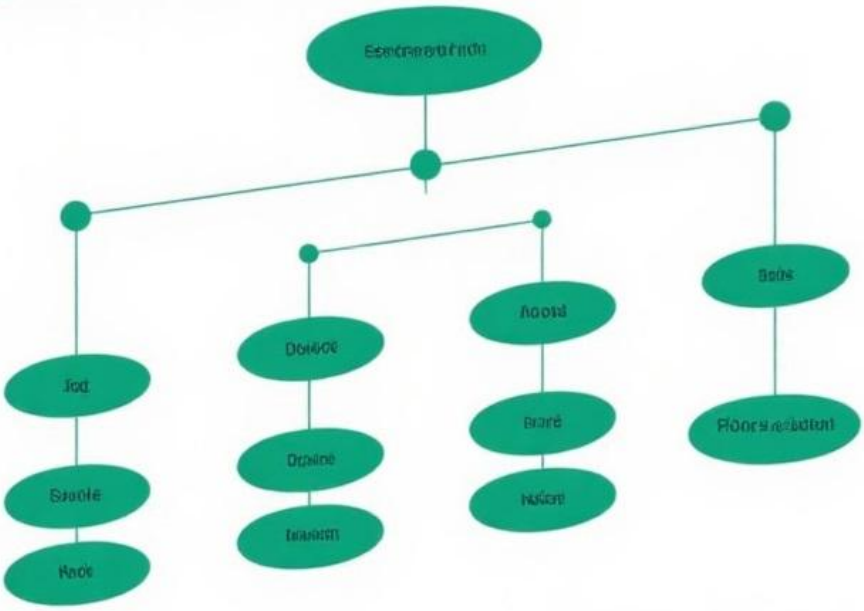
Oracle Storage Internals

XIN ZENGGUO



Contents

Preface

In 2005, I participated in an IT project to develop a bond business system for a bank, using Oracle RDBMS as the database. After the project concluded and the system went live, everything worked smoothly initially. However, a month later, a problem emerged. Before nightly batch operations began, the system needed to back up data from certain tables. At first, the data volume was small, and backups completed quickly—taking just a few minutes. But after a month, the backup process started requiring about half an hour, and as the data grew, the time kept increasing. We urgently needed a faster backup solution. One team member suggested directly reading data from Oracle data files for backups, as this would be the fastest method. However, our team lacked the technical expertise to implement this, so the idea was abandoned.

This suggestion posed a technical challenge, and from that point on, I began studying Oracle's internal storage structures. I soon realized how vast and complex the system was, requiring an understanding of numerous Oracle concepts and storage management mechanisms. I had to learn incrementally, accumulating knowledge with the goal of writing a program to extract table data directly from storage files. Later, I came across a tool called DUL (Data Unloader), which could extract data from Oracle files without starting the database. This discovery boosted my confidence, proving that developing such a tool wasn't just a pipe dream—it was achievable.

After years of effort, the tool I developed finally matched DUL's capabilities in exporting data, which was incredibly encouraging. Over this period, Oracle evolved from 10g to 11g, introducing new storage types like SecureFile LOBs, which demanded further research.

A few years ago, I compiled my research methods and processes into a document meant solely for myself, as a guide for building a custom data extraction tool. In recent years, I've explored other aspects of Oracle and deepened my understanding of its internal storage structures. Now, I've restructured that original document into a book, making it more reader-friendly. I've added diagrams and expanded the content to clarify key points, hoping to make the material accessible not just to me, but to others who might find it useful.

Audience

This book is intended for readers with a foundational understanding of Oracle databases. While basic database concepts are occasionally explained in context, prior familiarity with core principles is assumed. To fully grasp certain concepts, supplementary knowledge from Oracle documentation or external resources may be necessary.

Oracle Database is implemented in the C programming language, so familiarity with C will aid in comprehending underlying data structures. However, prior knowledge of C is not strictly required.

If you have explored Oracle's internal data storage mechanisms and seek deeper insights into its data structures—or have unresolved questions—this book will prove invaluable.

Oracle Database Administrators (DBAs) and developers will also benefit significantly. By studying Oracle's internal storage mechanics, you can:

- Enhance database performance.
- Design database tables with optimal column types.
- Write more efficient SQL statements in applications.

This book bridges theoretical knowledge and practical implementation, empowering professionals to leverage Oracle's architecture for robust, high-performance solutions.

What can you gain from this book

By reading this book, you will gain insights into how data is stored in ordinary Oracle database tables. You'll observe how each row of data is distributed within data blocks and understand how a single block manages these data rows. Delving deeper, you'll explore the storage mechanisms of clustered tables and learn how data rows from multiple tables are managed within a single data block. The book also reveals how Oracle indexes are stored in data blocks and explains the process of locating corresponding data rows through indexes. Additionally, you'll discover the storage architecture of index-organized tables (IOTs).

A significant portion of the book is dedicated to Oracle LOB data storage, providing detailed descriptions of storage structures from the legacy BasicFile LOB to the modern SecureFile LOB, along with analysis of their performance implications.

Mastering this knowledge will enable you to:

1. Gain profound understanding of Oracle's internal storage mechanisms.
2. Diagnose root causes behind Oracle's external behaviors through storage-level insights.
3. Advance your expertise to a new level in Oracle database management.
4. Enhance professional capabilities regardless of your specific role in database administration, development, or optimization.

This comprehensive exploration of storage architecture not only demystifies Oracle's internal operations but also serves as a critical foundation for optimizing database performance, troubleshooting complex issues, and designing efficient storage solutions.

Book Features

Each knowledge point begins with practical examples to provide an overview, followed by analysis of core structures, explanations of key fields, and insights into Oracle's operational principles, culminating in actionable conclusions and answers to common questions.

Complex logic is clarified through visual diagrams in challenging sections to enhance reader comprehension.

Limitations & Conventions

All examples are executed on Oracle 11.2.0.1, using an Intel x86-64 architecture and Linux OS. Results may vary in real-world environments, and readers should prioritize their own operational outcomes.

Foundational concepts are included for readers new to Oracle, advanced users may skip these sections.

Study method

Oracle provides a method to convert the binary content of data blocks into human-readable text information, making it easy to read. We use the dump command to generate a trace file containing the block's decoded metadata and data layers, then compare the textual output with the original binary block content to interpret fields like:

- Cache layer verification
- Transaction layer details
- Data layer organization

The dump command can be executed in SQL*Plus, with the command format being:

```
ALTER SYSTEM DUMP DATAFILE <fno> BLOCK <blkn>;
<fno>: Absolute file number
<blkn>: Block number
Both parameters can be extracted from a row's ROWID.
```

Another way is to use BBED to view the structure of a particular block. The Block Browser and Editor (BBED) enables low-level block manipulation, it is an internal tool of Oracle that can be used to view and edit data blocks directly, but it does not work for all types of blocks.

Chapter 1: Basic Concepts

Before starting to analyze the structure of the data file, there are some basic concepts that need to be clarified. Here are some of the concepts used in this book, or some of the concepts that are not covered here, and readers can search for the meaning of the concepts on the Internet.

Oracle Logical Structure of Storage

In Oracle's logical storage hierarchy from top to bottom, the storage is organized as tablespaces, segments, extents, and data blocks.

A **tablespace** is the highest-level storage structure. Every database object that requires data storage (such as tables, clusters, indexes, partitions, etc.) is assigned a tablespace upon creation. A tablespace can contain multiple data files, up to a maximum of 1023. Later, we will analyze why this limit is set to 1023 files.

A **segment** is the next lower-level storage structure. Each object requiring storage is allocated a segment, which resides entirely within a single tablespace. The segment's header location is determined by the data file number and block number. A segment consists of multiple extents, and its header block lists information about these extents, which may be contiguous or non-contiguous.

An **extent** is a lower-level storage structure composed of a set of contiguous data blocks. An extent is defined by its starting block address and the number of blocks it contains. Extents represent the smallest unit of space allocation in Oracle.

A **data block** is the smallest storage unit. Oracle data is distributed across data blocks, which come in different types for management or data storage purposes. A data block comprises one or more OS disk blocks. Oracle blocks are typically sized at 8K, with a maximum size of 32K.

Oracle Physical Structure of Storage

Oracle's physical storage structure consists of **data files**. Multiple data files form a tablespace, and a data file can belong to only one tablespace. Data files are composed of operating system (OS) blocks, regardless of whether the data files are OS files or ASM (Automatic Storage Management) files. Each data file has a file name,

and Oracle assigns a unique file number to each data file for management purposes. These files are managed via their assigned file numbers.

Oracle Storage Objects

In Oracle databases, the primary objects capable of storing data are tables, which can be categorized into heap-organized tables, clustered tables, and index-organized tables. LOBs (Large Objects) are used to store large data separately from tables, with two subtypes: BasicFile LOB and SecureFile LOB, each featuring distinct storage architectures and management mechanisms.

Both tables and LOBs can be subdivided into smaller units called partitions based on specific rules. Partitions may further divide into subpartitions, with each partition or subpartition allocated its dedicated segment for data storage.

In subsequent chapters, we will systematically analyze the storage structures of these objects in the following order: heap-organized tables, clustered tables, index-organized tables, and LOBs.

File Number

In Oracle, a data file is assigned **two file identifiers**:

Absolute File Number (AFN)

- **Globally unique** within the entire database.
- Assigned sequentially based on the order in which data files are added to the database.

Relative File Number (RFN)

- **Unique within its tablespace** and capped at a maximum value of **1023**.
- Introduced to bypass the legacy limitation of 1023 data files per database in earlier Oracle versions.
- When the total number of database files is ≤1023, the Absolute File Number and Relative File Number **match**.
- Exception for Bigfile Tablespaces:

The Relative File Number for a bigfile tablespace is always 1024 (or 4096 on OS/390 platforms).

Both identifiers often appear in Oracle's data dictionaries (e.g., DBA_DATA_FILES, V$DATAFILE) together. Understanding their meanings can help clarify the differences between them.

RDBA

RDBA stands for **R**elative **D**ata **B**lock **A**ddress, which is a 32-bit integer composed of a 10-bit relative file number and a 22-bit block number. For example, given an RDBA=0x01400f87, converting it to binary yields 0000 0001 0100 0000 0000 1111 1000 0111. Here, the first 10 bits (0000 0001 01) equal 0x05, and the remaining 22 bits (00 0000 0000 1111 1000 0111) equal 0x0f87 (3975 in decimal). This indicates that the block address is located in the file with relative file number 5, at block 3975.

The RDBA uses 10 bits to represent the relative file number, with a maximum value of 0x3ff (1023 in decimal). Since numbering starts from 1, this allows for a maximum of 1023 data files in a tablespace. Similarly, the 22 bits allocated for block numbering limit the maximum number of blocks. For a default block size of 8K, the maximum file size calculates to 0x3fffff * 8192 / 1024 / 1024 = 32767 MB = 32 GB. Thus, when using the default 8K block size, a data file is limited to 32 GB in size.

Bigfile Tablespace

Generally, a tablespace contains multiple data files to distribute I/O. However, when dealing with large datasets, the size limitations of individual files necessitate numerous files to meet storage requirements. This complicates the creation, management of data files, and overall administration of the tablespace. To address this, Oracle introduced **Bigfile Tablespaces** starting with Oracle 10g. The concept is straightforward: the 10 bits originally allocated for the relative file number in the RDBA are repurposed for block numbering. This allows the full 32 bits to represent block numbers, supporting up to 4 billion blocks (4G).

With a default block size of 8K, a single Bigfile data file can reach 32 TB (4G blocks * 8K = 32 TB). If using the maximum block size of 32K, a single data file can grow to 128 TB (4G blocks * 32K = 128 TB). Since the RDBA no longer encodes a relative file

number, a Bigfile Tablespace can contain only **one data file**, with its relative file number fixed at 1024. Due to bit overflow, this effectively represents a file number of 0, eliminating the need for a relative file number in the RDBA structure.

ROWID

The ROWID is the physical address of a row of data, enabling the fastest possible access to locate the data. A ROWID is composed of three components:

1. **DATA_OBJECT_ID** (Segment Storage ID): Identifies the database object (segment) to which the data belongs. The DATA_OBJECT_ID, rather than the OBJECT_ID, is stored in the data block to validate ownership.
2. **RDBA** (Relative Data Block Address): Specifies the physical location of the block containing the row. The RDBA includes the relative file number and block number, allowing precise identification of the file and block.
3. **SLOT** (Row Slot Number): Indicates the row's position within the block. Combined with the block's base address, the SLOT determines the offset where the row is stored within the block.

This structure ensures efficient navigation and validation of row storage in Oracle databases.

All rowids in the data file are stored in big-endian byte order.

SCN

SCN is an abbreviation for **S**ystem **C**hange **N**umber. Whenever any change occurs in the database, Oracle assigns an SCN to record the precise point in time of that change. The SCN is unique within the database and monotonically increases over time. When a transaction is committed, this SCN is referred to as the **commit SCN**. SCN tracks all changes (transactions, data modifications, structural changes) in the database and ensures data consistency, recovery, and synchronization.
Nearly all files in an Oracle database store SCN values, including data files, data block headers, control files, redo log files, etc., each maintaining their respective SCNs.

An SCN is a 6-byte numeric value divided into two components:

- **SCN_BASE** occupies 4 bytes
- **SCN_WRAP** occupies 2 bytes

When SCN_BASE reaches its maximum value (2^{32} - 1), SCN_WRAP increments by 1 and SCN_BASE resets to 0.

XID

XID (Transaction ID) is a unique identifier assigned by Oracle when a transaction begins. It functions not only as an identifier but also as a logical address. The XID comprises three components:

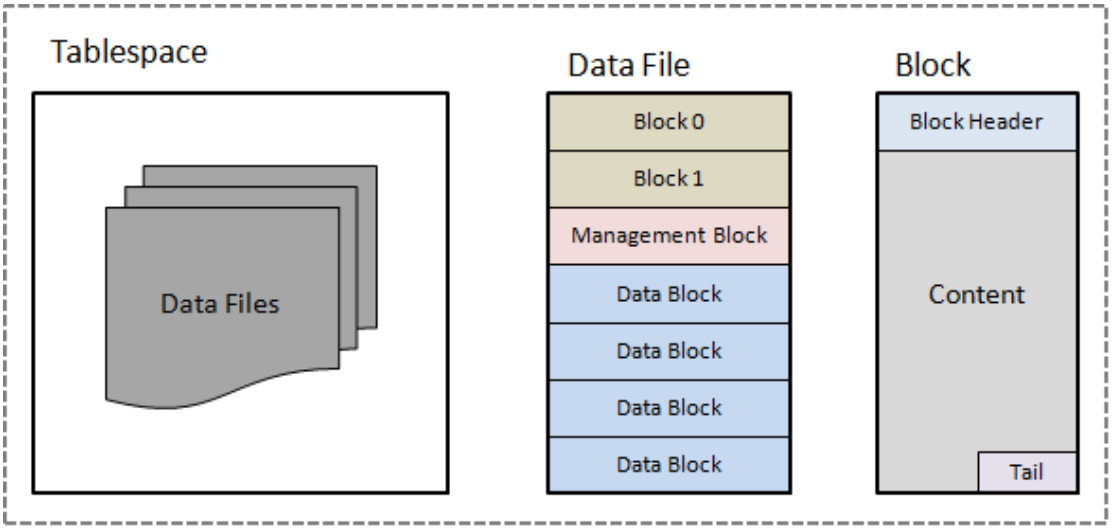
1. **usn** (Undo Segment Number): Identifies the undo segment.
2. **slt** (Slot Number): Specifies the transaction slot within the undo segment.
3. **sqn** (Sequence Number): Increments each time the transaction slot is reused.

Within data blocks, XIDs are prominently visible in the transaction layer.

Chapter 2: Datafile Structure

A tablespace is composed of multiple data files, and Oracle data is stored within these files. Data files consist of data blocks, which vary in structure depending on their type. The 0th and 1st blocks of a data file differ from other blocks, as they contain file header information. Each data block begins with a **block header**, ends with a **block trailer**, and contains the block's data in between, as illustrated in Figure 2-1.

Figure 2-1 Data File Composition



Block Header

Each block header in an Oracle data file has the same structure. Within the shared memory buffer, there is a **Cache Layer** component called the **Buffer Header**—a fixed-size structure totaling 20 bytes. It primarily includes information such as the **data block type**, the block's address (**RDBA**, Relative Data Block Address), the **SCN (System Change Number)** of the block modification, and a **checksum**, among other metadata.

The structure named **kcbh** (Kernel Cache Buffer Header) observed in BBED contains the following fields with their respective names and lengths:

struct kcbh, 20 bytes

```
ub1  type_kcbh;   /* block type * /
ub1  frmt_kcbh;
ub1  spare1_kcbh;
```

```
ub1  spare2_kcbh;
ub4  rdba_kcbh;   /* relative DBA /
ub4  bas_kcbh;    /* base of SCN */
ub2  wrp_kcbh;    /* wrap of SCN */
ub1  seq_kcbh;    /* sequence # of changes at same scn */
ub1  flg_kcbh;
ub2  chkval_kcbh;
ub2  spare3_kcbh;
```

The offsets of each field within the structure relative to the block header are as follows:

```
0x0000-0x0000  1 byte,  block type
0x0001-0x0001  1 byte,  format
0x0002-0x0002  1 byte,  spare1_kcbh
0x0003-0x0003  1 byte,  spare2_kcbh
0x0004-0x0007  4 bytes, rdba
0x0008-0x000B  4 bytes, SCN base
0x000C-0x000D  2 bytes, SCN wrap
0x000E-0x000E  1 byte,  seqence number
0x000F-0x000F  1 byte,  flag
0x0010-0x0011  2 bytes, check value
0x0012-0x0013  2 bytes, spare3_kcbh
```

Table 2-1 lists the description for each field in **kcbh** structure.

Table 2-1 Description for kcbh Fields

Field	Description
type_kcbh	Block type, the most common is 0x06 transaction data block, the full definition is listed in Table 2-2.
frmt_kcbh	Version format. Before Oracle 8i: 0x01 From Oracle 8i onward: 0x02 Starting in Oracle 10g, the high-order nibble (first half-byte) of the version field is used to indicate the block size: **2KB block: 0x62** **4KB block: 0x82** **8KB block: 0xA2** **16KB block: 0xC2** This encoding reflects the block size in hexadecimal notation, where the high-order nibble (e.g., 6, 8, A, C) corresponds to the block size

	identifier.
spare1_kcbh	The spare field is compatible with previous versions and is no longer used. It is always 0.
spare2_kcbh	The spare field is compatible with previous versions and is no longer used. It is always 0.
rdba_kcbh	Relative data block address, the DBA of this block.
bas_kcbh	The base part of the **System Change Number(SCN)**. When a **data block** is modified, the transaction records the **current SCN** in the block header.
wrp_kcbh	The wrap part of the **SCN**.
seq_kcbh	Sequence number, when the same SCN changes the block content many times, sequence is added to 1 each time.

flg_kcbh	Flag.
	`#define KCBHFNEW 0x01 /* new block - zeroed data area */`
	`#define KCBHFDLC 0x02 /* Delayed Logging Change advance SCN/seq */`
	`#define KCBHFCKV 0x04 /* ChecK Value saved-block xor's to zero */`
	`#define KCBHFTMP 0x08 /* Temporary block */`

chkval_kcbh	Checksum value. A **checksum value** is calculated by treating every **two-byte pair** (as a 16-bit integer) from the start of the block (excluding the checksum bytes themselves) and performing a cumulative **XOR operation** with subsequent values. This computed checksum is stored in the block and used to validate the **integrity of the block**.
spare3_kcbh	The spare field.

The type of block (type_kcbh) is defined as shown in Table 2-2.

Table 2-2 Block Type Definition

0x01 - KTU UNDO HEADER (undo segment header block)
0x02 - KTU UNDO BLOCK (undo data block)
0x03 - KTT SAVE UNDO HEADER
0x04 - KTT SAVE UNDO BLOCK
0x05 - DATA SEGMENT HEADER
0x06 - trans data
0x07 - Unknown
0x08 - Unknown
0x09 - Unknown
0x0A - DATA SEGMENT FREE LIST BLOCK
0x0B - data file header (block 1)
0x0C - DATA SEGMENT HEADER WITH FREE LIST BLOCKS

0x0D - Compatibility segment

0x0E - KTU UNDO HEADER W/UNLIMITED EXTENTS

0x0F - KTT SAVE UNDO HEADER W/UNLIMITED EXTENTS

0x10 - DATA SEGMENT HEADER - UNLIMITED

0x11 - DATA SEGMENT HEADER WITH FREE LIST BLKS - UNLIMITED

0x12 - EXTENT MAP BLOCK

0x13 - Unknown = rman file header block (block 1)

0x14 - Unknown = rman file directory block (block 2)

0x15 - Unknown = control file header block (block 1)

0x16 - 22 DATA SEGMENT FREE LIST BLOCK WITH FREE BLOCK COUNT

0x17 - BITMAPPED DATA SEGMENT HEADER

0x18 - BITMAPPED DATA SEGMENT FREELIST

0x19 - BITMAP INDEX BLOCK

0x1A - BITMAP BLOCK

0x1B - LOB BLOCK

0x1C - KTU BITMAP UNDO HEADER - LIMITED EXTENTS

0x1D - KTFB Bitmapped File Space Header

0x1E - KTFB Bitmapped File Space Bitmap

0x1F - TEMP INDEX BLOCK

0x20 - FIRST LEVEL BITMAP BLOCK

0x21 - SECOND LEVEL BITMAP BLOCK

0x22 - THIRD LEVEL BITMAP BLOCK

0x23 - PAGETABLE SEGMENT HEADER (BMB for ASSM)

0x24 - PAGETABLE EXTENT MAP BLOCK

0x25 - EXTENT MAP BLOCK OF SYSTEM MANAGED UNDO SEGMENT

0x26 - KTU SMU HEADER BLOCK

0x27 - Unknown

0x28 - PAGETABLE MANAGED LOB BLOCK

0x29 - Unknown

0x2A - Unknown

0x2B - Unknown

0x2C - Unknown

0x2D - Unknown

0x2E - Unknown

```
0x2F - Unknown
```

Block Tail

In the dump information of a block, there is always an entry called **tail**. A real-world example:

```
Block dump from disk:
buffer tsn: 6 rdba: 0x01400f87 (5/3975)
scn: 0x0000.00313140 seq: 0x01 flg: 0x06 tail: 0x31400601
frmt: 0x02 chkval: 0x2014 type: 0x06=trans data
```

The block tail information consists of four bytes, which are also used to verify the integrity of the block. By observing other details on the same line of the dumped information, you can see the composition of the block tail: the first two bytes represent the lower two bytes of the **SCN base**, followed by one byte for the flag (**flg**), and another byte for the sequence number (**seq**). The value of the block tail is formed by these three parts.

Block 0

Each data file's block 0 contains basic metadata, though its structure varies between Oracle versions. **Prior to Oracle 10g**, block 0 did not include the **Buffer Header** structure. In these older versions, the metadata starts at **offset 0x04**, with the first four bytes reserved for **padding**. **In Oracle 10g and later**, the metadata begins after the Buffer Header (at **offset 0x14**). Specifically, it consists of three main parts as follows:

```
ub4  block0_size;
ub4  blocks_in_file;
ub1  platform_id[4];
```

The meanings of the fields in the structure of block 0 are listed in Table 2-3.

Table 2-3 Meanings of the Fields in Block 0

Field	Description
Block0_size	The size of block 0, may differ from the size of other blocks in the file. The actual block size used for the file is defined in block 1.
blocks_in_file	The number of blocks in the data file (excluding block 0), **when multiplied by** the block size specified in block 1 **and added to** the size of block 0, **equals** the complete size of the data file.
platform_id	The platform identifier for Oracle Server, it is defined as follows: **Prior to Oracle 10g**: UNIX platforms used the identifier 5A5B5C5D. Windows platforms used 6A6B6C6D. **From Oracle 10g onwards**: The identifier 7A7B7C7D is used uniformly, regardless of platform (UNIX or Windows). This identifier is additionally used to determine the byte order: If the identifier is stored as **0x7A7B7C7D**, it indicates **big-endian** byte order. If the identifier is stored as **0x7D7C7B7A**, it indicates **little-endian** byte order.

Let's create a new datafile to examine its structure. In SQL*Plus, execute the following command (requires DBA privileges):

```
CREATE TABLESPACE study DATAFILE 'study_01.dbf' SIZE 200M AUTOEXTEND ON;
```

After executing the command, Oracle creates the new file **study_01.dbf** under $ORACLE_HOME/dbs directory.

```
-rw-r----- 1 oracle oinstall 209723392 Nov  2 16:09 study_01.dbf
```

To query the file number and path of datafiles, use the following SQL statement:

```
SELECT file_id, relative_fno, file_name
FROM dba_data_files WHERE tablespace_name='STUDY';

   FILE_ID RELATIVE_FNO FILE_NAME
---------- ------------ ----------------------------------------
         8            8 /oracle11/product/11.2.0/dbs/study_01.dbf
```

The binary content of block 0 in the datafile is shown below:

```
----------  +0-+1-+2-+3-+4-+5-+6-+7-+8-+9-+A-+B-+C-+D-+E-+F-    0123456789ABCDEF
0x00000000  00 A2 00 00 00 00 C0 FF-00 00 00 00 00 00 00 00    ................
            Buffer Header
0x00000010  66 9E 00 00 00 20 00 00-00 64 00 00 7D 7C 7B 7A    f.... ...d..}|{z
                        |           |           platform_id
                        |           blocks_in_file
                        block0_size
0x00000020  A0 81 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
```

The hexadecimal value 0x7D7C7B7A in the platform_id field indicates that the data storage follows the **little-endian** byte order format. Then

block0_size = 0x00002000 = 8192 bytes

blocks_in_file = 0x00006400 = 25600 blocks

The block size in the datafile is **8192 bytes** (as observed when analyzing block 1), so the total file size can be calculated as:

file_size = 25600 * 8192 = 209715200 = 200M bytes

The file size exactly matches the **SIZE 200M** specified in the command, plus the **8192-byte** size of block 0, the file size in the Operating System is:

os_file_size = 209715200 + 8192 = 209723392 bytes

This is **exactly the physical file size** displayed by the OS.

Block 1

The **first block (Block 1)** of each datafile contains critical database-related information. This block cannot be extracted as text via the dump command. However, you can use the following statement to dump the file header information into a trace file. By performing a binary comparison with Block 1, you can analyze its contents. Here is the dump statement:

```
ALTER SESSION SET EVENTS 'IMMEDIATE TRACE NAME FILE_HDRS LEVEL 10';
```

The trace file content is shown below. Repetitive "0" data has been omitted for clarity.

```
DATA FILE #8:
  name #12: /oracle11/product/11.2.0/dbs/study_01.dbf
creation size=25600 block size=8192 status=0xe head=12 tail=12 dup=1
 tablespace 9, index=9 krfil=8 prev_file=0
 unrecoverable scn: 0x0000.00000000 01/01/1988 00:00:00
 Checkpoint cnt:11 scn: 0x0000.003f34a1 11/04/2024 23:15:50
 Stop scn: 0xffff.ffffffff 11/03/2024 23:49:55
 Creation Checkpointed at scn:  0x0000.003ebede 11/01/2024 22:13:24
 thread:1 rba:(0xb7.8f0.10)
 enabled  threads:  01000000 00000000 00000000 00000000 00000000 00000000
                    Repeat 120 times
  00000000 00000000 00000000 00000000 00000000 00000000
 Offline scn: 0x0000.00000000 prev_range: 0
 Online Checkpointed at scn:  0x0000.00000000
 thread:0 rba:(0x0.0.0)
 enabled  threads:  00000000 00000000 00000000 00000000 00000000 00000000
                    Repeat 120 times
  00000000 00000000 00000000 00000000 00000000 00000000
 Hot Backup end marker scn: 0x0000.00000000
 aux_file is NOT DEFINED
Plugged readony: NO
 Plugin scnscn: 0x0000.00000000
 Plugin resetlogs scn/timescn: 0x0000.00000000 01/01/1988 00:00:00
 Foreign creation scn/timescn: 0x0000.00000000 01/01/1988 00:00:00
 Foreign checkpoint scn/timescn: 0x0000.00000000 01/01/1988 00:00:00
 Online move state: 0
 V10 STYLE FILE HEADER:
        Compatibility Vsn = 186646528=0xb200000
        Db ID=139822064=0x85583f0, Db Name='ORA11G'
        Activation ID=0=0x0
        Control Seq=20370=0x4f92, File size=25600=0x6400
        File Number=8, Blksiz=8192, File Type=3 DATA
Tablespace #9 - STUDY  rel_fn:8
Creation   at   scn: 0x0000.003ebede 11/01/2024 22:13:24
Backup taken at scn: 0x0000.00000000 01/01/1988 00:00:00 thread:0
 reset logs count:0x3de79d33 scn: 0x0000.000e6c20
 prev reset logs count:0x296b946b scn: 0x0000.00000001
 recovered at 01/01/1988 00:00:00
 status:0x4 root dba:0x00000000 chkpt cnt: 11 ctl cnt:10
begin-hot-backup file size: 0
```

```
Checkpointed at scn:  0x0000.003f34a1 11/04/2024 23:15:50
 thread:1 rba:(0xb9.2.10)
 enabled  threads:  01000000 00000000 00000000 00000000 00000000 00000000
                    Repeat 120 times
  00000000 00000000 00000000 00000000 00000000 00000000
Backup Checkpointed at scn:  0x0000.00000000
thread:0 rba:(0x0.0.0)
 enabled  threads:  00000000 00000000 00000000 00000000 00000000 00000000
                    Repeat 120 times
  00000000 00000000 00000000 00000000 00000000 00000000
External cache id: 0x0 0x0 0x0 0x0
Absolute fuzzy scn: 0x0000.00000000
Recovery fuzzy scn: 0x0000.00000000 01/01/1988 00:00:00
Terminal Recovery Stamp  01/01/1988 00:00:00
Platform Information:    Creation Platform ID: 13
Current Platform ID: 13 Last Platform ID: 13
```

The information above appears disorganized. For precise analysis, you can use BBED to inspect the file header details, enter BBED and execute the map command.

```
BBED> map
 File: /oracle11/product/11.2.0/dbs/study_01.dbf (8)
 Block: 1                                        Dba:0x02000001
------------------------------------------------------------
 Data File Header

 struct kcvfh, 860 bytes                    @0

 ub4 tailchk                                @8188
```

Here, we observe that **Block 1** begins with an **860-byte structure** named **kcvfh**, which holds file header information. The last **4 bytes** correspond to the **block tail information**. To examine the detailed structure of kcvfh, use the print command:

```
BBED> print kcvfh
struct kcvfh, 860 bytes                    @0
   struct kcvfhbfh, 20 bytes               @0
      ub1 type_kcbh                        @0        0x0b
      ub1 frmt_kcbh                        @1        0xa2
      ub1 spare1_kcbh                      @2        0x00
      ub1 spare2_kcbh                      @3        0x00
```

```
      ub4 rdba_kcbh                          @4        0x02000001
      ub4 bas_kcbh                           @8        0x00000000
      ub2 wrp_kcbh                           @12       0x0000
      ub1 seq_kcbh                           @14       0x01
      ub1 flg_kcbh                           @15       0x04 (KCBHFCKV)
      ub2 chkval_kcbh                        @16       0xbc69
      ub2 spare3_kcbh                        @18       0x0000
   struct kcvfhhdr, 76 bytes                 @20
      ub4 kccfhswv                           @20       0x00000000
      ub4 kccfhcvn                           @24       0x0b200000
      ub4 kccfhdbi                           @28       0x085583f0
      text kccfhdbn[0]                       @32       O
      text kccfhdbn[1]                       @33       R
      text kccfhdbn[2]                       @34       A
      text kccfhdbn[3]                       @35       1
      text kccfhdbn[4]                       @36       1
      text kccfhdbn[5]                       @37       G
      text kccfhdbn[6]                       @38
      text kccfhdbn[7]                       @39
      ... ...
```

Let's review the binary data of **Block 1**. The printed text output is as follows:

```
----------  +0-+1-+2-+3-+4-+5-+6-+7-+8-+9-+A-+B-+C-+D-+E-+F-    0123456789ABCDEF
0x00000000  0B A2 00 00 01 00 00 02-00 00 00 00 00 00 01 04    ................
            Buffer Header
0x00000010  69 BC 00 00 00 00 00 00-00 00 20 0B F0 83 55 08    i........ ...U.
                                    cvn=11.2    db id
0x00000020  4F 52 41 31 31 47 00 00-30 4F 00 00 00 64 00 00    ORA11G..0O...d..
            db name                             blocks count
0x00000030  00 20 00 00 08 00 03 00-00 00 00 00 00 00 00 00    . ..............
            block size  file number
0x00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
0x00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
0x00000060  00 00 00 00 DE BE 3E 00-00 00 00 00 84 61 91 46    ......>......a.F
            root dba
0x00000070  33 9D E7 3D 20 6C 0E 00-00 00 00 00 00 00 00 00    3..= l..........
0x00000080  00 00 00 00 00 00 00 00-00 00 04 00 07 00 00 00    ................
0x00000090  00 00 00 00 06 00 00 00-00 00 00 00 00 00 00 00    ................
0x000000A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
0x000000B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
0x000000C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
```

```
0x000000D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x000000E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x000000F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x00000100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x00000110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x00000120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x00000130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x00000140  00 00 00 00 00 00 00 00-00 00 00 00 09 00 00 00     ................
                                                tablespace number
0x00000150  05 00 53 54 55 44 59 00-00 00 00 00 00 00 00 00     ..STUDY.........
                  tablespace name
0x00000160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x00000170  08 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
            relative file number
0x00000180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x00000190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x000001A0  6B 94 6B 29 01 00 00 00-00 00 00 00 00 00 00 00     k.k).............
0x000001B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x000001C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x000001D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
0x000001E0  00 00 00 00 84 F3 3E 00-00 00 00 00 54 EB 93 46     ......>.....T..F
0x000001F0  01 00 00 00 B8 00 00 00-6A 18 00 00 10 00 00 00     ........j.......

----------  +0-+1-+2-+3-+4-+5-+6-+7-+8-+9-+A-+B-+C-+D-+E-+F-    0123456789ABCDEF
0x00000200  02 00 00 00 00 00 00 00
```

By examining the **binary data of Block 1** in detail, the specific **field positions** can be precisely mapped. The **field offsets** for **Block 1** in an Oracle 11.2.0.1 data file are shown below:

```
0x0000-0x0013  Buffer Header(common structure, 20 bytes)
0x0014-0x0017  db version
0x0018-0x001B  compatible version
0x001C-0x001F  db id
0x0020-0x0027  db name
0x0028-0x002B  control sequence
0x002C-0x002F  blocks count
0x0030-0x0033  block size
0x0034-0x0035  file number
0x0036-0x0037  file type
0x0038-0x003B  activation id
0x003C-0x003F  cks (kccfhcks)
0x0040-0x005F  tag
```

```
0x0060-0x0063  root dba
0x0064-0x0067  creation checkpoint SCN base
0x0068-0x0069  creation checkpoint SCN wrap
0x006A-0x006B  padding bytes[2]
0x006C-0x006F  creation checkpoint SCN time
0x0070-0x0073  reset logs count
0x0074-0x0077  reset logs SCN base
0x0078-0x0079  reset logs SCN wrap
0x007A-0x007B  padding bytes[2]
0x007C-0x007F  begin backup time
0x0080-0x0083  begin backup SCN base
0x0084-0x0085  begin backup SCN wrap
0x0086-0x0087  padding bytes[2]
0x0088-0x0089  begin backup thread#
0x008A-0x008B  file status
0x008C-0x008F  checkpoint count
0x0090-0x0093  recovered time
0x0094-0x0097  control count
0x0098-0x009B  backup checkpoint SCN base
0x009C-0x009D  backup checkpoint SCN wrap
0x009E-0x009F  padding bytes[2]
0x00A0-0x00A3  backup checkpoint time
0x00A4-0x00A5  backup thread#
0x00A6-0x00A7  padding bytes[2]
0x00A8-0x00AB  backup rba.sequence#
0x00AC-0x00AF  backup rba.block#
0x00B0-0x00B1  backup rba.offset
0x00B2-0x00B3  padding bytes[2]
0x00B4-0x00BB  enabled threads
0x00BC-0x0137  unknown
0x0138-0x013B  bhz (kcvfhbhz)
0x013C-0x013F  xcd[0] (space_kcvmxcd)
0x0140-0x0143  xcd[1]
0x0144-0x0147  xcd[2]
0x0148-0x014B  xcd[3]
0x014C-0x014F  tablespace number
0x0150-0x0151  tablespace name length
0x0152-0x016F  tablespace name 30 bytes
0x0170-0x0173  relative file number
0x0174-0x0177  recovery fuzzy SCN base
0x0178-0x0179  recovery fuzzy SCN wrap
0x017A-0x017B  padding bytes[2]
0x017C-0x017F  recovery fuzzy time
0x0180-0x0183  absolute fuzzy SCN base
```

```
0x0184-0x0185  absolute fuzzy SCN wrap
0x0186-0x0187  padding bytes[2]
0x0188-0x018B  bbc (kcvfhbbc)
0x018C-0x018F  ncb (kcvfhncb)
0x0190-0x0193  mcb (kcvfhmcb)
0x0194-0x0197  lcb (kcvfhlcb)
0x0198-0x019B  bcs (kcvfhbcs)
0x019C-0x019D  ofb (kcvfhofb)
0x019E-0x019F  nfb (kcvfhnfb)
0x01A0-0x01A3  prev reset logs count
0x01A4-0x01A7  prev reset logs SCN base
0x01A8-0x01A9  prev reset logs SCN wrap
0x01AA-0x01AB  padding bytes[2]
0x01AC-0x01AF  prfs (kcvfhprfs) SCN base
0x01B0-0x01B1  prfs (kcvfhprfs) SCN wrap
0x01B2-0x01BB  unknown
0x01BC-0x01BF  trt (kcvfhtrt)
0x01C0-0x01E3  unknown
0x01E4-0x01E7  checkpoint SCN base
0x01E8-0x01E9  checkpoint SCN wrap
0x01EA-0x01EB  padding bytes[2]
0x01EC-0x01EF  checkpoint time
0x01F0-0x01F1  checkpoint thread#
0x01F2-0x01F3  padding bytes[2]
0x01F4-0x01F7  checkpoint rba.sequence
0x01F8-0x01FB  checkpoint rba.block
0x01FC-0x01FD  checkpoint rba.offset
0x01FE-0x01FF  padding bytes[2]
0x0200-0x0207  etb[8] (kcvcpetb) enabled threads
```

Block 1 contains extensive metadata (e.g., creation, backup, recovery, and resetlogs), which are scenario-specific and can be temporarily ignored. Below, we will focus exclusively on **data storage-related fields** and their meanings, as summarized in **Table 2-4**.

Table 2-4 Structure kcvfhhdr Fields (storage related)

Field	Description
compatible version @（0x0018-0x001B）	Compatibility Version Number: In this example, 0x0b200000 corresponds to Oracle 11.2.
db id @（0x001C-0x001F）	Database Identifier (DBID): A unique value assigned to a database instance, used to verify whether a file belongs to

	that specific database.
db name @（0x0020-0x0027）	Database name to which the file belongs.
blocks count @（0x002C-0x002F）	Number of blocks in the file, excluding Block 0.
blocks size @（0x0030-0x0033）	Block size in the file, which may differ from Block 0.
file number @（0x0034-0x0035）	Absolute file number.
file type @（0x0036-0x0037）	File type: 3 for data file, 2 for log file.
root dba @（0x0060-0x0063）	This field is only populated in File 1 of the SYSTEM tablespace, all others are set to 0. Oracle uses this address to locate the data dictionary bootstrap entry, which points to the bootstrap$ table.
ts number @（0x014C-0x014F）	**Tablespace number**, corresponding to the ts# column in the base table **TS$**, identifies which tablespace this file belongs to.
ts name length @（0x0150-0x0151）	Tablespace name length: The tablespace name can be up to 30 characters long.
ts name @（0x0152-0x016F）	Tablespace name.
relative fno @（0x0170-0x0173）	**Relative file number**, unique within a tablespace.

Chapter 3: Data Block Structure of Heap-Organized Tables

We create a user TOM assigned to the predefined tablespace STUDY, then create a table TEST_TAB1 within this user schema. By inserting data into the table, we can investigate the physical structure of Oracle data blocks.

```
CREATE USER tom IDENTIFIED BY tom DEFAULT TABLESPACE study;
GRANT CONNECT, RESOURCE TO tom;
```

The preceding commands created a database user *tom* (password: *tom*) and created a heap-organized table within the TOM user's schema using the following SQL statement:

```
CONN tom/tom;
CREATE TABLE test_tab1
(
    id    number PRIMARY KEY,
    fld1  char(30),
    fld2  varchar2(2000),
    fld3  varchar2(4000),
    fld4  varchar2(4000)
);
```

The commands above have created a heap-organized table, and the data will be stored in the **STUDY tablespace**. Since the tablespace contains only a single data file, the data will reside in **study_01.dbf**.

Now insert a row of data into the table using the following SQL statement:

```
INSERT INTO test_tab1 VALUES
(1, '1aaaaaaa', '1bbbbbbbbb', '1ccccccccccc', '1ddddddddddddd');
COMMIT;
```

Execute the following command to flush the data from the buffer cache to the data files.

```
ALTER SYSTEM checkpoint;
```

Use the following SQL statement to query the block number and row number of the row data.

```
SELECT
    id,
    dbms_rowid.rowid_relative_fno(rowid) rfn,
    dbms_rowid.rowid_block_number(rowid) block#,
    dbms_rowid.rowid_row_number(rowid) row#
FROM test_tab1;

        ID        RFN     BLOCK#       ROW#
---------- ---------- ---------- ----------
         1          8        135          0
```

Based on the query results, the row we inserted resides in **block 135**, **row 0** of the **8th** file (study_01.dbf). Since the total number of files in our database is fewer than 1023, the **absolute file number** and **relative file number** are identical. Use the following command to dump this block for inspection:

```
ALTER SYSTEM DUMP datafile 8 block 135;
```

Below is the complete dump information of the data block.

```
Start dump data blocks tsn: 9 file#:8 minblk 135 maxblk 135
Block dump from cache:
Dump of buffer cache at level 4 for tsn=9, rdba=33554567
BH (0x9efe2d78) file#: 8 rdba: 0x02000087 (8/135) class: 1 ba: 0x9ed26000
  set: 3 pool 3 bsz: 8192 bsi: 0 sflg: 2 pwc: 140,28
  dbwrid: 0 obj: 78733 objn: 78733 tsn: 9 afn: 8 hint: f
  hash: [0xc0305210,0xc0305210] lru: [0x9efe2f90,0x9eff7fa0]
  ckptq: [NULL] fileq: [NULL] objq: [0x9efe2fb8,0xb918e150]
  st: XCURRENT md: NULL tch: 2
  flags: block_written_once redo_since_read
  LRBA: [0x0.0.0] LSCN: [0x0.0] HSCN: [0xffff.ffffffff] HSUB: [1]
  cr pin refcnt: 0 sh pin refcnt: 0
Block dump from disk:
buffer tsn: 9 rdba: 0x02000087 (8/135)
scn: 0x0000.003f2481 seq: 0x01 flg: 0x06 tail: 0x24810601
frmt: 0x02 chkval: 0x70f9 type: 0x06=trans data
Hex dump of block: st=0, typ_found=1
```

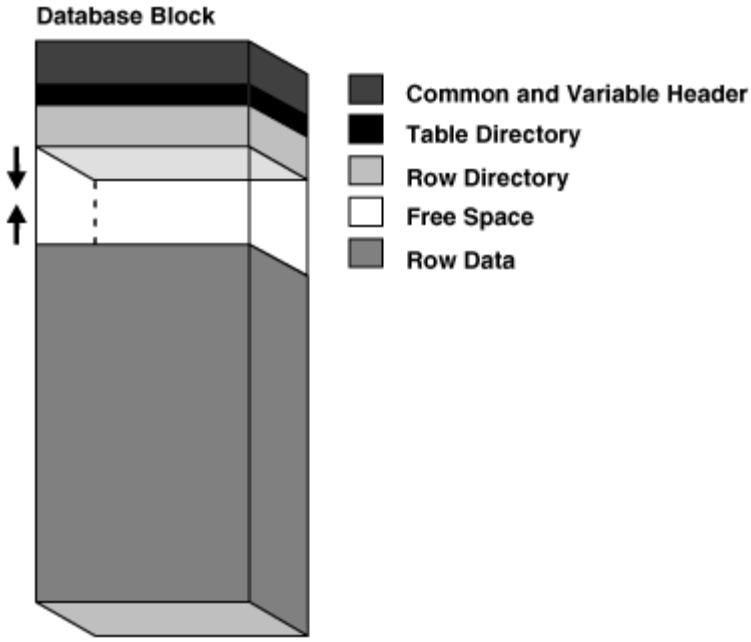
```
Dump of memory from 0x00002B57AACE6A00 to 0x00002B57AACE8A00
2B57AACE6A00 0000A206 02000087 003F2481 06010000  [.........$?.....]
2B57AACE6A10 000070F9 00000001 0001338D 003F2480  [.p.......3...$?.]
2B57AACE6A20 00000000 00320002 02000080 001C0009  [......2.........]
2B57AACE6A30 00000D6B 00C010D0 002C0309 00002001  [k.........,..  ..]
2B57AACE6A40 003F2481 00000000 00000000 00000000  [.$?.............]
2B57AACE6A50 00000000 00000000 00000000 00000000  [................]
2B57AACE6A60 00000000 00010100 0014FFFF 1F381F4C  [............L.8.]
2B57AACE6A70 00001F38 1F4C0001 00000000 00000000  [8.....L.........]
2B57AACE6A80 00000000 00000000 00000000 00000000  [................]
        Repeat 498 times
2B57AACE89B0 0205012C 311E02C1 61616161 20616161  [,......1aaaaaaa ]
2B57AACE89C0 20202020 20202020 20202020 20202020  [                ]
2B57AACE89D0 20202020 62310A20 62626262 62626262  [     .1bbbbbbbbb]
2B57AACE89E0 6363310C 63636363 63636363 64310E63  [.1ccccccccccc.1d]
2B57AACE89F0 64646464 64646464 64646464 24810601  [ddddddddddddd...$]
Block header dump:  0x02000087
 Object id on Block? Y
 seg/obj: 0x1338d  csc: 0x00.3f2480  itc: 2  flg: E  typ: 1 - DATA
     brn: 0  bdba: 0x2000080 ver: 0x01 opc: 0
     inc: 0  exflg: 0

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0009.01c.00000d6b  0x00c010d0.0309.2c  --U-    1  fsc 0x0000.003f2481
0x02   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
bdba: 0x02000087
data_block_dump,data header at 0x2b57aace6a64
==============
tsiz: 0x1f98
hsiz: 0x14
pbl: 0x2b57aace6a64
     76543210
flag=--------
ntab=1
nrow=1
frre=-1
fsbo=0x14
fseo=0x1f4c
avsp=0x1f38
tosp=0x1f38
0xe:pti[0]      nrow=1  offs=0
0x12:pri[0]     offs=0x1f4c
block_row_dump:
tab 0, row 0, @0x1f4c
```

```
tl: 76 fb: --H-FL-- lb: 0x1  cc: 5
col  0: [ 2]  c1 02
col  1: [30]
 31 61 61 61 61 61 61 61 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 20 20 20 20 20
col  2: [10]  31 62 62 62 62 62 62 62 62 62
col  3: [12]  31 63 63 63 63 63 63 63 63 63 63 63
col  4: [14]  31 64 64 64 64 64 64 64 64 64 64 64 64 64
end_of_block_dump
End dump data blocks tsn: 9 file#: 8 minblk 135 maxblk 135
```

Before studying the data block structure, let's first understand its layers. Oracle's official documentation provides a general structural diagram, as shown in Figure 3-1.

Figure 3-1 Database Block Layers



From the diagram above, we observe that the data block begins with a **header structure**, composed of **fixed-length and variable-length layers**. This is followed by the **table directory** and **row directory** structures. The **actual row data** resides at the end of the block, while **free space** occupies the middle. Notably, Oracle stores row data **from the bottom of the block upward** (toward the header), while the row directory grows **from the top downward**. When the remaining free space reaches the threshold defined by **PCTFREE**, the block is no longer eligible for new inserts.

The text dump above still appears insufficiently clear. Let’s use **BBED** (Block Browser

and Editor) to examine the block structure in detail.

```
BBED> map
 File: /oracle11/product/11.2.0/dbs/study_01.dbf (8)
 Block: 135                                      Dba:0x02000087
------------------------------------------------------------
 KTB Data Block (Table/Cluster)

 struct kcbh, 20 bytes                           @0

 struct ktbbh, 72 bytes                          @20

 struct kdbh, 14 bytes                           @100

 struct kdbt[1], 4 bytes                         @114

 sb2 kdbr[1]                                     @118

 ub1 freespace[7992]                             @120

 ub1 rowdata[76]                                 @8112

 ub4 tailchk                                     @8188
```

From the above description, we observe the block structure as follows:

1. **Buffer Header (kcbh)**
   - **20-byte** structure.
   - **k** = Kernel (core structure), **c** = Cache, **b** = Buffer, **h** = Header.
2. **Transaction Header (ktbbh)**
   - **72-byte** section storing transactional metadata (e.g., undo information, locks).
3. **Data Header (kdbh)**
   - **14-byte** structure containing data block-specific metadata (e.g., row count, free space pointers).
4. **Table Directory (kdbt[])**
   - A **struct array** mapping tables to rows within the block.
5. **Row Directory (kdbr[])**

- A **16-bit integer array** where each element represents a row offset.
- The array length equals the number of rows in the block.

6. **Free Space (freespace)**
   - Unallocated region reserved for future inserts/updates.
7. **Row Data (rowdata)**
   - Actual storage area for row contents.
8. **Block Trailer**
   - **4-byte** checksum or validation marker at the block's end.

Let's examine the **hexadecimal dump** of the block to explicitly observe its structural layout.

```
----------  +0-+1-+2-+3-+4-+5-+6-+7-+8-+9-+A-+B-+C-+D-+E-+F-     0123456789ABCDEF
0x00000000  06 A2 00 00 87 00 00 02-81 24 3F 00 00 00 01 06      .........$?.....
            kcbh
0x00000010  F9 70 00 00  01 00 00 00-8D 33 01 00 80 24 3F 00      .p.......3...$?.
                         ktbbh
0x00000020  00 00 00 00 02 00 32 00-80 00 00 02 09 00 1C 00      ......2.........
0x00000030  6B 0D 00 00 D0 10 C0 00-09 03 2C 00 01 20 00 00      k.........,.. ..
0x00000040  81 24 3F 00 00 00 00 00-00 00 00 00 00 00 00 00      .$?.............
0x00000050  00 00 00 00 00 00 00 00-00 00 00 00  00 00 00 00      ................
                                                 additional data
0x00000060  00 00 00 00  00 01 01 00-FF FF 14 00 4C 1F 38 1F      ............L.8.
                         kdbh
0x00000070  38 1F  00 00 01 00  4C 1F-00 00 00 00 00 00 00 00      8.....L.........
                   kdbt         kdbr  freespace
..............................  omitted zero data  ..............................
0x00001FA0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00      ................
0x00001FB0  2C 01 05 02 C1 02 1E 31-61 61 61 61 61 61 61 20      ,......1aaaaaaa
            rowdata
0x00001FC0  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
0x00001FD0  20 20 20 20 20 0A 31 62-62 62 62 62 62 62 62 62            .1bbbbbbbbb
0x00001FE0  0C 31 63 63 63 63 63 63-63 63 63 63 63 0E 31 64      .1ccccccccccc.1d
0x00001FF0  64 64 64 64 64 64 64 64-64 64 64 64  01 06 81 24      dddddddddddd...$
                                                 tailchk
```

In the following subsections, we will dissect the data block **from top to bottom**,

exploring the specifics of each structural component.

Buffer Header & tail

The first component is the **Buffer Header (kcbh)**, a 20-byte structure present in every data block, whose layout we have previously analyzed. Examining its hex dump, the block type is identified as **"trans data"** (transactional data).

```
buffer tsn: 9 rdba: 0x02000087 (8/135)
scn: 0x0000.003f2481 seq: 0x01 flg: 0x06 tail: 0x24810601
    frmt: 0x02 chkval: 0x70f9 type: 0x06=trans data
```

The block also includes **4 bytes at its tail** (end), which we previously dissected: these comprise the **lower 2 bytes of the SCN base**, followed by the **FLG** (flags) byte, and the **SEQ** (sequence) byte.

Block header

Following the **KCBH** is the **block header structure** from the hex dump, with the content listed below:

```
Block header dump:  0x02000087
 Object id on Block? Y
 seg/obj: 0x1338d  csc: 0x00.3f2480  itc: 2  flg: E  typ: 1 - DATA
     brn: 0  bdba: 0x2000080 ver: 0x01 opc: 0
     inc: 0  exflg: 0

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0009.01c.00000d6b  0x00c010d0.0309.2c  --U-    1  fsc 0x0000.003f2481
0x02   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
```

The block header structure corresponds to the **common and variable header** shown in Figure 3-1. It contains transactional information and is subdivided into a **fixed-size portion** and a **variable portion**. Tools like BBED (Block Browser and Editor) can reveal detailed insights into these structural variations.

```
struct ktbbh, 72 bytes                    @20
    ub1 ktbbhtyp                          @20
    union ktbbhsid, 4 bytes               @24
```

```
struct ktbbhcsc, 8 bytes                @28
sb2 ktbbhict                            @36
ub1 ktbbhflg                            @38
ub1 ktbbhfsl                            @39
ub4 ktbbhfnx                            @40
struct ktbbhitl[2], 48 bytes            @44
```

Let's first examine the **fixed portion**, which spans **24 bytes**:

```
0x0000-0x0000  1 byte,  type
0x0001-0x0003  3 bytes, padding bytes[3]
0x0004-0x0007  4 bytes, dataobj#
0x0008-0x000B  4 bytes, csc_base
0x000C-0x000D  2 bytes, csc_wrap
0x000E-0x000F  2 bytes, padding
0x0010-0x0011  2 bytes, itc
0x0012-0x0012  1 byte,  flg
0x0013-0x0013  1 byte,  fsl
0x0014-0x0017  4 bytes, fnx
```

This section describes the **common header**, which occupies **24 bytes**. The meanings of each field are detailed in **Table 3-1** below.

Table 3-1 Common Header Field Meanings

Field	Description
type	Data Type: 01-data, 02-index, 05-local lobs.
dataobj#	seg/obj, object number in segment.
csc_base	Cleanout SCN base.
csc_wrap	Cleanout SCN wrap.
itc	The **number of ITL entries** and the **size of the ITL structure** are fixed. The **ITC (Transaction Slot Count)** determines the size of the variable portion of the transaction layer. Since **ITC ≤ 255**, the high-order bits should be masked off (ITC & 0xFF) after retrieving the value.
flg	Flags. The **meanings of the flag bits** will be explained in

	detail later, as shown in **Table 3-3**.
fsl	Free space lock.
fnx	A pointer to the next block on the free list chain.

The subsequent section is the **variable portion of the transaction layer**, known as the **ITL (Interested Transaction List)**. The number of **ITL slots** is determined by the **ITC (Transaction Slot Count)** mentioned earlier. By default, there are **2 ITL slots**. The ITL is an integral component of Oracle data blocks, recording transactions affecting the block. Each **ITL slot** (also called an ITL entry) acts as a transaction record. In BBED, ITL slots are explicitly named **ktbbhitl** and occupy **24 bytes each**.

```
struct ktbbhitl, 24 bytes                  @44
  struct ktbitxid, 8 bytes                 @44
    ub2 kxidusn                            @44
    ub2 kxidslt                            @46
    ub4 kxidsqn                            @48
  struct ktbituba, 8 bytes                 @52
    ub4 kubadba                            @52
    ub2 kubaseq                            @56
    ub1 kubarec                            @58
  ub2 ktbitflg                             @60
  union _ktbitun, 2 bytes                  @62
    sb2 _ktbitfsc                          @62
    ub2 _ktbitwrp                          @62
  ub4 ktbitbas                             @64
```

The **ITL structure** is as follows:

```
0x0000-0x0001  2 bytes, xid.usn
0x0002-0x0003  2 bytes, xid.slt
0x0004-0x0007  4 bytes, xid.sqn
0x0008-0x000B  4 bytes, uba.dba
0x000C-0x000D  2 bytes, uba.seq
0x000E-0x000E  1 byte,  uba.rec
0x000F-0x000F  1 byte,  padding
0x0010-0x0011  2 bytes, flag
0x0012-0x0013  2 bytes, fsc/scn wrap
0x0014-0x0017  4 bytes, fsc/scn base
```

Table 3-2 below explains the fields of the **ITL (Interested Transaction List)**, with each ITL slot occupying **24 bytes**.

Table 3-2 ITL Fields Description

Field	Description
xid	Transaction ID, **unique identifier** for the transaction. It is composed of undo segment number (**usn**), slot number (**slt**), and sequence number (**sqn**).
uba	The **block address of an Undo segment** (referred to as the **UBA** or **Undo Block Address**) is assigned to each transaction upon initiation. A UBA consists of three components: **Block Number (DBA)**: The physical address of the Undo block. **Sequence Number (SEQ)**: A unique identifier for the Undo record version. **Record Number (REC)**: The specific slot within the Undo block.
flag	In the **ITL structure**, the **Flags** field shares 2 bytes with **lock information**: **Flags** occupy the **upper 4 bits** (high-order nibble). **Lock** occupies the **lower 12 bits** (low-order 3 nibbles). Flags(4 bits): ---- = transaction is active or committed pending cleanout C--- = transaction has been committed and locks cleaned out -B-- = this undo record contains the undo for this ITL entry --U- = transaction committed (maybe long ago); SCN is an upper bound ---T = transaction was still active at block cleanout SCN Lock(12 bites): The **lock** field specifies the number of **row-level locks** held by the transaction in this block. In this example, **flag=0x2001**: **High-order nibble (4 bits)**: 0x2 (hex 0x2, binary 0010), represented as **U**. **Remaining 3 nibbles (12 bits)**: 0x001 (hex 0x001, binary 0000 0000 0001), representing **Lck=1** (1 row locked).
fsc/scn	For deferred block cleanout，this is commit SCN.

	For fast commit block cleanout，this is FSC.

The subsequent **ITL entries** follow the **same structure**, and their quantity is controlled by the **ITC (Interested Transaction List Count)**.

Additional Data

After the **ITL entries**, the **block header** ends, and the **data layer** begins. However, when the tablespace uses **ASSM (Automatic Segment Space Management)**, there are **additional 8 bytes** between the block header and the data layer. These bytes consist of two fields:

- **inc** (4 bytes): Used for space management in ASSM.
- **exflg** (4 bytes): Extended flags for block metadata.

In the DUMP output, these fields are **highlighted in yellow**. When calculating addresses for subsequent structures (e.g., row directories), you **must account for these 8 bytes**; otherwise, offsets will be misaligned.

When inspecting ASSM (Automatic Segment Space Management) and MSSM (Manual Segment Space Management) blocks using **BBED**, it was observed that BBED **automatically determines** whether to account for the **additional 8 bytes** (inc + exflg). This suggests that BBED does not rely on the segment header type but instead uses **block-level metadata** to decide whether to include these bytes.

Upon analyzing the data fields preceding **ktbbh** (the transaction layer structure), it was discovered that the **ktbbhflg** (transaction layer flag) in the block header controls this behavior. This flag (referred to as **ktbbhflg** in BBED) contains critical bits that dictate block-specific attributes.

Through experimentation, key flag bits were identified and mapped to their functionalities. The definitions of these flag bits are summarized in **Table 3-3** below:

Table 3-3 ktbbhflg Bits Definitions

Ktbbhflg definition：	
0x01	KTBFONFL, indicates block is placed on the **Free List**.
0x02	This flag indicates the **Segment's Object ID** associated with the block.
0x10	This flag indicates that the block is managed by **ASSM.**
0x20	The **flag (flg: E as shown in dump information)** indicates the presence of

additional data. This flag controls whether **8 additional bytes** are appended to the block's header. If the flag is **set (1)**, 8 bytes are added.

The **lower two bytes of the exflg field** act as an **offset** to adjust the starting address of the data layer, as shown in bellowing example.

(flg>>1)&0x03 The value calculated through this process corresponds to the **ver**.

In this sample, both fields in the **Additional Data** are 0, what happens when their values change? Let's modify the **exflg** value in the Additional Data and observe how the **kdbh (Kernel Data Block Header)** position shifts.

```
BBED> p ktbbh
... ... ... ...
   ub1 ktbbhflg                            @38        0x32 (NONE)
... ... ... ...
```

With the flag set to 0x32, the expression (0x32 & 0x20) is true, requiring an additional 8 bytes of data to added to the block header.

```
BBED> map
 File: /oracle11/product/11.2.0/dbs/study_01.dbf (8)
 Block: 135                                      Dba:0x02000087
------------------------------------------------------------
 KTB Data Block (Table/Cluster)
 struct kcbh, 20 bytes                          @0
 struct ktbbh, 72 bytes                         @20
 struct kdbh, 14 bytes                          @100
 struct kdbt[1], 4 bytes                        @114
 sb2 kdbr[1]                                    @118
 ub1 freespace[7992]                            @120
 ub1 rowdata[76]                                @8112
 ub4 tailchk                                    @8188
```

As shown above，kdbh offset is 20+72+8=100

```
BBED> set offset 96
        OFFSET          96

BBED> modify /x 0x02
Warning: contents of previous BIFILE will be lost. Proceed? (Y/N) y
```

```
 File: /oracle11/product/11.2.0/dbs/study_01.dbf (8)
 Block: 135              Offsets:   96 to  607           Dba:0x02000087
------------------------------------------------------------------------
 02000000 00010100 ffff1400 4c1f381f 381f0000 01004c1f 00000000 00000000
 ... ... ... ...
```

The offset corresponding to exflg is 96. If we modify the first two bytes to 2, then exflg = 0x00000002. Assuming the above analysis is correct, the offset of kdbh would be 20 + 72 + 8 + 2 = 102

```
BBED> map
 File: /oracle11/product/11.2.0/dbs/study_01.dbf (8)
 Block: 135                                      Dba:0x02000087
------------------------------------------------------------
 KTB Data Block (Table/Cluster)
 struct kcbh, 20 bytes                      @0
 struct ktbbh, 72 bytes                     @20
 struct kdbh, 14 bytes                      @102
 struct kdbt[0], 0 bytes                    @116
 sb2 kdbr[9223372036854775807]              @116
 ub1 freespace[18446744073709551596]        @114
 ub1 rowdata[94]                            @94
 ub4 tailchk                                @8188
```

As we analyzed, after the modifications, the offset of each structure observed using the map command show that kdbh has changed to 102

```
BBED> modify /x 0x020001
 File: /oracle11/product/11.2.0/dbs/study_01.dbf (8)
 Block: 135              Offsets:   96 to  607           Dba:0x02000087
------------------------------------------------------------------------
 02000100 00010100 ffff1400 4c1f381f 381f0000 01004c1f 00000000 00000000

BBED> map
 File: /oracle11/product/11.2.0/dbs/study_01.dbf (8)
 Block: 135                                      Dba:0x02000087
------------------------------------------------------------
 KTB Data Block (Table/Cluster)
 struct kcbh, 20 bytes                      @0
 struct ktbbh, 72 bytes                     @20
 struct kdbh, 14 bytes                      @102
 struct kdbt[0], 0 bytes                    @116
 sb2 kdbr[9223372036854775807]              @116
```

```
ub1 freespace[18446744073709551596]        @114
ub1 rowdata[94]                            @94
ub4 tailchk                                @8188
```

When exflg was modified to 0x00010002, it was observed that the offset of kdbh remained 102. This indicates that the increment in the offset is only determined by the value of the lower 2 bytes of exflg.
Similarly, it can be verified that modifying the value of the inc field does not alter the offsets of other structures.