 나누어 배치하자

동일 디스크상에서 파티션, 디렉토리, 파일과 같은 단위로 나누는 것도 디스크를 분할하는 것에 비교하면 효과는 작다, 장해 범위의 축소하는 하는 효과는 있다.

(3) 데이터의 중요도에 따라 내장해성이 높은 디스크 장치를 이용하자

간단히 실행할 수 있는 캐이스는 아니지만, DB내의 데이터의 중요도에 따라 차이가 있어, 비교적 디스크로 할애할 수 있는 예산이 한정되어 있는 경우는, 중요한 데이터는 RAID 구성 디스크 장치의 테이블 스페이스 영역을 배치하고, 중요하지 않은 데이터는 RAID가 아닌 디스크 장치의 테이블 스페이스 영역을 배치한다고 하는 것도 생각 할 수 있다.

퍼포먼스의 관점

대용량의 디스크를 소량 준비하는 것보다도, 소용량의 디스크를 대량으로 준비해, 최대한으로 데이터를 분산시키는 것으로, 내장해성 뿐만 아니라 퍼포먼스도 향상시킬 수 있다.

(1) 테이블과 그 테이블에 대한 인덱스

인덱스 스캔이 실행되는 경우, 같은 디스크에 테이블과 인덱스가 있으면, 테이블과 인덱스의 사이의 HDD의 ARM의 이동이 빈번히 발생한다. 검색 대상 레코드수가 많을 정도 퍼포먼스에 영향을 미친다. .

(2) 결합 대상의 테이블사이

결합 방법에 의해 차이가 있지만, 결합 대상의 테이블사이의 HDD의 ARM 이동이 빈번히 발생한다.검색 대상 레코드수가 많을 정도 퍼포먼스에 영향을 미친다.

(3) 병렬 처리의 대상이 되는 테이블이나 인덱스

병렬 처리는 복수의 프로세스를 시작하고, 각 프로세스가 같은 테이블이나 인덱스에 액세스 하기 위해, 같은 디스크에 배치하고 있으면 I/O의 경합이 발생해, 오히려 퍼포먼스가 떨어지는 일이 있다. 따라서 세그먼트(segment)에 대해서 복수의 데이터 파일을 작성하든지, 파티션화 해서 데이터 파일을 나누는 등, 별도의 디스크에 배치하도록 한다.

(4) 시스템 테이블 스페이스

시스템 테이블 스페이스는 기타 테이블 스페이스, REDO 로그 파일과는 다른 디스크에 배치하도록 한다.특히 오브젝트수가 많은 시스템, 감사 이력을 테이블에 기록하는 시스템, 멀티 마스터·replication나 advanced· 큐잉을 이용하는 시스템에서는 시스템 테이블 스페이스의 I/O가 다분히 발생한다.시스템 테이블스페이스에서의 I/O가 다발하는 경우, 시스템 테이블 스페이스 자체를 복수의 데이터 파일로 작성해, 데이터 파일을 분산시키면 유리하다.

(5) UNDO 테이블 스페이스

UNDO테이블 스페이스은 기타 테이블 스페이스, REDO 로그 파일과는 다른 디스크에 배치하도록 한다.특히 갱신처리가 많은 시스템에서는 UNDO테이블 스페이스에의 I/O가 다발적으로 발생한다. UNDO테이블 스페이스에서의 I/O가 다발하는 경우, UNDO테이블 스페이스를 복수의 데이터 파일로 작성해, 데이터 파일을 분산시키면 유리하다.

(6) 임시 테이블 스페이스

임시 테이블스페이스 영역은 기타 테이블 스페이스, REDO 로그 파일과는 다른 디스크에 배치하도록 한다 특히 대량 소트 처리가 많은 시스템에서는 임시 테이블스페이스에서의 I/O가 다발적으로 발생한다. 임시 테이블스페이스에서의 I/O가 다발하는 경우, 임시 테이블스페이스 자체를 복수의 데이터 파일로 작성해, 데이터 파일을 분산시키면 유리하다. 임시 테이블스페이스의 경우는 임시 테이블 스페이스 그 자체를 복수 작성해, 유저에 따라서 각각 다른 임시테이블 스페이스를 할당해서 사용하는 방법도 있다.

◆ 제2장 로컬 관리 테이블 스페이스(LMT)

로컬 관리 테이블 스페이스란?

로컬 관리 테이블 스페이스는 Oracle8*i*부터 채용된, 새로운 extent 관리의 방법을 채용한 테이블 스페이스이다. 로컬 관리 테이블 스페이스에 대해, 기존버전의 extent 관리의 방법을 채용한 테이블 스페이스를 딕셔너리 관리 테이블 스페이스이라고 부른다 .양자의 extent 관리의 차이는, 우선 딕셔너리 관리 테이블 스페이스는 그 이름과 같이 시스템 테이블 스페이스에 있는 딕셔너리에서 extent의 관리를 실시하고 있다.한편 로컬 관리 테이블 스페이스는 테이블 스페이스의 데이터 파일의 헤더 부분에 64 KB의 영역을 잡아, 이 영역에서 비트 맵을 이용해 extent 관리를 실시하고 있다. Oracle9*i*R1로부터 로컬 관리 테이블스페이스가 디폴트의 테이블 스페이스가 되었다. Oracle9*i*R2에서는 시스템 테이블 스페이스를 로컬 관리로 할 수 있게 되었다(R1에는 디폴트는 시스템 테이블 스페이스는 딕셔너리 관리하고 로컬 관리로 할 수 없다)

로컬 관리 테이블 스페이스의 장점

로컬 관리 테이블스페이스는 딕셔너리 관리 방식에 비해 많은 장점이 있다.

(1) 퍼포먼스의 향상

딕셔너리 테이블 스페이스에서는 extent 관리는 테이블 스페이스에서 집중되어 실행된다. 따라서 extent의 추가나 삭제의 처리는 딕셔너리에 대한 I/O가 많기 때문에, 이러한 처리가 하면 퍼포먼스에 큰 영향을 준다.한편 로컬 관리 테이블 스페이스는 extent 관리를 각 데이블 스페이스에서 실시하기 위해, 시스템 테이블 스페이스에서의 I/O는 최소한으로 억제시킨다.또, 딕셔너리의 갱신은 UNDO의 생성을 수반하지만, 로컬 관리 테이블 스페이스의 헤더의 비트 맵의 갱신은 UNDO를 거의 생성하지 않는다.로컬 관리 테이블 스페이스 에서는 딕셔너리 관리 테이블 스페이스에 비해 다음과 같은 처리가 빠르게 진행된다.

- extent의 확장
- UNDO 세그먼트(segment)를 이용하는 처리(갱신 SQL등)
- TRUNCATE
- 세그먼트(segment)의 DROP

(2) 세그먼트(segment)의 설계·관리의 간소화

로컬 관리 테이블 스페이스에서의 extent 관리 방법의 변경에 수반해, extent 관리를 위한 파라미터가 줄어든다.따라서, 설계·관리가 간단해진다. 또, extent 확보의 방법이 단순하게 되기 위해, 그 점에서도 관리가 편해진다(자세한 것은(3)을 참조).로컬 관리 테이블 스페이스에 있어 CREATE/ALTER TABLESPACE문으로 지정 불필요하게 된 파라미터는 다음과 같다.

- DEFAULT STORAGE구
- MINIMUM EXTENT
- TEMPORARY

CREATE/ALTER TABLE, CREATE/ALTER INDEX의 파라미터에 대해서는 다음 번 이후에 설명한다.

(3) 단편화의 감소

로컬 테이블 스페이스에서는 extent 확장하는 방법이 2가지가 있다.하나는 디폴트의 AUTOALLOCATE 지정으로, 세그먼트(segment) 사이즈에 따라 64KB, 1MB, 8MB, 64 MB의 extent를 Oracle가 자동으로 확보합니다.또 하나는 UNIFORM 지정으로, CREATE TABLESPACE시로 지정한 고정 사이즈로 extent를 확보합니다.AUTOALLOCATE 지정 때는 소수의, UNIFORM 지정 때는 하나의 사이즈의 extent 밖에 작성되지 않기 때문에, extent·레벨로 미사용이 되는 영역의 발생이 방지된다.

(4) 테이블스페이스에 대한 coalesce가 불필요

딕셔너리 관리 테이블 스페이스에서는 연속한 빈extent를 하나의 큰 extent에 결합하기 위해서 coalescing 처리가 필요했다. 그러나 로컬 관리에서는 coalesce 하지 않아도 연속한 빈 extent를 인식 할 수 있다.

로컬 관리 테이블 스페이스의 단점

로컬 관리 테이블 스페이스는 딕셔내리 방식에 비해 많은 장점이 있지만, 단점도 일부 존재한다.

(1) DIRECT 처리에 대해 퍼포먼스 저하

다이렉트·로드나 다이렉트·로드·인서트 처리는, 데이터를 일단 임시 세그먼트(segment)에 쓰고, 실제로 쓴 크기에서 임시 세그먼트(segment)를 테이블 내지 인덱스의 extent로 변환하는 것이 고속화의 한 요인이 되어 있다. 그러나 로컬 관리 테이블 스페이스는 extent 사이즈가 정해져 있기 위해, 실제로 쓴 크기는 아니고 정해진 크기로 임시 세그먼트(segment)를 작성하지 않으면 안 되기 때문에, extent의 formatting에 관한 LOSS가 발생한다. 하지만, 늦어진다고 해도 딕셔너리 관리 방식에 비해 늦은 것 뿐이어서, 로컬 관리 방식상의 종래 패스보다는 속도가 더 빨라진다.

(2) 치밀한 EXTENT 관리를 할 수 없다.

관리가 간단하게 되는 반면, 테이블이나 인덱스에 필요한 크기에 딱 맞는 extent·사이즈를 개별 지정하는 것이 불가능해진다. 딕셔너리 관리 방식에서 필요 용량을 빈틈없이 결정해 관리했을 경우에 비하면, extent 내의 미사용 영역이 발생하기 쉽다

로컬 관리 방식과 딕셔너리 관리 방식 중 어느것을 사용할까?

로컬 관리 방식은 딕셔너리 관리 방식에 비해 많은 장점이 있는 것은 일반적으로 알려진 사실이다 .Oracle9*i*R1에서는 테이블 스페이스의 디폴트가 되어 있는 것도 있기 때문에, 기본적으로는 로컬 관리 방식을 사용하도록 권장하고 있다. 로컬 관리 방식이 대세이고 앞으로는 로컬 관리 방식을 기본적으로 사용하자.

로컬 관리 테이블 스페이스 설정에 관한 Tips

(1) 데이터 파일의 용량 산정

이하의 순서로 산정합니다.

1. 데이터 파일에 생성하는 테이블이나 인덱스의 견적 합계 사이즈에 여유 치를 더한 값을 산출합니다.
2. 1.의 값을, AUTOALLOCATE 지정의 경우는 64 KB의 배수에, UNIFORM 지정의 경우는 UNIFORM SIZE의 배수가 되도록 한다.
3. 2.의 값에 테이블 스페이스의 헤더와 extent 관리용의 비트 맵의 영역으로서 64 KB를 더한다.

여유치가 충분한 값이면, 2.나 3을 실행할 필요는 없지만, 이 계산 방법이라면 데이터 파일이 최대한 이용되었을 때에 파일의 마지막에 쓸데 없는 빈 곳이 생기지 않는다. 무엇보다 요즈음의 디스크 용량이면 수 MB 정도의 낭비가 있어도 문제가 되지 않는다고 생각되므로, 계산이 귀찮으면 1만으로도 계산이 충분하다.

(2) AUTOALLOCATE와 UNIFORM의 어느 쪽을 이용해야 할까?

AUTOALLOCATE 지정은 세그먼트(segment)·사이즈에 대한 extent를 자동으로 할당하기 위해, UNIFORM에 비해 영역이 효율적으로 이용할 수 있다. 한편 extent 사이즈가 몇 종류가 되기 위해, extent의 추가가 반복할 때에, extent간에 이용할 수 없는 단편화 영역이 발생할 가능성이 있다.한편 UNIFORM 지정은 extent 사이즈가 고정이므로 매우 알기 쉽다고 하는 이점이 있다. 영역의 효율적인 이용에 대해서는, UNIFORM SIZE의 조정으로 그만큼 헛됨은 나오지 않게 된다. 또, UNIFORM 지정은(1) 방법으로 사이즈를 설정하면 extent·레벨로의 미사용 영역의 단편화는 발생하지 않는다. 기본적으로 UNIFORM 지정을 이용하는 것을 추천한다. AUTOALLOCATE 지정은 영역의 지정이 느슨해도 상관없다고 판단할 수 있는 시스템(소규모 시스템이나 개발기의 DB 등), 소량 데이터용의 테이블 스페이스 영역 등에 사용하는 것을 추천한다.

(3) 자동 확장이 지나치게 되면 어떤가?

기본적으로 자동 확장(AUTOEXTEND구)에 의지하는 것은 좋지 않다.자동 확장을 ON로 하면, 파일의 레벨로 단편화가 발생할 가능성이 있고, 어떠한 SQL 처리중의 데이터 파일의 확장은 부하가 높은 처리가 된다. 데이터 파일로서 필요한 용량을 제대로 추측해, 자동 확장이 발생하지 않도록 한다.단지, 산정을 제대로 한 다음 보험용으로서 넉넉한 값을 설정해 두는 것도 나쁜 일 것은 아니다.

◆ 제3장 시스템 테이블 스페이스의 설계

데이터 파일의 용량 산정

Partitioning Option를 추가한 Enterpirse Edition에서, DBCA로 표시되는 옵션을 일절 선택하지 않고 인스톨하면 약 200 MB, 반대로 모두 선택해 인스톨 하면 약 400 MB의 시스템 테이블 스페이스 영역이 잡힌다.
오브젝트 정보용의 공간도 확보되고 있기 때문에, 기본적으로는 이러한 용량에 여유 치를 봐두면 좋다.

다만, 다음과 같은 케이스에서는 더욱 용량이 필요하게 되는 일이 있다.

- 감사 로그가 테이블에 쌓이는 경우
- replication를 이용하는 경우
- 오브젝트수가 많아지는 경우(특히 STORED나 파티션 등)

로컬 관리 테이블 스페이스 영역에 지나치게 되면

로컬 관리 테이블 스페이스로 하는 것이 퍼포먼스가 뛰어나다. 한편으로 아래와 같은 제약이 있기 때문에, 제약에 신경이 쓰이는 경우는 딕셔너리 관리로 한다. 덧붙여 로컬 관리의 시스템 테이블 스페이스 영역은 AUTOALLOCATE 지정만 가능한 반면에 UNIFORM 지정은 할 수 없다.

- 디폴트 임시 테이블 스페이스 영역의 지정이 필요
- 시스템 테이블 스페이스를 딕셔너리 관리방식으로 변경할 수 없다.
- 신규 딕셔너리 관리 테이블 스페이스 영역 생성할 수 없다.

◆ 제4장 UNDO 테이블 스페이스의 설계

자동 UNDO 관리와 수동 UNDO 관리의 차이점과 선택 기준

Oracle9*i*으로부터 롤백(rollback)·세그먼트(segment) 전용으로 생성되는 테이블 스페이스를 UNDO 테이블 스페이스라고 부른다. 그리고 롤백(rollback) 세그먼트(segment)의 작성·관리를 Oracle 이 알아서 자동으로 관리해주는 기능도 추가되었다. 이것이 자동 UNDO 관리이다. Oracle8*i*까지 DBA가 알아서 롤백(rollback)·세그먼트(segment)를 생성하는 케이스는 수동 UNDO 관리라고 부른다. Oracle9*i*그렇지만 수동 UNDO 관리를 선택할 수 가 있다. 이것은 초기화 파라미터 UNDO_MANAGEMENT로 결정 되어진다. 디폴트가 MANUAL라면 수동 UNDO 관리, AUTO라면 자동 UNDO 관리가 된다.
자동 UNDO 관리와 수동 UNDO 관리는 인스턴스 기동 시는 어느 한쪽만 이용할 수 있다.그리고 기동 중의 변환도 할 수 없다. 따라서, 어느 쪽을 선택해서 이용할지는 설계의 단계에서 결정할 필요가 있다. 이때 자동 UNDO 관리를 선택하면, 수동 UNDO 관리에 비해 이하와 같은 장점을 얻을 수 있다.

(1) 설계·관리가 용이

설계나 관리가 어려운 롤백(rollback)·세그먼트(segment)에 대해 거의 아무것도 고려하지 않아도 괜찮다고 하는 것은, DBA에 있어서 매우 큰 장점이다. 자동 UNDO로 설계할 때 단지 설정하는 것은 테이블 스페이스의 이름과 데이터 파일의 용량, 초기화 파라미터 UNDO_RETENTION의 값, 초기에 작성

하는 UNDO 세그먼트(segment)의 수(기본은 Oracle이 자동 계산) 뿐이다.

(2) ORA-01555 에러 발생 가능성을 방지

어느 정도 경험이 있는 DBA분이라면, 한 번은 대량 갱신 처리에 있어서의 ORA-01555의 발생에 골치를 썩였던 적이 있을까 생각된다. 대량 갱신 처리에 있어서의 ORA-01555는 읽기 일관성을 위해 확보된 UNDO 세그먼트(segment)상의 데이터가 다른 트랜잭션(transaction)에서 OVERWRITE 되고 나서, 그 데이터에 대한 참조 요구가 왔을 때에 발생한다. 자동 UNDO 관리로 하면 초기화 파라미터 UNDO_RETENTION로 설정한 시간동안 UNDO 데이터가 OVERWRITE되지 않고 남기 때문에, ORA-01555에러 발생을 막을 수 있다.

(3) 영역 부족 에러 발생 가능성 방지

자동 UNDO 관리로 하면, A라는 UNDO 세그먼트(segment)가 UNDO 테이블 스페이스의 나머지 영역을 다 사용해 테이블 스페이스의 확장도 할 수 없는 경우, 다른 UNDO 세그먼트(segment)에 빈 곳이 있으면 그 빈 곳을 미사용 영역으로서 인식해서, 처리를 계속한다.수동 UNDO 관리로 같은 상황에 빠졌을 경우는 다른 롤백(rollback)·세그먼트(segment)에 빈 영역이 있었다고 해도 영역 부족 에러가 발생하게 된다.

(4) 플래시백·쿼리가 보다 확실히 이용이 가능하다

플래시백·쿼리란, 커밋된 과거의 데이터를 거슬러 올라가 볼 수 있는 Oracle9*i*신기능이다. 잘못해서 데이터를 갱신해서 커밋해 버렸을 경우의 리커버리에 편리한 기능이다. 수동 UNDO 관리에서도 이 기능은 이용할 수 있지만, 과거의 데이터 보존 기간이 보증되지 않는다. 자동 UNDO 관리로 하면 초기화 파라미터 UNDO_RETENTION를 지정할 수 있게 되어, 이 파라미터로 설정한 시간 내는 UNDO 데이터가 보존이 가능하다.

위와 같은 장점이 있으므로, 기본적으로는 자동 UNDO 관리가 이용되는 것을 추천한다. 한편, 관리하는 요소가 없다고 하는 것은 반대로 말하면 퍼포먼스·튜닝의 여지가 없다고 하는 것이 된다. 시스템의 사정에 맞추어 퍼포먼스를 최대한으로 끌어올릴 수 있는 것은 수동 UNDO 관리이다. 예를 들면 동시 트랜잭션(transaction)수가 많기 때문에 많은 롤백(rollback)·세그먼트(segment)를 사용하는, 대량 배치용으로 거대 사이즈의 롤백(rollback)·세그먼트(segment)를 사용하는 것이 이와 같은 케이스이다. 퍼포먼스를 엄격하게 강화할 경우 수동 UNDO 관리로 사용하는 것도 좋다.

데이터 파일의 용량 산정

자동 UNDO 관리의 경우, UNDO 세그먼트(segment)를 자동으로 작성하기 때문에, UNDO 세그먼트

(segment)의 크기에 관한 산정이나 설정은 불필요하다.데이터 파일의 용량만 추측해 주면 된다.산정식은 이하와 같다.

(A × B ＋ 64 ÷ C ＋ D × 2) × C ＋ 여유치(단위:킬로바이트)

A : UNDO_RETENTION의 값(0의 경우는 1)

B : V$UNDOSTAT 뷰의 UNDOBLKS의 값을 600으로 제산한 값

제일 갱신이 많은 시간대의 레코드를 바탕으로 하는 것

사전 산정의 경우는 1초 당의 발생 UNDO 블록수를 가정

C : 블록 사이즈(2/4/8/16/32의 어느쪽이든 상관없음)

D : V$UNDOSTAT 뷰의 MAXCONCURRENCY의 값

사전 산정의 경우는 최대 동시 트랜잭션(transaction)수를 가정

초기화 파라미터 UNDO_TABELSPACE의 설정

이 파라미터에는, UNDO 테이블 스페이스 영역으로서 이용하고 싶은 테이블 스페이스 영역의 이름을 지정한다.지정하지 않는 경우 시스템·롤백(rollback)·세그먼트(segment)가 UNDO 영역으로서 이용될 가능성이 있으므로, 자동 UNDO 관리의 경우는 반드시 지정하도록 한다..

초기화 파라미터 UNDO_RETENTION의 설정

UNDO_RETENTION의 값은 시스템의 요구에 따라 결정한다. 크게 하는 경우는 확실히 좀 더 긴 과거의 UNDO 정보를 남길 수 있고, 남기고 싶은 양에 따라 UNDO표 영역의 크기를 크게 설계할 필요가 있다. UNDO_RETENTION의 디폴트 값은 900(초)입니다.또, 이 파라미터는 아래와 같이 ALTER SYSTEM문으로 변경이 가능하다.

```
SQL> alter system set undo_retention = 1200;
```

초기 UNDO 세그먼트(segment)의 수

초기 UNDO 세그먼트(segment)의 수는 초기화 파라미터 SESSIONS수를 참조하고, 아래의 계산식에서 결정된다. 초기치는 2~10의 범위로 설정된다.기본적으로 Oracle 에 맡겨도 상관없지만, 기동 직후부터 대량의 동시 트랜잭션(transaction)가 발생하는 것을 알 수 있고 있는 경우는 SESSIONS의 값을 크게 설정한다.SESSIONS의 값이 46이상이면, 초기 UNDO 세그먼트(segment)의 수는 최대한 10이 된다.

least(greatest(1.1 * SESSIONS / 5, 2), 10) (개)
* least는 리스트중의 제일 작은 값을, greatest는 리스트중의 제일 큰 값을 돌려준다

◆ 제5 임시 테이블 스페이스의 설계

임시 파일을 이용한 전용 임시 테이블 스페이스

Oracle9*i*에서는 임시 파일을 이용한 전용 임시 테이블 스페이스를 작성하는 것을 추천 한다. 다른 타입의 임시 테이블 스페이스는, 퍼포먼스 면에서도 설계면 에서도 장점이 없다.굳이 말하자면 구 버젼으로의 임시 테이블 스페이스 작성 스크립트를 그대로 이용할 수 있다는 정도밖에 없다. 임시 파일을 이용한 전용 임시 테이블 스페이스의 장점는 아래와 같다.

(1) 퍼포먼스가 좋다

임시 파일을 이용한 전용 임시 테이블 스페이스는 REDO 로그를 생성하지 않기 때문에, 다른 타입의 임시 테이블 스페이스에 비해 매우 퍼포먼스가 뛰어나다.

(2) 향상된 속도의 백업· 리커버리가 가능하다

임시 파일을 이용한 전용 임시 테이블 스페이스는 백업의 대상이 아니다. 따라서 그만큼 백업과 리커버리가 빨라진다

extent·사이즈의 견적

extent·사이즈는 초기화 파라미터 SORT_AREA_SIZE의 배수에 블록·사이즈를 더한 값이 I/O효율이 좋아진

다.어느 정도의 배수로 하면 좋은가는, OLTP 처리 중심이면 수MB, DSS계 처리 중심이면 수십~수백 MB의 extent·사이즈로 검토한다. 처리 내용이 다르고 extent·사이즈가 다른 복수의 임시 테이블 스페이스를 작성해 구분하여 사용하는 것도 하나의 방편이다 또, 임시 테이블 스페이스를 사용하는 트랜잭션(transaction)가 동시에 발생하는 경우, 1개의 extent를 복수의 트랜잭션(transaction)이 공유할 수 없기 때문에, extent가 너무 크면 영역의 실패가 발생하기 쉬워진다.현재 확보하고 있는 extent수이상의 동시 트랜잭션(transaction)이 발생하면, 역시 extent의 확장이 발생한다.

데이터 파일의 용량 산정

임시 테이블 스페이스는 주로 메모리로 처리할 수 없는 정렬을 위한 영역으로서 사용된다. 어떤 A 소트에 필요한 영역은 최대로 소트 대상 데이터량의 2배 정도이다. 가장 큰 소트 처리의 대상이 되는 데이터 용량의 2배 정도로 여유 치를 더한 용량을 확보한다. 만약 그러한 처리가 동시에 복수 세션으로 처리된다면, 당연히 그 만큼을 확보해 둘 필요가 있다.사전 산정을 예측하지 못할 경우는 우선의 지침으로서 제일 큰 테이블의 용량의 2배 정도를 확보해 두면 좋을 것이다(실제로는 상당한 많은 산정 값이 되기 쉽다).

또, Oracle8*i*로부터 이용 가능한 임시 테이블은, 임시 테이블 스페이스에 생성된다. 임시 테이블 스페이스를 많이 사용하는 되고, 특히 큰 임시 테이블을 많이 만드는 경우는 그만큼을 산정한다. 임시 테이블 자체의 산정은 통상의 테이블과 같게 추측한다.

디폴트 임시 테이블 스페이스의 권장

Oracle9*i*보다, CREATE DATABASE문을 실행할 때에 이하의 예문과 같이 디폴트의 임시 테이블 스페이스 영역을 생성할 수 있다.

SQL> create database ..... *default temporary tablespace 임시테이블스페이스명* tempfile ....

이 지정이 없고, CREATE USER문으로 디폴트의 임시 테이블 스페이스를 지정하지 않으면 시스템 테이블 스페이스 영역이 디폴트의 임시 테이블 스페이스 영역이 되어 버린다.시스템 테이블 스페이스에 임시 세그먼트(segment)를 작성하면, 퍼포먼스에 심각한 영향을 주므로 이 지정은 반드시 하도록 한다.

◆ 제6장 데이터 및 인덱스용 테이블 스페이스의 설계

UNIFORM 지정시의 Tips

extent 관리 방식으로 UNIFORM 옵션의 로컬 관리 테이블 스페이스를 지정했을 경우, 크기가 닮은 테이블마다, 혹은 인덱스마다 같은 테이블 스페이스에 정리하면, 영역의 손실이 적게 된다.

데이터 파일의 용량 산정의 보충

테이블의 이동(ALTER TABLE MOVE)이나 인덱스의 재작성(ALTER INDEX REBUILD) 등을 실시하는 경우, 생성 전의 테이블 스페이스에 임시 세그먼트(segment)를 작성한다. 특히 원래의 세그먼트(segment)와 같은 테이블 스페이스에 이러한 처리를 실시하는 경우, 해당 테이블 영역에 SQL 커멘드 실시 후에 작성되는 세그먼트(segment)의 크기와 동일한 스페이스를 여분으로 필요로 한다.

◆ 제7장 제 1부· 제2부의 설명을 보충한 SQL문 샘플

이 장의 목적

본 장에서는 위의 장들을 해설한 내용을 바탕으로, 실제의 CREATE DATABASE문의 샘플을 제시합니다.

- Windows계 OS의 Oracle9*i*R2를 전제로 하고 있다 .패스명을 고치면 UNIX/Linux계 OS의 Oracle9*i*R2에서도 이용할 수 있다.
- 실제의 환경에서 이용하기 위해서는, 파라미터의 상세한 값의 재검토 , 본SQL를 실행하는 전후의 작업(환경 변수의 설정, catalog.sql의 실행등)이 필요하다.
- 본 스크립트는 필자의 PC로 동작 확인을 취하고 있지만 .디스크의 사정으로 전파일과도 같은 드라이브에 작성한다.
- OMF(Oracle Managed Files)를 이용하지 않는 케이스, 사용한 케이스의 2 종류를 게재하고 있다.

OMF를 이용하지 않는 케이스

(1) 초기화 파라미터(관련 부분만)

- db_name=case1

- undo_management=AUTO
- undo_tablespace=UNDOTBS1
- control_files=("C:₩oracle₩oradata₩case1₩CONTROL01.CTL",
  "C:₩oracle₩oradata₩case1₩CONTROL02.CTL",
  "C:₩oracle₩oradata₩case1₩CONTROL03.CTL")

(2) CREATE DATABASE문

행 번호	CREATE DATABASE문
1	CREATE DATABASE case1
2	USER SYS IDENTIFIED BY TEST123
3	USER SYSTEM IDENTIFIED BY TEST456
4	MAXINSTANCES 1
5	MAXLOGFILES 10
6	MAXLOGMEMBERS 3
7	MAXDATAFILES 100
8	DATAFILE 'C:₩oracle₩oradata₩case1₩system01.dbf' SIZE 250M REUSE
9	EXTENT MANAGEMENT LOCAL
10	DEFAULT TEMPORARY TABLESPACE TEMP TEMPFILE
11	'C:₩oracle₩oradata₩case1₩temp01.dbf' SIZE 40M REUSE
12	UNDO TABLESPACE UNDOTBS1 DATAFILE
13	'C:₩oracle₩oradata₩case1₩undotbs01.dbf' SIZE 200M REUSE
14	CHARACTER SET KO16MSWIN949
15	NATIONAL CHARACTER SET AL16UTF16
16	LOGFILE GROUP 1 ('C:₩oracle₩oradata₩case1₩redo01_1.log',
17	'C:₩oracle₩oradata₩case1₩redo01_2.log') SIZE 102400K,
18	GROUP 2 ('C:₩oracle₩oradata₩case1₩redo02_1.log',
19	'C:₩oracle₩oradata₩case1₩redo02_2.log') SIZE 102400K,
20	GROUP 3 ('C:₩oracle₩oradata₩case1₩redo03_1.log',
21	'C:₩oracle₩oradata₩case1₩redo03_2.log') SIZE 102400K;

(3) CREATE DATABASE문의 해설

- 2~3행째

  유저 SYS 및 SYSTEM에 디폴트가 아닌 패스워드를 설정해 있습니다. 시큐러티를 높이는 관점에

서, 디폴트가 아닌 패스워드를 설정하는 것을 추천한다.

- 8~9행째

  시스템 테이블 스페이스는 로컬 관리 테이블 스페이스로서 작성된다.

- 10~11행째

  디폴트의 임시 테이블 스페이스를 설정해 있다

- 13행목

  DB의 캐릭터·셋로서 KO16MSWIN949를 지정하고 있다.

- 16~21행째

  REDO 로그는 그룹을 3개, 멤버를 2개 작성한다.

OMF를 이용한 케이스

---

(1) 초기화 파라미터(관련 부분만)

- db_name=case2
- undo_management=AUTO
- undo_tablespace=UNDOTBS1
- db_create_file_dest=C:₩oracle₩oradata₩case5
- db_create_online_log_dest_1=C:₩oracle₩oradata₩case5₩log1
- db_create_online_log_dest_2=C:₩oracle₩oradata₩case5₩log2
- control_files→

  OMF 에서 하는 경우는 컨트롤 파일을 설정하지 않는다.

  CREATE DATABASE 실행시에 SPFILE를 사용하는 경우는, 작성된 제어 파일이 자동적으로 본 파라미터에 등록된다.

  한편 PFILE를 사용하는 경우는, CREATE DATABASE 실시 후에 PFILE에 본 파라미터를 추가할 필요가 있다. 엔트리에 기술해야 할 제어 파일의 명칭은, 초기화 파라미터 DB_CREATE_ONLINE_LOG_DEST_*n*,혹 은 CREATE_FILE_DEST(DB_CREATE_ONLINE_LOG_DEST_*n*를 지정하고 있지 않는 경우)로 지정한 디렉토리에 확장자(extension).ctl로 작성된다.

(2) CREATE DATABASE문

행 번호	CREATE DATABASE문

```
3  USER SYSTEM IDENTIFIED BY TEST456
4  MAXINSTANCES 1
5  MAXLOGFILES 10
6  MAXLOGMEMBERS 3
7  MAXDATAFILES 100
8  DATAFILE SIZE 250M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
9  EXTENT MANAGEMENT LOCAL
10 DEFAULT TEMPORARY TABLESPACE TEMP TEMPFILE SIZE 40M
11 AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
12 UNDO TABLESPACE UNDOTBS1 DATAFILE SIZE 200M
13 AUTOEXTEND ON NEXT 5M MAXSIZE UNLIMITED
14 CHARACTER SET KO16MSWIN949
15 NATIONAL CHARACTER SET AL16UTF16
16 LOGFILE GROUP 1 SIZE 102400K,
17 GROUP 2 SIZE 102400K,
18 GROUP 3 SIZE 102400K;
```

(3) CREATE DATABASE문의 해설

- 전반
  - OMF를 이용하고 있으므로 파일명을 지정해 있지 않다.
  - 파일명 이외의 파라미터(파일 사이즈등)도 지정을 줄일 수 있지만, 본 스크립트에서는 값을 설정하고 있다.
  - 본 스크립트에서는 데이터 파일의 자동 확장을 지정해 있다.
- 8~13행째

  시스템 테이블 스페이스, 임시 테이블 스페이스, undo 테이블 스페이스의 데이터 파일은 초기화 파라미터 DB_CREATE_FILE_DEST에서 지정한 디렉토리에 작성된다.
- 16~18행째

  REDO 로그 파일은 초기화 파라미터 DB_CREATE_ONLINE_LOG_DEST_1및 DB_CREATE_ONLINE_LOG_DEST_2에서 지정한 디렉토리에 멤버가 분산되어 작성된다.그룹은 각 디렉토리에 3개씩 작성된다.

제 3 부 : 테이블의 설계

◆ 제1장 테이블 용량의 산정

처음

여기서 설명하고 있는 산정 방법은, 자료 「영역 사이즈의 견적 방법」의 생각을 약간 간략화하고, 계산하기 쉽게 한 방법이다. 또, 지침을 내는 것을 우선해, 플랫폼이나 버젼에 의한 차이를 시작으로 세세한 요소를 생략하고 있으므로, 여기서 해설하고 있는 견적 방법은 완벽하게 정확하지는 않다. 그렇다고 해도, 오차는 실제의 필요 용량이 보다 많은 경우에서도 산정 후에 예상 할 여유 치 만큼 허용할 수 있을 정도이다. 또, 본견적인 산정 방법은 다음의 경우에는 산정 방법을 지원하지 않는다.

- 블록 사이즈보다 긴 레코드길이의 테이블의 산정
- LOB에 관한 산정

테이블의 견적 방법 개요

테이블의 산정 순서는, 대략적으로는 다음의 순서가 된다.

(순서 1) 1개의 레코드의 평균 길이을 요구한다
(순서 2) 1개의 블록에 들어가는 레코드 수를 요구한다
(순서 3) CEIL(예상 레코드 수 ÷ 순서 2의 값) × 블록 사이즈 = 테이블 용량

※CEIL는 지정된 수보다 크거나 같은 최소 정수값을 반환하는 함수

1개의 레코드의 평균길이를 구하는 방법

레코드는 대략적으로는 그림 1과 같은 방식로 저장된다.레코드는 레코드 헤더가 하나와 각 열의 헤더, 및 열 데이터로 구성된다.

그림 1 : 레코드의 구조

레코드 헤더	열 헤더	열 데이터	열 헤더	열 데이터	열 헤더	열 데이터

(1) 레코드 헤더의 사이즈

레코드 헤더의 사이즈는 3바이트이다.

(2) 열 헤더의 사이즈

대응하는 열의 데이터 길이가 250바이트 이하(NULL 포함)의 경우는 1바이트, 251바이트 이상의 경우는 3바이트이다.

(3) 열 데이터의 사이즈

데이터형에 따라서 다르다. 주요 데이터형에 대해서는 다음의 표 1을 참조한다. 데이터가 NULL의 경우는 어떤 데이터형도 0바이트이다.

표 1 : 데이터형에 의한 실 점유 사이즈

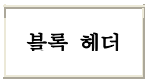
데이터형	고정/가변	데이터가 저장되었을 때의 길이
CHAR	고정/가변	• 바이트수지정시는 테이블 정의의 길이의 고정 길이 • 문자수지정시는 문자 수에 대해서 고정이지만, 실점유 바이트 수는 캐릭터·셋 및 실제의 데이터에 의해 다르다. 예를 들면 KO16KSC5601의 DB로 「char(6 char)」의 컬럼에 「AAA」라고 하는 데이터를 넣으면 3바이트 소비되지만, 「아아아」라고 하는 데이터를 넣으면 6바이트 소비한다.
VARCHAR2	가변	실제로 저장되고 있는 데이터의 길이(바이트 수 지정시· 문자수지정시)
NCHAR	고정	테이블 정의 문자수의 2배(AL16UTF16 지정시)
NVARCHAR2	가변	저장 문자수의 2배(AL16UTF16 지정시)
NUMBER	가변	길이 = 1 + CEIL (n / 2) • CEIL는 가장 가까운 정수값을 반환 • n는 저장된 수치의 정수부·소수부를 합한 총자리수. n>38의 경우에는 38 이다 • 유효 자리수 38자리수 미만의 음수의 경우는 1바이트 가산
DATE	고정	7바이트
TIMESTAMP	가변	초의 소수부분에 데이터가 있는 경우 : 11바이트 고정 초의 소수부분에 데이터가 없는 경우 : 7바이트 고정

TIMESTAMP WITH TIME ZONE	고정	13바이트
TIMESTAMP WITH LOCAL TIME ZONE	고정	11바이트
INTERVAL YEAR TO MONTH	고정	5바이트
INTERVAL DAY TO SECOND	고정	11바이트
RAW	가변	실제로 저장되고 있는 데이터의 길이
LONG	가변	실제로 저장되고 있는 데이터의 길이
LONG RAW	가변	실제로 저장되고 있는 데이터의 길이
BLOB/CLOB/NCLOB	지정에 의한	• DISABLE IN ROW 지정시 : 20바이트 • ENABLE IN ROW 지정으로 행 내에 저장시 : 실데이터 길이+ 36바이트 • ENABLE IN ROW 지정으로 행 이외(LOB 테이블)에 저장시 : 실데이터 길이에 의해 36~86바이트.산정 시는 여유 있게 86바이트 고정으로 계산한다. • LOB 테이블의 산정에 대해서는 나중에 기술 • EMPTY의 경우는 데이터 길이의 부분을 0으로 계산
BFILE	고정	530바이트
ROWID	고정	6바이트(~Oracle7) 10바이트(Oracle8~)

1개의 블록에 들어가는 레코드수의 계산하는 법

DB블록의 구조는 대략적으로는 그림 2와 같다. 블록은 블록 헤더와 데이터 저장부, 그리고 테이블 파라미터 PCTFREE의 지정으로 확보한 예비 영역으로 구성됩니다.

그림 2 : 블록의 구조

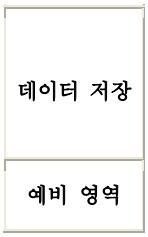


(1) 블록 헤더의 사이즈

블록 헤더의 사이즈는 아래의 계산식보다 구할 수 있다. INITRANS는 테이블

파라미터이다.

헤더의 사이즈 = 90 + (INITRANS − 1)



(2) 예비 영역의 사이즈

예비 영역의 사이즈는 이하의 계산식보다 구할 수 있다. PCTFREE는 테이블 파라미터이다. PCTFREE는 단위가 % 이므로, 계산 때는 소수로 고쳐서 계산한다.

예비 영역의 사이즈 = CEIL((테이블 저장전의 테이블스페이스의 블록 사이즈 − 헤더의 사이즈) × PCTFREE)

(3) 데이터 저장부분의 사이즈

데이터 저장부분의 사이즈는 다음의 계산식으로 구할 수 있다.

데이터 저장부의 사이즈 = 테이블 저장전의 테이블 스페이스의 블록 사이즈-헤더의 사이즈-예비 영역의 사이즈

이를 개선해서

TRUNC(데이터 저장부의 사이즈 ÷ 평균 레코드 길이)

가 1개의 블록에 들어가는 레코드수가 된다.

평균 레코드 길이가 데이터 저장부보다 긴 경우의 산정 방법

평균 레코드길이가 데이터 저장부분보다 긴 경우, 당연히 레코드는 복수의 블록에 걸쳐 저장된다. 이 상태를 로우 체인현상이라 부른다. 로우 체인현상이 발생하는 평균 레코드길이의 경우의 테이블 용량의 산정은,

CEIL(평균 레코드길이 ÷ 데이터 저장부분의 길이) × 산정 레코드 수 × 블록 사이즈

가 된다.데이터 저장부분의 계산 시에는, PCTFREE는 실제의 지정에 관련되지 않고 0으로서 계산해 주면 된다.또, 이 산정 방법은 레코드의 사이즈의 격차가 적은 경우는 유효하지만, 불규칙한 경우는 실제의 레코드의 저장순서에 실제저장 사이즈가 큰 폭으로 영향을 받기 때문에, 산정 값이 크게 나올 가능성이 있다.

LOB 테이블의 산정 방법

LOB 테이블의 산정은 아래의 순서와 같다.

(순서 1) CHUNK의 값을 블록 사이즈의 배수 값에 반올림한다

LOB 테이블의 파라미터 CHUNK는 LOB의 I/O사이즈가 된다.하지만, CHUNK의 사이즈가 블록 사이즈의 배수가 아닌 경우는, CHUNK의 사이즈를 초과한다. 제일 작은 블록 사이즈의 배수의 값이 실제의 I/O사이즈가 된다. 예를 들면 CHUNK가 5 KB, 블록 사이즈가 4 KB의 경우는, 실 I/O사이즈는 8 KB가 되지만. 이 경우 딕셔너리 상에 5 KB는 아니고 8 KB가 되게 된다.

(순서 2) 평균 LOB 길이를 순서 1의 배수값에 반올림한다

LOB의 데이터는 순서 1으로 요구한 CHUNK 사이즈의 배수 값에 내림하여 저장된다. 예를 들면 평균 LOB길이가 28 KB, CHUNK의 사이즈가 8 KB의 경우는 32 KB가 된다.

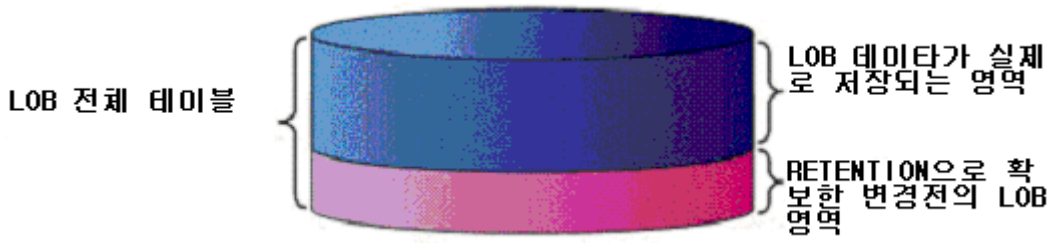
(순서 3) LOB 저장 영역의 사이즈를 구한다.

LOB 저장 공간의 사이즈는 아래의 계산식으로 구할 수 있다.

LOB 저장 공간의 사이즈 = 순서 2의 값 × 예상 레코드 수

(순서 4) RETENTION(PCTVERSION) 영역을 구한다.

LOB 테이블에 대해서 갱신을 실행했을 경우, 갱신 전 데이터는 UNDO 세그먼트(segment)가 아니고, LOB 테이블 자신에게 저장된다.따라서, 산정 시에 이 부분을 예측 할 필요가 있다.자동 UNDO 관리를 이용하고 있는 경우는 파라미터 RETENTION, 수동 UNDO 관리를 이용하고 있는 경우는 파라미터 PCTVERSION로 지정한 영역이 LOB의 UNDO를 위하여 확보된다. 시스템 중에 트랜잭션(transaction)이 최대한 발생할 수 있는 변경 전 LOB 데이터가 저장 가능한 크기로 예상한다. 잘 모르는 경우는 순서 3에서 구한 용량의 20% 정도로 산정한다.



(순서 5) 순서3과 순서4의 합계가 LOB 테이블의 용량이 된다.

◆ 제2장 테이블의 산정예제

레코드가 한 개의 블록에 들어가는 길이의 경우의 산정예

SCOTT schema에 있는 EMP 테이블을 예로 산정을 실제로 해보자. 예상 건수는 10,000건으로 한다.EMP테이블

의 구조는 다음과 같다.

```
SQL> desc emp
 컬럼명                                    NULL 유무      데이터타입
 ----------------------------------------- -------- ------------------------
 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                              VARCHAR2(10)
 JOB                                                VARCHAR2(9)
 MGR                                                NUMBER(4)
 HIREDATE                                           DATE
 SAL                                                NUMBER(7,2)
 COMM                                               NUMBER(7,2)
 DEPTNO                                             NUMBER(2)
```

(1) 평균 레코드길이를 구하자

표 3 : 각 열의 길이의 계산예

열명	데이터형	열데이터가 길이	열헤더가 길이	합계열장
EMPNO	NUMBER(4)	1 + CEIL (4 / 2) = 3바이트	1바이트	4바이트
ENAME	VARCHAR2(10)	CEIL (10 * 0.7) = 7바이트	1바이트	8바이트
JOB	VARCHAR2(9)	CEIL (9 * 0.7) = 7바이트	1바이트	8바이트
MGR	NUMBER(4)	1 + CEIL (4 / 2) = 3바이트	1바이트	4바이트
HIREDATE	DATE	7바이트	1바이트	8바이트
SAL	NUMBER(7,2)	1 + CEIL (7 / 2) = 5바이트	1바이트	6바이트
COMM	NUMBER(7,2)	1 + CEIL (7 / 2) = 5바이트	1바이트	6바이트
DEPTNO	NUMBER(2)	1 + CEIL (2 / 2) = 3바이트	1바이트	3바이트
레코드길이 소계				47바이트
레코드길이(열 헤더 3바이트를 가산)				**50바이트**

(2) 한 개 블록에 들어가는 레코드 수를 구하자

필요한 테이블 파라미터는 모두 디폴트(INITRANS:1, PCTFREE:10)으로 한다.블록 사이즈는 8 KB(8,192바이

트)로 한다.

표 4 : 블록의 빈 영역을 구하자

요구하는 대상	계산 내용
블록 헤더의 사이즈	86 + (INITRANS − 1) = 90 + (1 − 1) = 90바이트
예비 영역의 사이즈	CEIL((테이블 저장부의 테이블 스페이스의 블록 사이즈 − 헤더의 사이즈) × PCTFREE) = CEIL((8,192 - 90) * 0.1) = 811바이트
데이터 저장영역의 사이즈	테이블 저장전의 테이블 스페이스 블록 사이즈 - 헤더의 사이즈 - 예비 영역의 사이즈 = 8,192 - 90 - 811 = **7,291바이트**

한 개 블록 중에서 실제로 데이터를 저장 할 수 있는 영역이 7,291바이트가 구해졌으므로, 이것을(1)에서 요구한 평균 레코드길이 50바이트로 나누는(소수점은 버림)것과 한 개 블록에 146건 들어가는 것을 알 수 있다.

(3) 테이블 용량을 구하자

(2)에서부터 테이블 용량은

예상 레코드 수÷(2)의 값 × 블록 사이즈

가 된다. 실제로 계산하면,

CEIL(10,000 / 146) * 8,192 = 565,248바이트 = **552킬로바이트**

가 된다.

레코드가 한 개 블록에 들어가지 않는 길이의 경우의 산정

블록 사이즈 8 KB에 대해서 VARACHAR2(3000)의 3개 열의 레코드가 10,000건 삽입되는 테이블의 용량을 추측한다. 데이터는 어느 열도 항상 3,000바이트 들어오는 것으로 가정하자.

(1) 평균 레코드 길이를 구하자

레코드 길이는(3,000 + 3(열헤더)) * 3(열) + 3(레코드 헤더) = 9,012바이트가 된다.

(2) 데이터 저장부분의 사이즈를 구한다

레코드가 블록에 들어간다면, 7,291바이트가 된다.

(3) 테이블 용량을 구하자

테이블 용량의 계산방법은

CEIL(평균 레코드 길이 ÷ 데이터 저장부분의 길이) × 예상 레코드 수 × 블록 사이즈

이므로,

CEIL(9,012 / 7,291) * 10,000 * 8,192 = 163,840,000바이트 = **약 164 메가바이트**

가 됩니다.

LOB 테이블의 견적

블록 사이즈 8 KB에 대해서 평균 1 MB의 LOB를 1000건 저장하는 LOB 테이블의 산정 예이다. RETENTION로 필요한 영역은 LOB 저장 영역의20%, DISABLE STORAGE IN ROW 지정으로 1000건 모두가 LOB 테이블에 저장되는 것으로 가정하자. CHUNK는 3 KB로 한다.

(1) CHUNK의 값을 블록 사이즈의 배수에 반올림한다

CHUNK가 3 KB로 블록 사이즈가 8 KB이므로, 8 KB가 된다

(2) 평균 LOB길이를 (1)의 배수에 반올림한다

1 MB는 8 KB의 배수이므로, 1 MB 그대로이다.

(3) LOB 저장 영역의 사이즈를 구하자

LOB 저장 영역의 사이즈는

LOB 저장 영역 = (2)의 값 × 예상 레코드 수

이므로,

1,048,576 * 1,000 = 1,048,576,000바이트 = 1,000 메가바이트

가 됩니다.

(4) RETENTION 영역을 구해서 더한 테이블 용량으로 하자.

RETENTION 영역은 LOB 저장 영역의 20%이므로, (3)의 값을 1.2배가 되었다 **1,200 메가바이트(MB)**가 산정 결과가 된다.

◆ 제3장 열의 나열 순서에 대한 지침

열(컬럼)의 나열 순서의 기본 지침

열의 나열 순서는 기본적으로 관리가 하기 쉬운 것을 염두에 두어 설계한다. 예를 들면 아래와 같은 점에 유의하자.

- 주키(PK)를 처음에 정의하자
- 관련성이 있는 열(예 : 성과 이름, 부 코드와 과 코드 등)은 연속 적으로 정의한다
- 자주 이용되는 열을 앞쪽에 배치하자

레코드 길이을 짧게 하는 테크닉

레코드 길이을 짧게 할 수 있으면 있을수록, 블록에 저장할 수 있는 레코드 수가 많아진다. 그리고, 캐쉬의 히트율 또한 높아지고, 용량의 절약도 가능하게 된다.

(1) 가변 길이 데이터형의 이용

예를 들면 우편번호와 같이 자리수가 정해져 있는 데이터를 저장하는 열을 제외하고, VARCHAR2와 같은 가변길이을 이용하도록 하자. 정의된 길이에 대해 실 데이터가 짧으면 짧을수록 용량의 낭비를 막을 수 있다.

(2) NULL값이 저장 되기 쉬운 열을 뒤쪽에 정의하자

NULL가 되는 것이 많은 열은 정리해서 뒤로 배치하면, 열 헤더가 생략 되기 때문에 레코드 길이를 짧게 할 수 있다.

(그림) NULL값의 컬럼을 뒤로 정의한 결과 행의 길이가 짧아졌다.

◆ 제4장 CREATE TABLE의 파라미터의 설정

본 장은 로컬 관리 테이블 스페이스를 이용하고 있는 것을 전제로 진행한다.

로컬 관리와 딕셔너리 관리로 의미가 다른 파라미터

이하의 파라미터는 로컬 관리와 딕셔너리 관리로 의미가 다르다.

- INITIAL
- NEXT
- PCTINCREASE
- MAXEXTENTS

표 5에서 차이를 설명하고 있다.로컬 관리 테이블 스페이스에 있어 파라미터의 의미가 크게 다른 것은, 크기를 지정하는 파라미터(INITIAL/NEXT)가, extent의 사이즈를 나타내는 것이 아니라, 초기에 확보하는 총테이블 용량이 되는 것이다.

표 5 : 로컬 관리와 딕셔너리 관리로 의미가 달라지는 파라미터

파라미터	딕셔내리 관리	로컬 관리
INITIAL	초기에 확보하는 extent의 사이즈.예를 들면 100 MB의 INITIAL를 지정하면, 1개의	초기에 확보하는 extent의 총사이즈.예를 들면 1 MB의 UNIFORM으로 지정된 테이블 스페이
NEXT	두번째의 extent의 사이즈. 예를 들면 70 MB의 NEXT를 지정하면, 두번째의 extent	MINEXTENTS가 2개 이상일때에 INITIAL와 NEXT의 합계의 사이즈가 되도록 하나 이상의
PCTINCREASE	세번째 이후의 extent를 확보할 때의, 전에 할당 받은 extent 사이즈로부터의 증분.디	MINEXTENTS가 3개 이상일 때에, 딕셔너리 관리와 같은 계산방법으로 하나 이상의 extent

	의 extent의40% 증가로 196 MB가 된다.	100 + 100 + 140 + 196 = 536개의 1 MB의 extent를 확보된다.
MAXEXTENTS	지정한 수보다 많은 extent를 작성할 수 없게 된다.	의미는 딕셔내리 관리와 같지만, 지정해도 무시되어 항상 UNLIMITED가 된다.

테이블 작성시로 지정을 고려해야 할 파라미터

로컬 관리 테이블 스페이스에 있어서는, 항상 지정해야 할 파라미터는 기본적으로 INITIAL만이라고 생각하는 것이 일반적이지만 . 물론 이것은 틀린 말이 아니다. 상당히 처리량이 많은 경우를 제외하고는 어느 정도 다른 파라미터를 고려하지 않아도 상관이 없다

INITIAL는 가능한 한 예상 레코드 수를 수용 할 만한 크기를 지정한다. 로컬 관리 테이블 스페이스에서 extent의 수가 많은 것 자체는 퍼포먼스에 별로 영향을 주지 않지만, extent의 확장의 경우 딕셔너리 관리 테이블 스페이스 정도는 아니지만, 어느 정도의 부하가 발생한다.

경우에 의해 지정을 고려하는 파라미터

---

(1) NEXT/MINEXTENTS/PCTINCREASE

대규모 테이블의 경우, extent를 복수의 데이터 파일에 분산시켜 I/O성능의 향상을 도모하는 케이스가 있다. 이와 같은 경우에 이 파라미터 사용을 검토한다. 그 때 PCTINCREASE는 디폴트의 0인 편이 영역 계산이 하기 쉽다.

(2) FREELIST GROUPS

디폴트로 1 인 그대로 두어도 상관없다. RAC 환경에서는 노드 수에 맞춘 값을 기본하자.

(3) FREELISTS

디폴트의 1 그대로 두어도 상관없다. 대량 삽입의 트랜잭션(transaction)이 동시에 발생하는 경우는 값을 늘리는 것을 검토하자.

(4) PCTFREE

초기 레코드 사이즈로부터 최종 레코드 사이즈가 길어질 수 있는 비율을 설정하는 것이 제일 효율적인 영역 관리를 할 수 있다. 예를 들면 INSERT시의 레코드의 평균 사이즈가 100바이트로, 갱신을 거듭해 최종적으로 평균 130바이트가 되는 것이면 30(%)을 지정하면 좋다. 레코드 사이즈의 증가율을 알 수 없는 경우는, 디폴트의 10으로 운용해, 재 편성 시에 값을 조정한다. 읽기 전용의 테이블이라면 0으로 해도 상관

없다.

(5) PCTUSED

기본적으로 디폴트의 40으로 상관하지 않습니다.PCTUSED의 값을 비싸게 하면 블록의 재이용이 해져서 싸지므로 영역을 효율적으로 이용할 수 있습니다만, FreeList에 블록이 많이 등록되기 쉽기 때문에 갱신계의 퍼포먼스가 떨어집니다.낮게 하면 퍼포먼스는 오르지만 영역의 이용 효율은 떨어진다. 퍼포먼스를 우선시키는 경우는 보다 낮게, 영역의 이용 효율을 우선시키는 경우는 보다 조금 높게 설정한다. 또, 풀테이블 스캔의 퍼포먼스를 높이고 싶은 경우는 데이터를 채워 저장하는 것이 좋기 때문에 높은 값으로 설정한다. 덧붙여 PCTFREE와 PCTUSED의 값의 합계는 100을 넘을 수 없기 때문에, 이 범위에서 조정하도록 한다.

(6) INITRANS

디폴트의 1인 채로 상관하지 없지만 복수의 트랜잭션(transaction)으로 동시에 같은 블록에의 갱신이 빈번한 경우는, 값을 늘리는 것을 검토하자

(7) LOGGING/NOLOGGING

테이블에 대한 처리로 NOLOGGING 옵션이 효과가 있는 것은 아래의 작업에서 이다. 이러한 처리의 속도를 올리고 싶은 경우는 NOLOGGING의 지정을 검토하자. 다만, REDO 로그에 처리 내용이 기록되지 않기 때문에, 이러한 처리를 실시한 다음은 백업을 하는 것을 권고한다.

- 다이렉트·로드(SQL*Loader)
- 다이렉트·로드·인서트(APPEND 힌트포함의 INSERT SELECT)
- CREATE TABLE ... AS SELECT
- ALTER TABLE ... MOVE PARTITION
- ALTER TABLE ... SPLIT PARTITION
- LOB 테이블에 저장되는 NOCACHE NOLOGGING 모드의 LOB의 INSERT, UPDATE 및 DELETE

제 4 부 : 인덱스의 설계

◆ 제1장 인덱스 용량의 견적

여기서 설명하고 있는 산정 방법은, 앞에서와 같이 자료 「영역 사이즈의 견적 방법」의 방법을 약간 간략화해서, 계산하기 쉽게 한 방법이다. 또, 지침을 내는 것을 우선해서, 플랫폼이나 버젼에 의한 차이를 시작으로 세세한 요소를 생략

하고 있으므로, 여기서 설명하고 있는 산정 방법은 완벽하게 정확하지는 않다.

B-Tree 인덱스의 산정 방법 개요

인덱스의 산정 순서는 대략적으로 아래의 순서가 된다.

(순서 1) 1개의 레코드의 평균 길이을 구한다
(순서 2) 1개의 블록에 들어가는 레코드 수를 구한다
(순서 3) CEIL(예상 레코드수 × 1.05 ÷ 순서 2의 값) × 블록 사이즈가 인덱스 용량이 된다
(순서 4) 빈 테이블에 대해서 인덱스를 작성하는 경우는 순서 2의 값×1.3 이 인덱스 용량이 된다

순서 4가 필요하게 되는 이유는, 순서 3까지 방법으로 구할 수 있는 용량이 이미 예상 건수의 레코드가 저장되고 있는 테이블에 대해서 인덱스를 작성했을 경우의 용량이기 때문이다. 빈 테이블과 인덱스를 작성하고 나서 예상 건수의 레코드를 삽입했을 경우, 어떠한 값을 가지는 레코드가 어떠한 차례로 삽입될지를 사전에 알지 못한다. 그 때문에, 레코드가 이미 존재하고 있는 상태로부터 인덱스를 작성했을 경우에 비하면, 블록 내에서의 레코드의 저장 효율이 떨어져 버린다. 덧붙여 곱해지는 수 1.3은 필자의 경험에 의하는 것이라는 것을 말해둔다. 저장되는 데이터의 특성이나 차례로 따라 더욱 여유공간을 갖는 것이 좋은 케이스가 될지도 모르지만. 1.1 정도의 값이 들어가는 케이스도 있다.

1개 레코드의 평균길이의 구하는 방법

인덱스의 레코드는 대략적으로는 그림 1과 같은 이미지로 저장된다.

그림 1 : B-Tree 인덱스의 레코드의 구조

레코드 헤더	ROWID	오버 헤드	열헤더	열데이터	열헤더	열데이터	열헤더	열데이터

(1) 레코드 헤더의 사이즈

레코드 헤더의 사이즈는 3바이트입니다.

(2) ROWID의 사이즈

Oracle9*i*의 본래의 ROWID의 사이즈는 10바이트지만, 색인 레코드 중의 ROWID는 Oracle7 때의 6바이트의 포맷으로 저장된다. 다만, 글로벌 파티션·인덱스의 경우는 10바이트가 된다.

(3) 오버헤드의 사이즈

(127바이트 이하의 열수 × 1) + (128바이트 이상의 열수 × 2) + 4 바이트가 됩니다. 인덱스의 경우는 1바이트가 더 필요하다.

(4) 열헤더의 사이즈

대응하는 열의 데이터 길이가 250바이트 이하의 경우는 1바이트, 251바이트 이상의 경우는 3바이트이다.

(5) 열 데이터의 사이즈

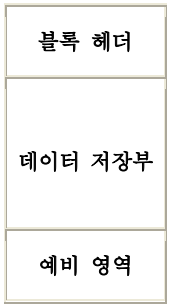
데이터형에 따라서 다르다. 주요 데이터형에 대해서는 앞서 설명한 테이타 형의 표를 참조하자.

주의점으로서(1)+(4)+(5)의 합계(복합 인덱스의 경우는 전 인덱스 대상 열의(4)+(5)가 대상)가 9바이트에 못 미친 경우는 9바이트로 반올림해준다.

1블록에 들어가는 레코드수의 구하는 방법

DB블록의 구조는 대략적으로는 그림 2와 같다. 인덱스에서도 테이블과 다르지 않지만. 블록은 블록 헤더와 데이터 저장부, 그리고 인덱스 파라미터 PCTFREE의 지정으로 확보한 예비 영역으로 구성된다.

그림 2 : 블록의 구조



(1) 블록 헤더의 사이즈

블록 헤더의 사이즈는 아래의 계산식보다 구할 수 있다.INITRANS는 인덱스 파라미터이다.

헤더의 사이즈 = 96 + 24 × INITRANS

(2) 예비 영역의 사이즈

(3) 데이터 저장부분의 사이즈

데이터 저장부분의 사이즈는 아래의 계산식으로부터 구할 수 있다.

데이터 저장부분의 사이즈 = 인덱스 저장 테이블 스페이스의 블록 사이즈-헤더의 사이즈-예비 영역의 사이즈

이상의 값으로부터

TRUNC(데이터 저장부분의 사이즈 ÷ 평균 레코드길이)

가 1개 블록에 들어가는 레코드수가 된다. TRUNC는 소수를 버리는 함수이다.

LOB 인덱스의 산정 방법

Oracle9*i*에서 LOB 인덱스는 LOB 테이블을 생성했을 때에 LOB 테이블과 같은 테이블 스페이스에 Oracle이 내부적으로 생성한다. 따라서 자동으로 생성되기 때문에 용량을 간과하기 쉽지만, LOB 인덱스의 산정도 정확하게 하도록 한다. LOB 인덱스의 용량은 LOB 테이블의 최저5%의 크기를 Oracle이 자동으로 확보한다.그러므로 LOB 인덱스의 용량은 LOB 테이블의 용량의 5%로 추측하면 된다.

비트 맵·인덱스의 산정 방법

비트 맵·인덱스의 산정에 대해서는, 실제로 들어 오는 데이터의 내용이나, 데이터 압축의 상황에 따라 실용량이 좌우되기 때문에, 사전에 추측하는 게 힘들다. 단지, 같은 열에 대해서 작성한 B-Tree 인덱스보다는 작아지므로, 큰 견적이 되어 버려서 B-Tree 인덱스의 산정 법을 이용하자.

◆ 제2장 인덱스의 산정예

B-Tree 인덱스의 산정법

SCOTT schema에 있는 EMP 테이블을 예로, ENAME 열에 B-Tree 인덱스를 작성했을 경우의 산정을 실제로 해보자. 예상 건수는 10,000건으로 한다. EMP테이블의 정의는 아래와 같다.

SQL> desc emp

이름	NULL?	형
EMPNO	NOT NULL	NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(9)
MGR		NUMBER(4)
HIREDATE		DATE
SAL		NUMBER(7,2)
COMM		NUMBER(7,2)
DEPTNO		NUMBER(2)

(1) 평균 레코드길이를 구하자

이번은 ENAME열의 평균열의 길이가 7바이트로 가정해 계산한다. 이것에 열 헤더 사이즈 1바이트를 더한 8바이트가 열의 평균길이가 된다. 레코드 헤더의 사이즈를 더하면 9바이트가 넘기 때문에 , 8바이트의 길이로 계산을 계속하자. 이것에 레코드 헤더 3바이트, ROWID6 바이트, 오버헤드 6바이트(1 + 4 + 1, 최초의 1바이트는 열의 오버헤드, 4는 고정치, 마지막 1은 인덱스의 몫이다)을 더한 23바이트가 평균 레코드길이가 된다.

(2) 1개의 블록에 들어가는 레코드 수를 구하자

필요한 인덱스 파라미터는 모두 디폴트(INITRANS:2, PCTFREE:10)으로 한다.블록 사이즈는 8 KB(8,192바이트)로 한다

표 1 : 블록의 빈 공간을 구하자

요구하는 대상	계산 내용
블록 헤더의 사이즈	96 + 24 × INITRANS = 96 + 24 × 2 = 144바이트
예비 영역의 사이즈	CEIL((인덱스 저장공간의 테이블 스페이스의 블록 사이즈 − 헤더의 사이즈) × PCTFREE) = CEIL((8,192 - 144) * 0.1) = 805바이트
데이터 저장부분의 사이즈	인덱스 저장공간의 테이블 스페이스의 블록 사이즈-헤더의 사이즈-예비 영역의 사이즈 = 8,192 - 144 - 805 = **7,243바이트**

1개 블록 중에서 실제로 데이터를 저장할 수 있는 영역이 7,243바이트가 구해졌으므로, 이것을 (1)에서 구한 평균 레코드 길이 23바이트로 나누면(소수점은 버린다) 1 블록에 314건 들어가는 것을 알 수 있다.

(3) 인덱스 용량을 구하자

인덱스 용량의 계산방법은

CEIL(예상 레코드수÷1 블록에 들어가는 레코드 수)×블록 사이즈

이므로,

제 5 부 : 영역 감시

◆ 제1장 영역 감시의 대상

영역 감시라고 하는 것은 그 목적과 수단에 따라서 여러 가지가 존재한다. 본 장에서는 그 중에서도 이른바 「단편화」라고 하는 키워드에 주목해서, 단편화의 발생을 감시한다고 하는 관점에서 각종 영역 감시의 대상·방법에 대해 설명하기로 한다.감시를 실시하고 문제를 찾아낸 후, 그 문제를 해결하는 방법에 대해서는 6부에서 설명하기로 한다. 이번 장은 SQL가 많이 실행되고 있고, 특별한 기술이 없는 한 SYSTEM 유저로 실행한다. 다른 유저로 실행하는 경우, 검색 대상의 딕셔너리나 권한 등을 부여해 주세요

단편화란?

Oracle에 있어서의 단편화란, Oracle의 물리 영역이 어떠한 형태로 불연속이 되어 있는 상태를 말한다. 단편화가 발생하면, 발생의 정도로 따라 주로 이하와 같은 영향이 나온다.

- 영역이 효율적으로 이용되지 않고, 실 데이터량에 비해 디스크 용량을 많이 차지한다
- I/O 시간이 걸려, 퍼포먼스가 떨어진다

한마디로 단편화 라고 하여, Oracle의 경우 DB의 물리 구조가 계층 구조가 되어 있어, 계층에 따라 나타나는 단편화의 현상에 차이가 있습니다. 표 1으로 DB의 물리 구조에 대응한 단편화 현상을 정리했다. 다음 장 이후에 각각의 단편화 현상에 대하고 설명을 할 것이다.

표 1 : Oracle의 물리 구조와 단편화

계층	단편화 현상
데이터 파일(테이블 스페이스)	·파일 레벨의 단편화 ·데이터 파일 레벨의 미사용 영역의 발생
세그먼트(segment)	·수위가 높은 하이 워터 마크 ·세그먼트(segment) 레벨의 미사용 영역의 발생 ·계층이 깊은 인덱스
extent	·불연속의 extent
블록	·행 이행 ·행 chaining ·블록내의 미사용 영역의 발생

◆ 제2장 데이터 파일(테이블 스페이스) 레벨의 단편화

파일 레벨의 단편화

아무리 블록 레벨이나 세그먼트(segment) 레벨이라고 한 세세한 레벨로 단편화를 해소해도, 데이터 파일의 레벨로 단편화가 발생하고 있으면 의미가 없다. 특히 하나의 디스크에 데이터베이스가 들어가, 초기의 물리 설계의 시스템에 대해서 빈번한 단편화가 발생해, 시스템의 퍼포먼스를 떨어뜨릴 가능성이 있다. 또, 테이블 스페이스의 설정을 자동 확장으로 지정으로 하고 있었을 경우, 파일 레벨의 단편화가 일어나기 쉬워진다. 파일 레벨의 단편화의 상황은 OS레벨과 관련 있으므로, 예를 들면 Windows이면 defrag(조각모음) 기능을 이용해 조사하면 된다.

하이 워터 마크란?

이 의미대로, Oracle에 있어서의 HWM는 과거에 데이터가 저장된 것이 있는 제일 높은(마지막) 위치를 나타내는 지표가 된다. 하이 워터 마크의 개념은 데이터 파일과 세그먼트(segment)에 존재합니다.

그림1:하이 워터 마크의 개념

데이터 파일 레벨의 미사용 영역의 발생 ~HWM 이후의 미사용 영역~

데이터 파일내의 미사용 영역은, HWM 이후의 미사용 영역 및 HWM 이전의 세그먼트(segment)간의 미사용 영역으로부터 구성이 된다. HWM 이후의 미사용 영역은 데이터의 검색이나 갱신에는 영향을 주지 않지만, 저장하고 있는 데이터 용량에 비교해 데이터 파일의 사이즈가 크기 때문에, 실 데이터량에 비교해 백업·restore에 시간을 더 필요로 하게 된다.그 한편, 장래의 데이터 파일의 확장을 막기 위해서 사전 확보하고 있는 케이스도 있으므로, 통틀어 HWM 이후의 미사용 영역이 큰 것이 나쁜 것이라고는 말할 수 없다.장래의 데이터의 증가 예상과의 균형으로 사이즈의 조정을 실시해 주길 바란다. 어떤 A 테이블 스페이스에 존재하는 데이터 파일의 HWM 이후의 미사용 영역의 합계에 관한 정보는 이하의 SQL으로 파악이 가능하다.

＜테이블 스페이스의 데이터 파일마다 HWM 이후의 미사용 영역의 사이즈를 구하자＞

```
select sumdf.file_name "데이터 파일명",
to_char(sumdf.total_bytes, 'FM999,999,999,990') "테이블 스페이스의 사이즈",
to_char(sumfs.free_bytes, 'FM999,999,999,990') "HWM 이후의 미사용 공간사이즈"
from (select df.file_id, df.file_name, sum(df.bytes) total_bytes
from dba_data_files df
where df.tablespace_name = upper('&tsname')
group by df.file_id, df.file_name) sumdf
left outer join (select fs.file_id, fs.bytes free_bytes
  from (select fs2.file_id, fs2.bytes, fs2.block_id,
        max(fs2.block_id) over (partition by fs2.file_id) max_block
from dba_free_space fs2
  where fs2.tablespace_name = upper('&tsname')) fs
where fs.block_id = fs.max_block) sumfs
on (sumdf.file_id = sumfs.file_id);
```

```
tsname에 값을 입력해 주세요: USERS3

구    6:         where df.tablespace_name = upper('&&tsname')
신    6:         where df.tablespace_name = upper('USERS3')
구   12:                         where fs2.tablespace_name = upper('&&tsname')) fs
신   12:                         where fs2.tablespace_name = upper('USERS3')) fs

데이터 파일명                    테이블스페이스의 사이즈     HWM 이후의 미사용 영역의 사이즈
------------------------------------------------------------------------------
D:₩ORACLE₩ORADATA₩TEST1₩USERS03.DBF        786,432,000                34,504,704
D:₩ORACLE₩ORADATA₩TEST1₩USERS03_2.ORA      134,217,728               133,464,064
```

tsname에는 테이블스페이스명을 입력해 한다. DBA_FREE_SAPCE 딕셔너리의 각 데이터 파일마다 블록 ID가 제일 큰 빈 공간을 추출하고 있다.

덧붙여서, 어떤 A 테이블 스페이스 영역 전체의 미사용 공간을 구하는 SQL도 구해보았다..

＜특정 테이블 스페이스의 미사용 공간의 총사이즈를 구하는 SQL 문＞

```
select    to_char(sumdf.total_bytes, 'FM999,999,999,990') "테이블 스페이스의 사이즈",
to_char(sumdf.total_bytes - sumfs.free_bytes, 'FM999,999,999,990') "사용중인 영역의 사이즈",
to_char(sumfs.free_bytes, 'FM999,999,999,990') "미 사용영역의 사이즈",
```

```
to_char((nvl(sumfs.free_bytes, 0) / sumdf.total_bytes) * 100, 'FM990.99') || '%' 사용율"
from (select df.tablespace_name, sum(df.bytes) total_bytes
from dba_data_files df
 group by df.tablespace_name) sumdf
 left outer join (select fs.tablespace_name, sum(fs.bytes) free_bytes
 from dba_free_space fs
 group by fs.tablespace_name) sumfs
 on (sumdf.tablespace_name = sumfs.tablespace_name)
   where sumdf.tablespace_name = upper('&tsname');
```

데이터 파일 레벨의 미사용 영역의 발생 ~HWM 이전의 세그먼트(segment)간의 미사용 영역~

데이터 파일내의 미사용 영역은, HWM 이후의 미사용 영역 및 HWM 이전의 세그먼트(segment)간의 미사용 영역으로부터 구성된다. 여기에서는 후자에 대해 설명하고 있다.

세그먼트(segment)간의 미사용 영역은 물리 단위에서는 extent가 된다.로컬 관리 테이블 스페이스에서, 특히 UNIFORM 사이즈 지정의 경우는 이러한 영역도 효율적으로 이용되지만, AUTOALLOCATE 지정의 경우는 미사용인 채 남을 가능성이 있다. HWM 이전의 세그먼트(segment)간에 어느 정도의 미사용 extent가 존재하는지에 대해서는 아래의 SQL문으로 구할 수 있다.

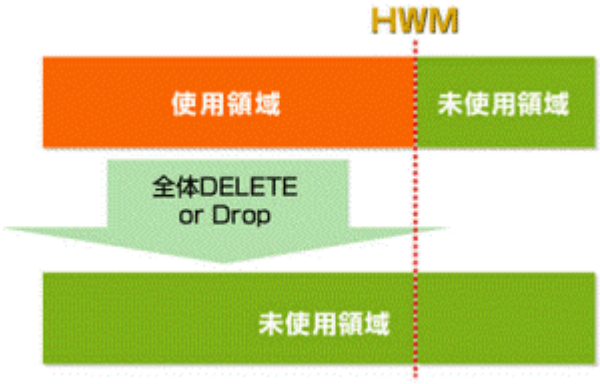
＜HWM 이전의 세그먼트(segment)간의 미사용 영역을 구하는 SQL 문＞

```
select   sumdf.file_name "데이터 파일명",
to_char(sumfs.free_bytes, 'FM999,999,999,990') "미사용영역의 사이즈"
from (select df.file_id, df.file_name from dba_data_files df
where df.tablespace_name = upper('POS_DATA')) sumdf
left outer join (select fs.file_id, fs.bytes free_bytes
from (select fs2.file_id, fs2.bytes, fs2.block_id,
 max(fs2.block_id) over (partition by fs2.file_id) max_block
   from dba_free_space fs2
    where fs2.tablespace_name = upper('POS_DATA')) fs
      where fs.block_id <> fs.max_block) sumfs
            on (sumdf.file_id = sumfs.file_id);
```

위치가 비싼 하이 워터 마크

HWM에 대한 속성 중의 하나는 HWM는 자동에서는 결코 내리지 않는다고 하는 점이다, 예를 들면 전체 레코드를 DELETE문으로 삭제했다고 해도, HWM의 위치는 그대로이다.

그림 2:DELETE로 움직이지 않는 하이 워터 마크



세그먼트(segment) 레벨의 HWM는 주로

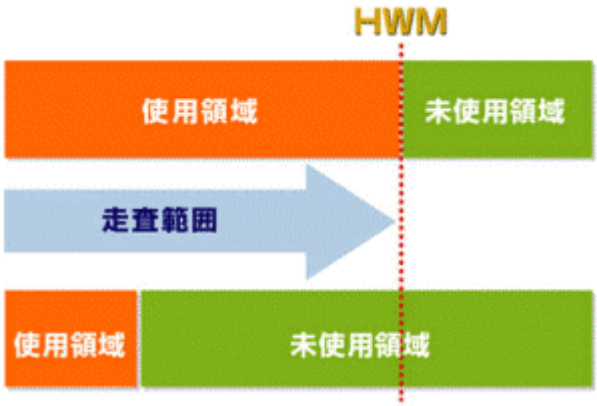
- 풀 테이블 스캔
- 다이렉트·로드/다이렉트·로드·인서트

에 영향을 준다.

(1) HWM의 풀 스캔에 대한 영향

테이블이나 인덱스의 풀 스캔을 실행하는 경우, 실제의 스캔 범위는 테이블이나 인덱스 전체가 아니고, HWM의 위치까지를 스캔 한다.이것에 의해, 그림 3의 윗부분과 같이, 실제로 용량을 확보하고 있는 사이즈에 비교해 데이터량이 적은 경우의 처리 시간을 단축하고 있다.그러나, HWM가 자동으로 내리지는 않는다.그 때문에, 일단 많이 데이터가 들어가 있는 상태로부터 대량 삭제가 있으면, 그림 3의 밑부분과 같이, 실제로는 데이터가 들어가 있지 않음에도 불구하고 HWM의 위치까지 스캔 해 버려, 실 데이터량에 비해 더 많은 검색 시간이 걸려 버린다.

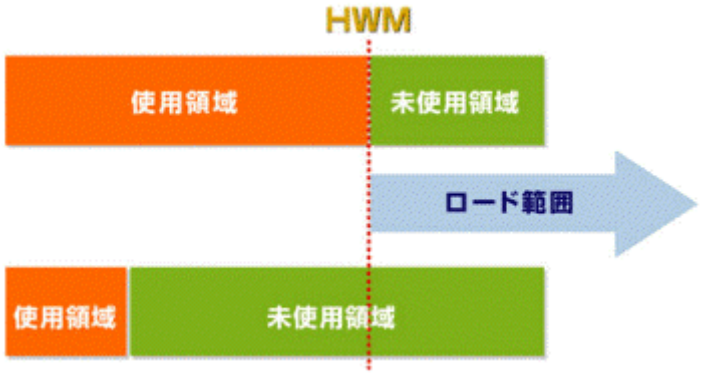
그림3:하이 워터 마크의 풀 스캔시의 영향



(2) HWM의 다이렉트 처리에 대한 영향

다이렉트·로드나 다이렉트·로드·인서트는 INSERT문에 의해 데이터를 삽입하는 것이 아니라, 먼저 블록에 저장된 포맷 이미지를 작성해, 그 블록 이미지를 직접 쓴다.그 때문에 조금이라도 데이터가 들어가 있는 블록에는 쓸 수가 없다. HWM 이후의 블록은 빈 것으로 간주되고 있기 때문에, 다이렉트 처리는 그림 4의 상부와 같이 HWM 이후의 블록에 데이터를 쓴다.따라서 이러한 처리에 의해서 다이렉트 처리는 퍼포먼스를 확보하고 있다.덧붙여서 패러렐·다이렉트·로드때는 HWM 이후의 extent로부터 쓴다. 만약 HWM 이전의 영역에 빈 곳이 많은 경우, 그림 4의 하부와 같이 세그먼트(segment)내에 큰 빈 공간이 생기게 된다. 만약 이 테이블에 다이렉트 처리에 의한 데이터 삽입 밖에 없는 경우, 이 빈 공간은 사용하지 없는 채 남아 버린다.

그림4:하이 워터 마크의 다이렉트 처리에의 영향



(3) HWM의 위치를 아는 방법

HWM의 위치를 알려면 DBMS_SPACE 패키지에 있는 UNUSED_SPACE 프로시저를 이용한다. 이 프로시저를 이용하기 위해서는, ANALYZE 혹은 ANALYZE ANY 시스템 권한이 필요하다. 다만, 실제로 ANALYZE를 실시하는 것은 아니기 때문에, RBO 로 운용하고 있는 시스템에서도 이용 가능하다. 또, 빈 공간 관리를 FREELIST가 아닌 자동 세그먼트(segment) 관리(ASSM)를 이용하고 있는 경우는, UNUSED_SPACE 프로시저는 아닌 SPACE_USAGE프로시저를 이용하지 않으면 잘못된 결과가 나와 버린다.

다음에 UNUSED_SPACE 프로시저의 이용 예를 이용해 HWM의 위치를 구해보자.

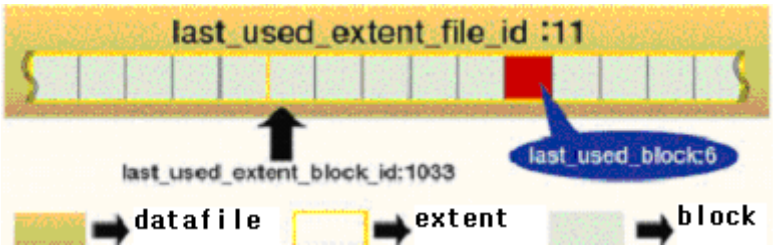
```
SQL> set serveroutput on
SQL> declare
  2     v_total_blocks                  number;
  3     v_total_bytes                   number;
  4     v_unused_blocks                   number;
  5     v_unused_bytes                    number;
  6     v_last_used_extent_file_id    number;
  7     v_last_used_extent_block_id number;
  8     v_last_used_block                 number;
  9  begin
 10     dbms_space.unused_space(upper('&uname'), upper('&sename'), '&stype',
 11        v_total_blocks, v_total_bytes, v_unused_blocks, v_unused_bytes,
 12        v_last_used_extent_file_id, v_last_used_extent_block_id, v_last_used_block);
 13     dbms_output.put_line('HWM 가 있는 데이터 파일의 ID              : '
 14        || to_char(v_last_used_extent_file_id, '9,999,990'));
 15     dbms_output.put_line('HWM 가 있는 extent 의 시작 블록 ID : '
 16        || to_char(v_last_used_extent_block_id, '9,999,990'));
 17     dbms_output.put_line('HWM 가 있는 블록의 위치                   : '
 18        || to_char(v_last_used_block, '9,999,990'));
 19  end;
 20  /
uname 에 값을 입력해 주세요: SCOTT
sename 에 값을 입력해 주세요: CUSTOMERS
stype 에 값을 입력해 주세요: TABLE
구  10:    dbms_space.unused_space(upper('&uname'), upper('&sename'),
'&stype',
신  10:    dbms_space.unused_space(upper('SCOTT'), upper('CUSTOMERS'),
'TABLE',
HWM 가 있는 데이터 파일의 ID              :          11
HWM 가 있는 extent 의 시작 블록 ID       :       1,033
HWM 가 있는 블록의 위치                   :           6


PL/SQL 프로시저가 정상적으로 완료했습니다
```

위의 SQL 스크립트의 실행에서, uname에는 세그먼트(segment) 소유자명을, sename에는 세그먼트(segment)명을,

stype에는 세그먼트(segment)의 타입(TABLE/TABLE PARTITION/TABLE SUBPARTITION/INDEX/INDEX PARTITION/INDEX SUBPARTITION/CLUSTER/LOB의 어느쪽이든)을 입력한다.

SQL 스크립트의 실행 결과를 설명하면, 「 HWM가 있는 데이터 파일의 ID 」 는, 프로시저의 V_LAST_USED_EXTENT_FILE_ID파라미터의 값이 된다.DBA_DATA_FILES 딕셔너리의 FILE_ID열이 V_LAST_USED_EXTENT_FILE_ID파라미터와 일치하는 데이터 파일 중에 HWM가 존재하는 것을 나타내고 있다. 「HWM가 있는 extent의 시작 블록 ID」는, 프로시저의 V_LAST_USED_EXTENT_BLOCK_ID파라미터의 값이 된다. *???*_EXTENTS(*???*는DBA/ALL/USER) 딕셔너리의 FILE_ID열이 V_LAST_USED_EXTENT_FILE_ID와 같고, 한편 BLOCK_ID열이 V_LAST_USED_EXTENT_BLOCK_ID와 일치하는 익스텐트 중에 HWM가 존재하는 것을 나타내고 있다.「HWM가 있는 블록의 위치」는 프로시저의 V_LAST_USED_BLOCK 파라미터의 값이 된다.

그림5:하이 워터 마크의 위치



세그먼트(segment) 레벨의 미사용 영역의 발생 ~HWM 이후의 미사용 영역~

HWM 이후의 미사용 영역은 다음과 같은 방법으로 파악이 가능합니다.

(1) DBMS_SPACE.UNUSED_SPACE 프로시저를 이용한다

바로 전에 기술된 DBMS_SPACE.UNUSED_SPACE 프로시저를 이용하는 것으로 세그먼트(segment)의 HWM 이후의 미사용 영역의 크기를 계산할 수 있다.

<HWM 이후의 미사용 영역을 구하는 실행문(1)>

```
SQL> set serveroutput on
SQL> declare
  2     v_total_blocks                number;
  3     v_total_bytes                 number;
  4     v_unused_blocks                 number;
  5     v_unused_bytes                  number;
  6     v_last_used_extent_file_id    number;
  7     v_last_used_extent_block_id number;
```

```
  8     v_last_used_block               number;
  9   begin
 10     dbms_space.unused_space(upper('&uname'), upper('&sename'), '&stype',
 11       v_total_blocks, v_total_bytes, v_unused_blocks, v_unused_bytes,
 12       v_last_used_extent_file_id, v_last_used_extent_block_id, v_last_used_block);
 13     dbms_output.put_line('현세그먼트(segment) 용량 : '
 14       || to_char(v_total_bytes, '999,999,999,990') || '아르바이트');
 15     dbms_output.put_line('소비 용량            : '
 16       || to_char(v_total_bytes - v_unused_bytes, '999,999,999,990') ||
'아르바이트');
 17     dbms_output.put_line('나머지 용량          : '
 18       || to_char(v_unused_bytes, '999,999,999,990') || '아르바이트');
 19     dbms_output.put_line('소비율               :           '
 20       || to_char((v_total_bytes - v_unused_bytes) / v_total_bytes * 100, '990.99')
|| ' %');
 21   end;
 22   /
uname에 값을 입력해 주세요: SCOTT
sename에 값을 입력해 주세요: CUSTOMERS
stype에 값을 입력해 주세요: TABLE
구  10:    dbms_space.unused_space(upper('&uname'), upper('&sename'), '&stype',
신  10:    dbms_space.unused_space(upper('SCOTT'), upper('CUSTOMERS'),
'TABLE',
현세그먼트(segment) 용량 :         9,437,184 바이트
소비 용량          :         8,593,408 바이트
나머지 용량          :           843,776 바이트
소비율             :             91.06 %

PL/SQL 프로시저가 정상적으로 완료했습니다.
```

위의 SQL의 실행에서, uname에는 세그먼트(segment) 소유자명을, sname에는 세그먼트(segment)명을, stype에는 세그먼트(segment)의 타입(TABLE/TABLE PARTITION/TABLE SUBPARTITION/INDEX/INDEX PARTITION/INDEX SUBPARTITION/CLUSTER/LOB)을 입력한다.

(2) ANALYZE 실행 후 딕셔너리를 참조한다.

테이블의 경우는 ANALYZE를 실행한 후의 *XXX*_TABLES 딕셔너리의 BLOCKS와 EMPTY_BLOCKS의 값을 검색하는 것으로 HWM 이후의 미사용 영역의 크기가 구할 수 있다.BLOCKS는 세그먼트(segment)내의 사용이 끝난 블록수, EMPTY_BLOCKS는 세그먼트(segment)내의 미사용 블록수(HWM 이후)를 나타낸다.

<HWM 이후의 미사용 영역을 구하는 실행문(2)>

```
SQL> analyze table scott.customers compute statistics;

테이블이 분석되었습니다.

SQL> select to_char((blocks + empty_blocks) * 8192, 'FM999,999,999,990') as
"테이블 용량",
  2  to_char(empty_blocks * 8192, 'FM999,999,999,990') as "나머지 용량"
  3  from dba_tables where owner = '&uname' and table_name = '&tname';
uname에 값을 입력해 주세요: SCOTT
tname에 값을 입력해 주세요: CUSTOMERS
구   3: from dba_tables where owner = '&uname' and table_name = '&tname'
신   3: from dba_tables where owner = 'SCOTT' and table_name = 'CUSTOMERS'

테이블 용량 나머지 용량
------------------------------ ------------------------------
9,428,992                      843,776
```

상기는 테이블의 사용량에 관해서 SQL의 실행에 임하여, uname에는 세그먼트(segment) 소유자명을, tname에는 테이블명을 입력한다.

세그먼트(segment) 레벨의 미사용 영역의 발생 ~HWM 이전의 미사용 영역~

HWM 이전의 미사용 영역의 영향은 , 본 장의 처음에 설명한 HWM의 설명을 참조하자. 계산방법은, 테이블만의 방식이지만, 아래와 같이 SQL으로 산출할 수 있다.

<HWM 이전의 미사용 영역을 구하자>

```
SQL> analyze table scott.customers compute statistics;

테이블이 분석되었습니다.
```

```
SQL> select to_char(avg_space * blocks, 'FM999,999,999,999') "빈영역"
  2  from dba_tables where owner = '&uname' and table_name = '&tname';
uname 에 값을 입력해 주세요: SCOTT
tname 에 값을 입력해 주세요: CUSTOMERS
구    2: from dba_tables where owner = '&uname' and table_name = '&tname'
신    2: from dba_tables where owner = 'SCOTT' and table_name = 'CUSTOMERS'

빈영역
--------------------------------
595,264
```

위의 테이블의 사용량에 관해서 SQL의 실행에 대해, uname에는 세그먼트(segment) 소유자명을, tname에는 테이블명을 입력한다.

계층이 깊은 인덱스

B*Tree 인덱스는 계층 구조가 되어 있습니다. 이 계층이 깊으면 검색에 시간이 걸리게 된다.계층의 깊이는, 인덱스를 ANALYZE 한 뒤, INDEX_STATS 딕셔너리의 HEIGHT열, 이것이 없다면 *xxx*_INDEXES 딕셔너리의 BLEVEL 컬럼으로 알 수 있다. 이러한 컬럼의 값이 4 이상일 경우는, 인덱스를 이용한 검색의 퍼포먼스에 영향을 주기 때문에 재구성을 해주어야 한다. 참고로 INDEX_STATS 딕셔너리를 검색하는 경우는, VALIDATE STRUCTURE 옵션으로 ANALYZE를 실행할 필요가 있다.

◆ 제4장 extent 레벨의 단편화

불연속의 extent

어떤 A 세그먼트(segment)를 구성하는 extent가 연속해서 확보되어 있지 않아도 퍼포먼스에 거의 영향을 주지 않는다. 단지, 불연속의 extent간의 타 세그먼트(segment)의 extent가 미사용 extent가 되었을 경우, 특히 딕셔너리 관리 테이블 스페이스에서 세그먼트(segment)마다 개별의 INITIAL/NEXT/PCTINCREASE를 지정해 있는 경우는 공간의 낭비가 발생하기 쉽다. 로컬 관리의 경우는 이러한 일이 일어나기 힘들고, 이를 신경 쓰지 않아도 된다. 이를 검시하고 싶은 경우는 아래와 같은 스크립트로 검사를 할 수 있다.

< extent 의 연속 여부를 조사한다 >

```
select ext2.extent_id "extent ID",
```

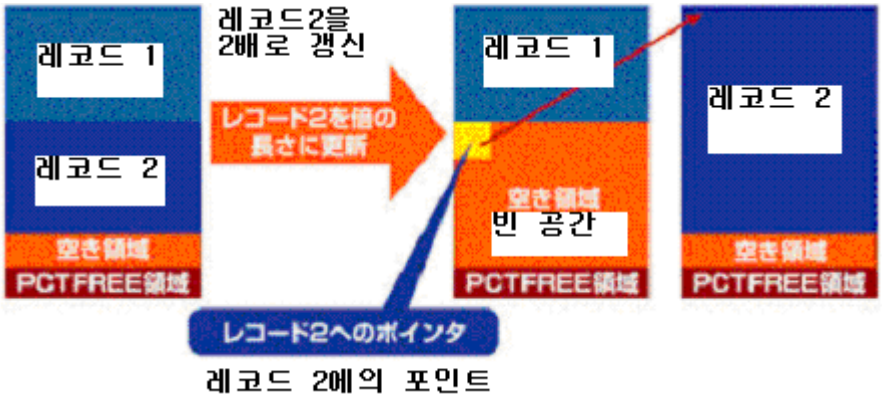
```
ext2.file_id “파일ID",
ext2.block_id “개시블록ID",
ext2.blocks "블록수",
case when ext2.extent_id = 0 then ‘처음의 EXTENT입니다’
when ext2.old_fid <> ext2.file_id then ‘데이터 파일이 다릅니다’
when ext2.old_blk_id <> ext2.block_id then ‘불연속의 EXTENT입니다’
else '연속하는 EXTENT입니다’
end “단편화상황”
from (select ext1.extent_id, ext1.file_id, ext1.block_id, ext1.blocks,
lag(ext1.file_id, 1) over (order by ext1.extent_id) old_fid,
lag(ext1.block_id, 1) over (order by ext1.extent_id) +
lag(ext1.blocks, 1) over (order by ext1.extent_id) old_blk_id
from dba_extents ext1
where ext1.owner = '&uname' and ext1.segment_name = = '&sname') ext2;
```

```
uname 에 값을 입력해 주세요: SCOTT
sname 에 값을 입력해 주세요: C3
구  13:    where ext1.owner = '&uname' and ext1.segment_name = '&sname') ext2
신  13:    where ext1.owner = 'SCOTT' and ext1.segment_name = 'C3') ext2
extent ID 파일 ID 개시 블록 ID 블록수단편화 상황
---------- ---------- ---------- ---------- ----------------
         0         12        113          2 선두의 extent 입니다
         1         13         93          2 데이터 파일이 다릅니다
         2         12        115          2 데이터 파일이 다릅니다
         8         12        123          2 불연속의 extent 입니다
         9         12        125          2 연속하는 extent 입니다
10 행이 선택되었습니다.
```

행 이행(ROW MIGRATION)

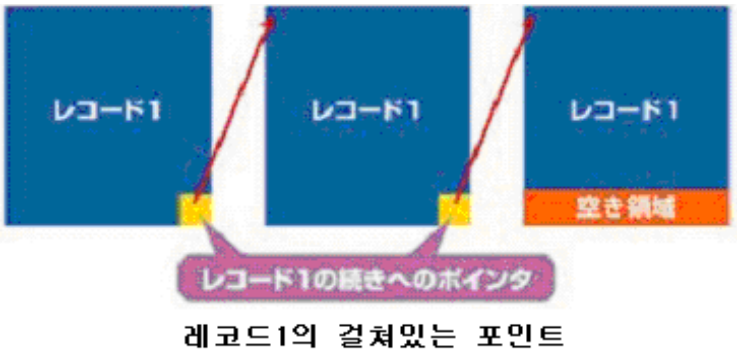
블록에 공간을 차지하고 있는 레코드에 대해서 갱신을 했을 때에, PCTFREE로 확보한 영역을 가지고 있어도 원래 존재하고 있던 블록에 들어가지 않을 경도의 크기라면 Oracle는 해당 레코드를 다른 블록에 저장한다. 이 때, 원래의 블록에 새로운 저장영역의 포인터를 남긴다. 이러한 상태를 행 이행(로우 마이그레이션)이라고 부른다.행 이행이 발생하면 해당 레코드는 본래 1 블록에 들어가는 길이에서 만나도 2 블록에 걸쳐서 저장된다. 즉, 이 레코드에 액세스 하기 위해서는 2개의 블록을 읽어 들일 필요가 있는 것이다. 행 이행이 발생하고 있는 레코드에의 액세스가 많으면 읽기 I/O의 증가나 캐쉬 히트율의 저하가 발생하게 된다.

그림 6: 행 이행(ROW MIGRATION)



행 체인(ROW CHANING)

행 체인현상은 행 이행과 같이 레코드가 복수 블록에 걸치는 현상이지만, 행 체인은 블록에 저장 가능한 사이즈 이상의 레코드가 복수 블록에 건너 저장되는 현상이다. 물리적으로 큰 레코드를 넣으려 하고 있는 이상 어쩔 수 없는 현상이다, 빈번히 발생한다면 행 이행과 같은 영향이 있다.



행 이행· 행 체인을 조사하자

행 이행과 행 체인은 같은 방법으로 파악할 수 있다.테이블을 ANALYZE 한 후, *XXX*_TABLES 딕셔너리의 CHAIN_CNT 열에 행 이행 또는 행 체인하고 있는 레코드 수가 나타난다. 즉, 행 이행과 행 체인는 따로 파악할 수 없다.

블록 레벨의 미사용 영역의 발생

세그먼트(segment) 용량에 비교해 저장 가능한 데이터 량이 적은 경우는, 블록의 내용이 효율적으로 이용되어 있지 않은 경우를 생각할 수 있다. 예를 들면 데이터 저장 가능 영역이 3000바이트에 대해서 레코드 사이즈가 평균 2000바이트의 경우, 단순 계산이라면 1 블록에 대해 1000바이트의 낭비가 발생하게 된다. 테이블이라면 DELETE 비교해 PCTUSED의 값이 낮은 경우에 블록 레벨의 미사용 영역이 발생하기 쉬워진다. 인덱스의 경우는 DELETE가 많은 경우, 인덱스 대상열의 값이 오름차순이 아니고 랜덤이나 내림차순에 INSERT 되는 경우, 인덱스 대상 열에 대해서 갱신이 발생하는 것이 많은 경우에 블록 레벨의 미사용 영역이 발생하기 쉬워진다. 산정수치에 대해서 실제의 용량(HWM까지의 용량)이 너무 큰 경우는 재편성을 검토해 하자.

제 6 부 : 단편화의 해결

◆ 제1장 단편화 해결의 개요

본 장에서는 단편화의 해결 방법에 대해 설명하기로 한다. 단편화의 해소의 작업은 가능한 한 발생하지 않는 것이 DBA에 있어서도 좋다고 생각되지만, 해당의 단편화가 발생되는 것을 방지하는 사전의 대책에 대해서도 설명하기로 한다.

단편화 해결의 개요

단편화의 해결 방법은, 단편화 현상에 의해 어느 정도는 다르지만, 공통의 방법으로 해결할 수 있는 것도 있다. 우선은 단편화 현상마다 해결 방법을 리스트로 정리해 보았다.

계층	단편화 현상	해결 방법(재편성 방법)	해설장
데이터 파일(테이블 스페이스)	파일 레벨의 단편화	OS 커멘드에 의한 defrag	제2장
	데이터 파일 레벨의 미사용 영역의 발생	테이블 스페이스 레벨의 재편성	제2장
		데이터 파일의 축소	제2장
세그먼트(segment)	위치가 높은 하이 워터 마크	세그먼트(segment) 레벨의 재구성	제3장
	세그먼트(segment) 레벨의 미사용 영역의 발생	extent의 잘라서 버림	제3장
		세그먼트(segment) 레벨의 재구성	제3장
	계층이 깊은 인덱스	세그먼트(segment) 레벨의 재구성	제3장
extent	불연속의 extent	세그먼트(segment) 레벨의 재구성	제3장
		표 영역 레벨의 재구성	제2장
블록	행 이행	세그먼트(segment) 레벨의 재구성	제3장
		행 이행 하고 있는 레코드만의 재구성	제4장
	행 체인	블록 사이즈 변경해 재구성	제4장
	블록내의 미사용 영역의 발생	세그먼트(segment) 레벨의 재구성	제3장

◆ 제2장 데이터 파일(겉(표) 영역) 레벨의 단편화 대책

OS커멘드에 의한 degrg(조각모음)

UNIX나 Linux등에서는 파일의 단편화는 별로 신경 쓸 필요는 없다. Windows계 OS에서도 심하지 않는 정도의 단편화는 퍼포먼스에의 영향은 거의 없다.단지, 시스템 구축 후 한번도 defrag를 실시했던 적이 없는 것이면 한 번쯤 실시하는 것이 좋을 것이다. Windows계 OS상에서 Oracle를 구축하는 경우는 **degrg(조각모음)** 툴로 파일 레벨의 단편화를 해소한다. 예를 들면 Windows2000계의 OS의 경우라면 「시작 버튼」→「프로그램」→「보조 프로그램」→「시스템 도구」→「디스크 조각모음」로 기동할 수 있습니다.덧붙여 defrag 툴 등으로 최적화를 실시하는 경우는 인스턴스는 정지해 두어야 한다.

테이블 스페이스 영역 레벨의 재구성

Oracle9*i*으로부터 있는 테이블 스페이스에 저장되는 테이블을 통째로 Export/Import 할 수 있게 되었다. 이 기능을 사용하면, 어떤 A 테이블 스페이스 전체를 정리해서 재구성 하는 것이 가능하다. 다음의 과정이 재구성의 순서가 된다.

(1) Export를 실시

Export시에 TABLESPACE 옵션에 테이블 스페이스 명을 지정하면, 지정한 테이블 스페이스 영역에 존재하는 오브젝트가 Export 됩니다. EXP_FULL_DATABASE role을 소유하고 있는 유저로 실시한다.

```
c:₩> exp system/manager tablespace=test01 file=c:₩temp₩test.dmp
log=c:₩temp₩testexp.log
```

(2) 오브젝트의 삭제＆재작성

테이블 스페이스의 파라미터를 변경하고 싶은 경우는 테이블 스페이스를 삭제→재구성합니다. 겉(표) 영역의 파라미터를 하지 않는 경우에서도 겉(표) 영역을 삭제→재작성하는 것이 편합니다.TABLESPACE 옵션을 지정한 Export로 작성한 덤프 파일에는 CREATE TABLESPACE문은 포함되지 않으므로, 겉(표) 영역은 먼저 작성해 둘 필요가 있습니다.만약 Export시와 다른 파라미터로 테이블이나 인덱스를 작성하고 싶은 경우는, 하늘의 테이블이나 인덱스를 새로운 파라미터의 값으로 사전에 작성해 주세요.

```
SQL> drop tablespace test01 including contents and datafiles;
SQL> create tablespace test01 datafile ....;
```

(3) Import를 실시

Import시는 TABLESPACE 옵션은 불필요합니다.만약(2)으로 하늘의 세그먼트(segment)를 작성한 경우는 IGNORE 옵션을 「Y」로 지정해 주세요.IMP_FULL_DATABASE 롤을 소유하고 있는 유저로 실시한다.

```
c:₩> imp system/manager tablespace=test file=c:₩temp₩test.dmp
log=c:₩temp₩testimp.log
```

데이터 파일의 축소

초기에 확보한 데이터 파일의 용량에 비교해 실제로 데이터가 들어 오지 않으면 미사용 영역으로서 경우에 따라서는 많은 디스크 용량을 소비한다. 데이터가 들어 올 가능성이 없는 경우 등은 미사용 영역을 잘라 버리는 것도 가능하다.데이터 파일을 축소하는 경우는, 이하의 SQL로 실시한다.

```
SQL> alter database datafile 'c:₩temp₩test.dbf' resize 100m;
```

데이터 파일명 및 축소 후의 사이즈를 지정한다.다만, 잘라 버릴 수 있는 것은 HWM(하이 워터 마크) 이후의 영역만이다. HWM에 대해서는 제5부 제 2장을 참조하자. 만일 HWM 이전에 단편화 된 미사용 영역이 있어도 축소의 대상으로는 되지 않는다. 축소라고 하는 것보다는 뒷부분의 미사용 영역의 잘라버리는 처리가 된다.덧붙여서 축소 뿐만이 아니라 확장도 같은 구문으로 가능하다.

파일 레벨의 단편화의 방지책

파일 레벨의 단편화를 막기 위해서는 이하와 같은 방책을 채택하면 유효하다

(1) 데이타베이스 작성전에 defrag(조각모음)를 실행한다

데이타베이스 작성 전에 조각모음을 하지 않은 상태고 신규 디스크가 아니라면 그 후에 작성하는 파일은 단편화 할 가능성이 높다. 테이블 스페이스를 배치할 예정의 디스크(드라이브, 파티션)에 대해서는, 신규디스크가 아니라면 사전에 defrag(조각모음)를 실행합시다.

(2) 최대한 전용 디스크에, 없다면 전용 파티션에 데이터 파일을 배치하자

데이타베이스를 구성하지 않는 파일 군과 같은 드라이브(파티션)에 데이터 파일을 배치하면 단편화 하기 쉽습니다. 소규모 시스템으로 디스크가 하나라고 했을 경우에서도, 새롭게 파티션을 작성하고, 거기에는 데이타베이스와 무관한 파일을 두지 않게 하자.

(3) 사전의 산정을 제대로 실시한다

다른 레벨의 단편화의 예방책이기도 이지만, 사전에 세그먼트(segment)나 데이터 파일의 견적을 실시해 필요분을 확보해 두면, 막상 예상이 빗나가 단편화가 발생해도 추측하지 않고 적당하게 확보했을 경우보다 단

편화의 영향을 억제할 수 있다.

(4) 데이터 파일의 자동 확장에 의지하지 말자

데이터 파일이 자동 확장하면, 그 드라이브(파티션)에 해당 파일 밖에 없는 경우 이외에는 단편화가 발생 할 수 있다.최대한 사전 산정을 하고, 필요 충분한 파일 용량을 사전에 확보해 주자. (제2부 제 2장의 마지막 항 참조).

◆ 제3장 세그먼트(segment) 레벨의 단편화 대책

세그먼트(segment) 레벨의 재구성 : 개요

개별의 세그먼트(segment)의 재구성 방법은 몇 개가 존재한다.

- 테이블, 인덱스의 Export/Import
- 테이블의 이동(MOVE)
- 인덱스의 재구성(REBUILD)

테이블, 인덱스의 Export/Import

테이블과 그 테이블에 부수적인 인덱스를 정리해 재구성 하는 경우에 유용하다. 복수의 테이블이나 인덱스를 정리해 실시하는 것도 가능하다. 그러나 재구성 중에 해당의 테이블에 액세스 할 수 없다. 다음의 순서로 실행한다.제2장의 테이블 스페이스 레벨의 재구성과 순서는 거의 같아서, 상세한 실행 로그는 생략한다.

(1) Export를 실시하자

재구성을 목적으로 Export를 실시할 때의 주의점으로서는,

- 순서 2로 테이블을 재 작성하는 경우는, 속해있는 인덱스나 제약이나 트리거 등 도 Export 해 둔다. 별도의 작성 SQL가 있으면 그것을 재실행해도 괜찮지만(이쪽이 재구성이 빨리 끝나는 케이스도 있다), 정리해서 Export/Import 하는 것이 쉽다.
- 로컬 관리 테이블 스페이스의 경우, COMPRESS 옵션은 의미가 없기 때문에, 지정은 불필요하다.

예)

```
c:₩> exp scott/tiger tables=(emp, dept) file=c:₩temp₩test.dmp
log=c:₩temp₩testexp.log
```

(2) 오브젝트의 삭제&재생성

Export 한 세그먼트(segment)를 삭제한다.세그먼트(segment)의 파라미터를 변경하고 싶은 경우는 세그먼트(segment)를 일단 삭제하고 빈공간에서 재생성한다.

(3) Import를 실시

만약(2)으로 빈공간의 세그먼트(segment)를 작성한 경우는 IGNORE 옵션을 「Y」로 지정해 준다.

```
c:₩> imp scott/tiger full=y file=c:₩temp₩test.dmp log=c:₩temp₩testimp.log
```

테이블의 이동(MOVE)

Oracle8i로부터 테이블이 존재하는 테이블 스페이스를 아래의 구문으로 변경 할 수 있게 되었다.결과적으로, 새로운 테이블 스페이스로 테이블이 이동된다. 이 때 재구성도 실행된다. 물론 STORAGE구를 적어 테이블의 크기를 조정하는 것도 가능하다.

```
SQL> alter table table1 move tablespace tbs1;
```

이 SQL로 tablespace구에 현재 테이블이 존재하는 테이블 스페이스와 같은 테이블 스페이스을 지정하든가, 혹은 tablespace구자체를 생략하면(alter table table1 move) 같은 테이블 스페이스내에 이동된다. 다른 테이블 스페이스로 이동하는 경우도 물론이거니와, 같은 테이블 스페이스로 이동하는 경우에서도 재구성 후의 테이블의 크기와 동일한 빈 공간이 이동되는 테이블 스페이스에 요구 된다.

인덱스의 재구성(REBUILD)

인덱스에 대해서는 Oracle7.3부터 재구성(REBUILD) SQL 구문을 지원하고 있다..

```
SQL> alter index index1 rebuild;
```

이 구문을 이용하면, 인덱스를 DROP 하고 나서 CREATE 하는 것보다도 고속으로 인덱스를 재구성 할 수 있다. TABLESPACE구를 적어 저장 테이블 스페이스 영역을 변경하거나 STORAGE구를 적어 인덱스의 크기를 조정하거나 할 수도 있다.

```
SQL> alter index index1 rebuild tablespace tbs2 storage(initial 50m);
```

extent의 잘라 버림

초기에 확보한 세그먼트(segment) 용량에 비교해 실제로 데이터가 들어 오지 않으면 미사용 영역으로서 계속 남아 있으면 디스크 용량을 낭비하게 된다. 데이터가 들어 올 가능성이 없는 경우 미사용 extent를 잘라 버리는 것도 가능하다. extent를 잘라 버리는 경우는 아래의 SQL로 실시한다.

```
SQL> alter table table1 deallocate unused;
```

인덱스의 경우는 「alter table」을 「alter index」로 바꾸면 된다. 이 SQL를 실행하면 해당의 세그먼트(segment)의 HWM 이후의 미사용의 extent 모두를 잘라 버릴 수 있다. 미사용의 extent를 조금 남기고 싶은 경우는

```
SQL> alter table table1 deallocate unused keep 120m;
```

과 keep구에 사이즈를 지정하면, 120 MB가 되도록 HWM 이후의 미사용 extent를 잘라 버립니다. 다만, 로컬 관리 테이블 스페이스의 경우 extent의 사이즈가 어떠한 형태로 테이블 스페이스 레벨로 정해져 있다.예를 들면 상기 SQL를 UNIFORM 사이즈 18 MB의 테이블 스페이스의 세그먼트(segment)에 대해서 실행했을 경우, 120 MB는 아니고 126 MB의 크기가 된다.

세그먼트(segment) 레벨의 단편화의 방지책

세그먼트(segment) 레벨의 단편화를 막기 위해서는 이하와 같은 방책을 채택하면 유용하다. 단지, 인덱스의 경우는 READ ONLY라든지, 값이 순차적으로 증가해 가는 INSERT 밖에 없는 경우를 제외하고, 점차적으로 세그먼트(segment) 레벨의 단편화가 발생한다. 테이블도, 삭제가 많은 테이블은 아무래도 세그먼트(segment) 레벨의 단편화가 발생하기 쉬워진다.

(1) 사전의 산정을 제대로 실시하자

다른 레벨의 단편화의 예방책이기도 하지만, 사전에 세그먼트(segment) 크기의 산정을 실시해 필요분 만큼을 확보해 두면, 막상 예상이 빗나가 단편화가 발생해도 추측하지 않고 적당하게 확보했을 경우보다 단편화의 영향을 억제할 수 있다.

(2) PCTFREE/PCTUSED의 조정

(1)과 같이 PCTFREE/PCTUSED를 제대로 실시하면 재구성의 필요성이 줄어든다. 상세한 것에 제3부,제4부를 참조하자.

(3) 세그먼트(segment)의 파티션화

Enterprise Edition 및 Partitioning Option를 이용의 경우에 한정되지만, 테이블이나 인덱스를 파티션화 하면 단편화의 발생을 막을 수 있다. 예를 들면 3개월 보존으로 3개월 이상 경과한 데이터를 삭제하는 사양이 되어 있는 경우, 1개월마다 파티션화하고, 월 처리로 제일 오래된 파티션을 삭제하도록 하면, 월 처리에 의해서 HWM가 쓸데 없게 높아지는 것이 없어진다. 만약 단지 DELETE문으로 삭제했을 경우는 HWM 이전의 미사용 영역이 대량으로 발생해, 곧바로 그 미사용 영역이 재이용되지 않으면 제5부 제 3장 그리고 설명한 것과 같은 영향이 미친다. 또, 재구성을 실행할 때에도 파티션 단위에 재구성을 실행할 수 있기 때문에, 재구성 대상의 총 세그먼트(segment) 용량을 줄여, 단시간에 재구성을 실행할 수 있는 효과를 기대할 수 있다.

◆ 제4장 블록 레벨의 단편화 대책

행 이행(MIGRATION)의 해소

행 이행(MIGRATION)의 해소는 세그먼트(segment) 레벨 내지 테이블 스페이스 레벨의 재구성을 실시하면 해소할 수 있다.그러나, 아주 일부의 레코드만이 행 이행 하고 있는 상태이면, 아래의 순서로 행 이행 하고 있는 레코드만 재구성이 가능하다.

(1) CHAINED_ROWS 테이블을 작성하자

ORACLE_HOME 밑의 rdbms₩admin₩utlchain.sql 스크립트 파일을 재구성 대상 테이블을 소유하고 있는 유저로 실행하고, CHAINED_ROWS 테이블을 작성한다.

(2) ANALYZE를 실시한다

list chained rows 옵션을 붙여 analyze를 실시합니다.

```
SQL> analyze table test list chained rows;
```

위의 SQL를 실행하면, CHAINED_ROWS 테이블의 HEAD_ROWID열에 행 이행이 발생하고 있는 레코드의 ROWID가 저장된다.

(3) 워크(임시) 테이블에 행 이행(MIGRATION) 되고 있는 레코드를 임시 저장시키자

재구성 대상 테이블과 같은 구성의 워크(임시) 테이블을 작성해, 여기에 CHAINED_ROWS 테이블에 저장되고 있는 ROWID의 레코드를 복사한다.

```
SQL> create table work_test as select * from test
  2  where exists (select 'X' from chained_rows
  3  where head_rowid = test.rowid);
```

(4) 행 이행 되어 있는 레코드를 삭제

```
SQL> delete from test where exists (select 'X' from chained_rows
  2  where head_rowid = test.rowid);
```

(5) 워크 테이블에 저장시킨 레코드를 다시 입력한다

INSERT 된 레코드에서는 행 이행은 발생하지 않는다.

```
SQL> insert into test select * from work_test;
```

(6) 워크(임시) 테이블을 삭제

```
SQL> drop table work_test;
```

행 체인의 재구성

행 체인의 해소 방법은, 해당의 테이블의 존재하는 테이블 스페이스 레벨의 재구성을 실시한다. 테이블 스페이스 레벨의 재편성에 대해서는 제 2장을 참조하자. 이 때, 테이블 스페이스의 블록 사이즈를 평균내지 최대 레코드 사이즈보다 크게 해 준다.다만, BLOB에 음성 데이터를 저장하고 있는 경우 등 Oracle의 블록 사이즈의 최대치보다 큰 데이터를 저장하고 있는 경우는 행 체인의 발생은 피할 수 없다. 이 경우는 행 체인의 발생 회수를 줄이는 것을 목적으로, 사용중의 Oracle로 지정할 수 있는 제일 큰 블록 사이즈로 해 재구성을 실시한다.

행 이행의 방지책

행 이행의 방지책은, PCTFREE의 적절한 조정이다. 상세한 것에 대하여는 제3부 (을)를 참조하자. 고정길이의 데이터형을 사용하면 행 이행은 발생하지 않을 것이다. 모든 열을 CHAR형 및 NOT NULL로 작성한 시스템에서는, 행 이행이 발생하지 않게 되는 장점은 있지만, 데이터의 I/O량이 증가해 버리는 단점이 훨씬 더 크기 때문에, 이러한 설계는 추천하지 않는다.

행 체인의 방지책

행 체인의 방지책은, 테이블의 산정을 제대로 해 최대 레코드길이나 평균 레코드 길이을 산출해, 거기에 알맞은 블록 사이즈를 선택하는 것이다. Oracle9*i*이후는 테이블 스페이스마다 블록 사이즈를 달리 선택할 수 있기 때문에, 레코드 사이즈가 넉넉한 테이블을 같은 테이블 스페이스에 모으는 것도 한 방법이다.

◆ 제5장 정리

본 장에서는 단편화의 해소 방법에 대해 설명했다. 단편화를 막는 최선의 방법은, 사전에 단편화 하기 어려운 물리 설계를 실시하는 것이다. 반년에 걸쳐 본 연재로 해 온 것을 향후의 데이타베이스의 물리 설계에 종합적으로 도입

하면, 단편화의 감소 뿐만 아니라, 데이타베이스의 유지보수 작업 전반을 줄이는 것이 가능하게 되어, 보다 안정된 운용을 할 수 있게 되었으면 한다. 우선은 본 연재 내용에 따른 물리 설계를 기초로서 나머지는 각 시스템의 특성에 따라 본 연재의 해설 내용을 커스터마이징 해주기 바란다.