ORACLE 10G BLOCK CHANGE TRACKING INSIDE OUT

Alexander Gorbachev, The Pythian Group

ABSTRACT

One of the new features in Oracle 10g is Fast Incremental Backups, which require that databases be run with Block Change Tracking enabled. While the usage of Fast Incremental Backups has been discussed quite thoroughly in recent years, the internals of the change-tracking feature implementation in Oracle 10g are still obscure. This presentation will show how change-tracking works inside an Oracle instance, and what CTWR process does. It will illustrate how other processes are involved and discuss the overhead caused by Block Change Tracking. This information is obtained by the means of experiments and advanced research techniques.

WHY THIS PAPER?

Oracle RMAN was able to take incremental backups already in 9i. However, prior to introduction of Oracle 10g *block change tracking* (BCT), RMAN had to scan the whole datafile to and filter out the blocks that were not changed since base incremental backup and overhead or incremental backup was as high as full backup. Oracle 10g new feature, block change tracking, minimizes number of blocks RMAN needs to read to a strict minimum. With block change tracking enabled RMAN accesses on disk only blocks that were changed since the latest base incremental backup.

This feature is widely known in the world of Oracle database administrators. However, hardly anything is available on internal implementation of block change tracking. This makes it difficult to evaluate the impact of enabling BCT in Oracle databases and quantify performance overhead.

This paper and presentation try to uncover internals of block change tracking and show which areas of Oracle database are involved, how processes work together, what are hidden limitations and impact of enabling block change tracking.

DISCLAIMER

I cannot provide any guarantee that presented material is absolutely correct. There is no publicly available documentation on change tracking internals (at least, I'm not aware of any) so this paper is based purely on experiments and research plus few hints from more knowledgeable peers that I'm very grateful for. Please take this material carefully and make sure you validate my assumptions before making critical decisions – either with your own tests or request Oracle support to provide more information if your business requires it.

METHODS AND TOOLS USED

`X$` tables provide wealth of information about Oracle internals and block change tracking in particular. The tables starting with `X$KRC` provide easy access to block change tracking file. Those tables are not simply exposing memory regions. When selecting from `X$KRC` tables, block change tracking file is accessed behind the scenes.

Extended SQL tracing using *event 10046* was used extensively on foreground sessions as well as background processes such as *DBWx*, *LGWR*, *CKPT*, and *CTWR*. It's worth to mention here that DBWR doesn't post "*db file parallel write*" wait event with *Async IO* enabled (Oracle 10.2.0.2 on Linux 32 bit). Here is where *strace* utility comes handy and let researcher see all system calls. In addition to strace, *lsof* utility can show open file descriptors for a given process.

To match content of `X$KRC` tables to the content of block change tracking file, hex dump or hex editor such as *bvi* is handy.

There are two events reserved for block change tracking tracing – 19774 and 19775. ORADEBUG utility is also very helpful.

BLOCK CHANGE TRACKING OVERVIEW

Without BCT enabled or in cases when change tracking information cannot be used, RMAN has to read every block in the datafile during incremental backup. Each block contains last *system change number (SCN)* of its last modification. The block is copied only if its SCN is higher or equal to the base backup SCN. To be precise, the backup SCN is a checkpoint SCN that

was made right before the backup. Since, usually, only handful of blocks is changed between incremental backups, RMAN does a lot of useless work reading the blocks not required for backup.

Block change tracking provides a way to identify the blocks required for backup without scanning the whole datafile. After that RMAN need only read blocks that are really required for this incremental backup.

However, improvement in incremental backup requires some sacrifice during normal database operations. According to Oracle this performance overhead is supposed to be minimal. Nevertheless, by default change tracking is disabled.

Oracle 10g introduces a new special background process – *Check Tracking Writer (CTWR)*. This process takes care of logging information about changed blocks in *block change tracking file*. Lets start with a closer look at this file.

BLOCK CHANGE TRACKING FILE

BCT file is one per database. In case of RAC database, change tracking file is shared amongst all instances. Thus, BCT file must be on shared storage. By default, BCT file is created in the location defined by parameter DB_CREATE_FILE_DEST as Oracle managed file (OMF). If parameter is not defined then location is platform specific and, usually, somewhere within `ORACLE_HOME`.

You can enable change tracking with the following statement:

```
SQL> ALTER DATABASE ENABLE BLOCK CHANGE TRACKING;
```

Alternatively, you can specify location of block change tracking file:

```
SQL> ALTER DATABASE ENABLE BLOCK CHANGE TRACKING USING FILE '/DB1/bct.ora';
```

To disable:

```
SQL> ALTER DATABASE DISABLE BLOCK CHANGE TRACKING;
```

View `V$BLOCK_CHANGE_TRACKING` can be queried to find out the status of change tracking in the database.

BCT FILE STRUCTURE

Block change tracking file consists of blocks. Each block has usual Oracle block format except that some of the bytes are not used, for example, block SCN. Block format code for BCT file is `0x22` in hex which is the second byte of every block. The first byte is the block type.

Below are the block types of BCT file with respective `X$` views providing interface to access information in those blocks:

- `0x29` – BCT file header (`X$KRCFH`)
- `0x2B` – unknown (`X$KRCCDE`)
- `0x2C` – unknown (`X$KRCCDS`)
- `0x2F` – datafile descriptors extent header
- `0x30` – datafile descriptor block (`X$KRCFDE`)
- `0x33` – bitmap extent header (`X$KRCFBH`)
- `0x34` – bitmap block (`X$KRCBIT`)
- `0x36` – probably, empty filler block

The block change tracking file block size is controlled by hidden parameter "*_bct_file_block_size*". On Linux platform the BCT block size is by default 512 bytes. The default might be different on other platforms. Block size is stored in BCT file header as we will see later. Figure 1 depicts the sample structure of a block change tracking file and sequence of the blocks. This example is from Linux x86 32 bit platform.

Note that the first block (or block zero if you wish) is often skipped just like with datafile but this might depend on the platform. There might be more than one chunk with datafile descriptors if they do not fit into one extent.

The bulk of BCT file is occupied by bitmap extents. Bitmaps are the bread and butter of change tracking – bitmap blocks store bit flags for every block (every chunk to be precise) in Oracle database. This is what CTWR updates when blocks are changed and what RMAN reads to determine which blocks it needs to backup.

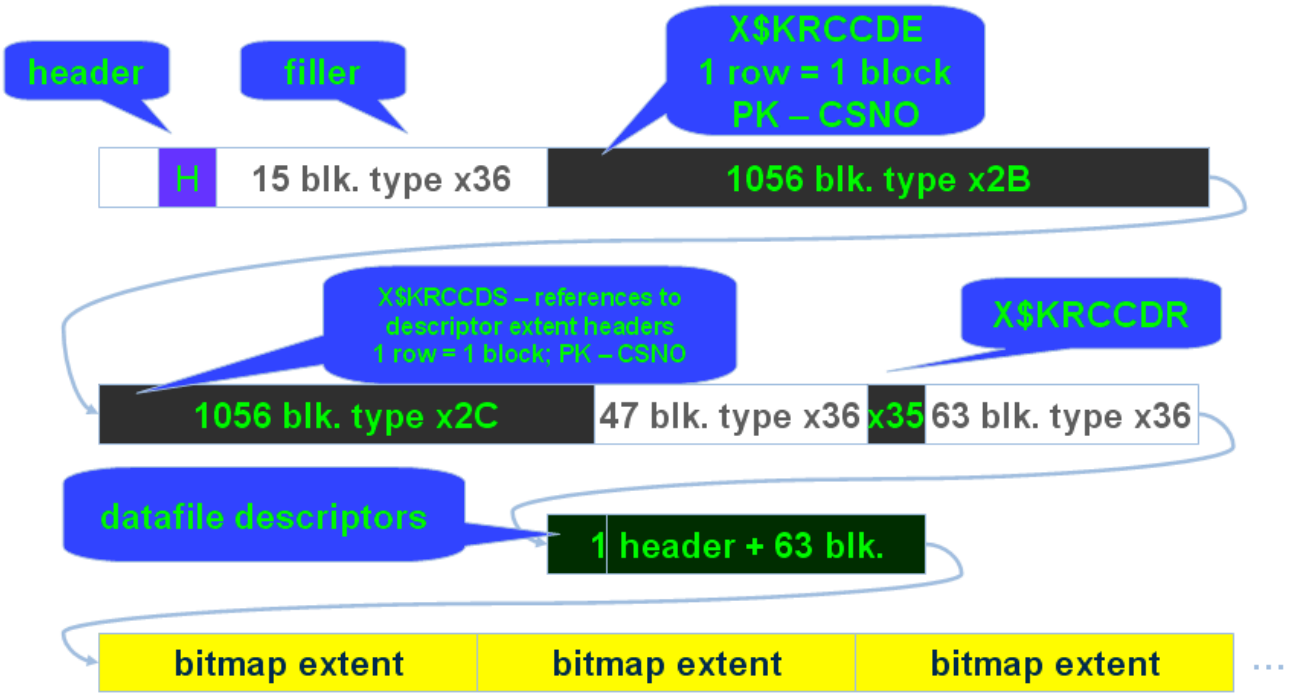

Figure 1 Block Change Tracking file structure

BCT File Header

Block type code is `0x29` Information form file header is presented via `X$KRCFH` table.

Most useful columns of `X$KRCFH` table:

- `FHSWV` – software version (10.2.0.2)
- `FHCVN` – BCT file compatibility version (10.0.0.0)
- `FHDBI` – DBID
- `FHDBN` – database name
- `FHFSZ` – BCT file size in blocks
- `FHBSZ` – block size in bytes (512)
- `EXTBLKS` – BCT file extent size in blocks (64 blocks = 32K)
- `RSZ_NEWEXTCNT` – BCT file increment size in extents (320 extents * 64 * 512 bytes = 10 MB)

There are many other columns that are not of so much practical interest or with less obvious purpose. For example, RESETLOGS SCN, checkpoint SCN, number of version switches, the last allocated extent in BCT file and etc.

Extent map

The first 2176 blocks (including first unused block) have predefined layout. The rest of the BCT file is allocated in extents. On Linux I observed extend size 32K or 64 blocks. It can be changed with hidden parameter "*_bct_file_extent_size*". The first block of an extent is extent header and the rest contain the data. View `X$KRCEXT` exposes *extent map*. Extent map only indicates whether extent is used or not. `X$KRCEXT` view is very simple and there are two interesting columns:

- BNO – extent header block number

- USED – flag; 0 – unused and 1 – used

Note that there is not information about extent type.

DATAFILE DESCRIPTORS

Oracle allocates the whole extent for *datafile descriptors*. Header block has type 0x2F and other 63 blocks with type 0x30. Information about datafile descriptors is externalized via `X$KRCFDE` fixed table.

One 512 byte block contains 4 datafile descriptors. Consequently, one extent with 63 useful blocks fits up to 252 descriptors. Datafile descriptor extents are pre-allocated and formatted based on *db_files* init.ora parameter.

Below are the most important columns:

- `CTFBNO` – change tracking file block number where descriptor is located
- `FNO` – absolute file number
- `CHUNK` – chunk size in database blocks (four 8K blocks for 32K chunk)
- `CRESCN` – datafile creation SCN
- `CRETIME` – datafile creation time
- `CURR_LOWSCN` – start SCN for the current version
- `CURR_HIGHSCN` – end SCN for the current version; for current version it's set to max possible SCN – 2^{48}-1
- `CURR_FIRST` – header block number of the first bitmap extent for current version
- `CURR_LAST` – header block number of the first bitmap extent for current version
- `CURR_EXTCNT` – number of extents in the current version
- `CURR_VERCNT` – current version number
- `CURR_VERTIME` – current version time; time when the version started
- `HIST_FIRST` – header block number of the first bitmap extent for previous version
- `HIST_LAST` – header block number of the first bitmap extent for previous version
- `HIST_EXTCNT` – number of extents in the previous version
- `HIST_VERCNT` – current previous number
- `HIST_VERTIME` – previous version time
- `OLDEST_LOW`- start SCN for the oldest bitmap version available in BCT file

Oracle should have called the feature *chunk* change tracking because it keeps track of changes for the whole chunk and not individual blocks. Chunk size seems is be default 32K but can be changed with underscore parameter "*_bct_chunk_size*". Consecutive blocks are joined in one chunk and it's considered dirty if any of the blocks is changed. There are four 8K blocks in one chunk, eight 4K blocks and one 32K block, for example. This approach, probably, simplifies implementation and change tracking file sizing doesn't depend on datafile block size. 32K seems to be chosen because this is the smallest chunk to fit maximum Oracle block size on all platforms. It could be that on platforms with max block size less than 32, Oracle tracks changes in smaller chunks, i.e. with higher granularity. Would be interesting to try setting chunk size to 16K and create tablespace with 32K block size – interesting what error Oracle would throw.

BITMAP EXTENTS

Bitmap extent associated with a single datafile. One datafile has at least one bitmap extent for each bitmap version stored.

The first block is a bitmap extent header. Other 63 blocks contain bitmap. Each bit corresponds to a datafile chunk. When chunk is not changes for particular version – the bit is 0. If any of the blocks in the chunk is changed, the bit is set to 1. So far I mentioned bitmap version several times so it's good place to describe it.

BITMAP VERSIONS

Oracle tracks which blocks (actually, chunks) are changed between two consecutive incremental backups of a datafile and not only since the last backup. There are separate bitmaps covering each period between incremental backups. This timeframe and associated bitmaps are called versions.

Every backup has *checkpoint SCN* associated with it. This is the SCN of the last checkpoint before the backup. Every change tracking bitmap version has *start SCN* (or low SCN), which is equal to a checkpoint SCN of the previous backup. Every version except current has *end SCN* (or high SCN) which is the checkpoint SCN of the next incremental backup. Current version has end SCN set to maximum possible SCN value which is 2^{48}-1. Versions are associated with datafile and each datafile has its own set of version even though they might be close to each other in terms of low and high SCN's.

Every version has dedicated bitmaps and bitmap extents. We can now refine our understanding of bitmap – bits associated with blocks that were changed between version's low and high SCN's are set to 1. Otherwise, they are zeroes.

Number of versions to keep is 8 by default and can be changed by hidden parameter "*_bct_bitmaps_per_file*".

BITMAP EXTENT HEADER

Information from bitmap extent headers is externalized via **X$KRCFBH** fixed table. Some of its columns are described below:

- **CTFBNO** – header block number in change tracking file
- **FNO** – absolute file number of a datafile
- VERCNT – version number for bitmap in this extent
- VERTIME – timestamp when this version was started
- **HIST_FIRST**, **HIST_LAST**, **HIST_EXTCNT**, **HIST_VERCNT**, **HIST_VERTIME** – same as in **X$KRCFDE**
- **LOWSCN** – start SCN for the version
- **HIGH** – end SCN for the version

BITMAP BLOCK

Every 512 bytes block provides 488 bytes for bitmaps which represent 3904 bits and can, therefore, cover 3904 chunks. Recall that every chunk is 32K. One block can track changes for 122 MB of a datafile (3904 * 32K). The whole extent contains 63 bitmap blocks and covers up to 7686 MB of one datafile. Note that this is only for one version. If file is resized and doesn't fit into existing number of extents, additional extents are allocated.

Bitmaps are externalized via **X$KRCBIT** fixed table. This **X$** table contains row for every bit that is set to 1 and nothing for bits set to 0. This makes sense as only *dirty chunks* are of interests for backups. In other words, each row in **X$KRCBIT** represents a dirty chunk in one of the version.

X$KRCBIT has the following columns:

- **CTFBNO** – block number of extent header (**X$KRCFBH**)
- **FNO** – absolute file number of a datafile (**X$KRCFBH**)
- **VERCNT** – version number (**X$KRCFBH**)
- **VERTIME** – timestamp when this version was started (**X$KRCFBH**)
- BNO – first block number of a chunk in the datafile
- BCT – chunks size in datafile blocks (**X$KRCFDE.CHUNK**)

All the columns are carried over from the extent header or the datafile descriptor except the first block number of a chunk that is inferred from bit offset.

BCT FILE SIZING

Minimal file size is about 11MB - 2176 initial blocks (1088K) + 320 extents (32K * 320 = 10 MB). 10 MB seems to be predefined increment for BCT file resizing. Out of those 320 extents, one is reserved for **X$KRCCDR** structure and at least one is reserved for datafile descriptors. Recall that one extent covers 252 files and they are pre-allocated based in init.ora parameter **db_files**. Thus, assuming that 8 versions for each datafile are kept, 318 extents can cover up to:

```
318 extents * 7686 MB / 8 version = 300 GB.
```

Note that in this case maximum number of existing datafiles can be up to:

```
318 extents / 8 version = 39 datafiles.
```

In real life, some datafiles are just few hundred megabytes and others sized in tens of megabytes so there is no simple formula to fit all cases. Here is the complete statement that can be used to predict maximum BCT file size based in the current database when all datafiles backed up with incremental backup at least 7 times:

```
SELECT((
  (SELECT SUM(ceil(bytes /(7686 * 1024 * 1024))) * 8 bitmap_ext
   FROM v$datafile) +
    (SELECT ceil(VALUE / 252) file_descr_ext
     FROM v$parameter
     WHERE name = 'db_files') + 1)
  * 32 + 1088) / 1024 bct_file_size_mb
FROM dual;
```

The statement is valid only for single instance. In RAC, every instance will use its own bitmap extents so number of bitmat extents should be multiplied by the number of RAC instance.

Oracle documentation has the following note about BCT file sizing:

> "For each datafile, a minimum of 320K of space is allocated in the change tracking file, regardless of the size of the file. Thus, if you have a large number of relatively small datafiles, the change tracking file is larger than for databases with a smaller number of larger datafiles containing the same data."

Actually, space is not allocated right away – only one bitmap extent is allocated initially. As soon as we start performing incremental backups, new extents for new versions of bitmaps get allocated. I believe that there are only 8 extents required for one small datafile – one for each version including current. However, I've been informed that there is 8 versions kept plus current version and one more extent for some overhead. I haven't been able to identify this overhead and tests showed that 8 versions include the current one. Thus, I believe that 256K is only required minimum for a datafile providing it's a part of incremental backup.

BCT FILE EVOLUTION

We will review how different actions are reflected in block change tracking file:

- Adding and dropping datafile
- Changing blocks
- Incremental backups
- Restore

For now I just illustrate what is changed in the BCT file. The processes and how they interact will be covered later.

ADDING DATAFILE

```
CREATE TABLESPACE TBS1 DATAFILE SIZE 128M;
```

First of all new extent is allocated. Extent allocation is not a straightforward procedure. As far as my observations go, there is some extents pre-allocated as reserve. It seems that column **RES_EXTCNT** of table **X$KRCCDS** returns number of extents in the reserve list and column **RES_FIRST** in the same table shows a header block number of the first reserved extent. So Oracle either take extent from reserve or allocates a new one and mars it as used which we can see in extent map table **X$KRCEXT**.

Next step is to format a new datafile descriptor (**X$KRCFDE**). Current version start SCN is set to zero and end SCN is set to 2^{48}-1. Version number is set to 1 unless there already was existing datafile with such absolute file number. It seems that in this case version is simply set to the next integer number.

Bitmap extent header block gets formatted (**X$KRCFBH**) as well as bitmap blocks themselves. Since Oracle formats (read changes) several data blocks in the new datafile, first chunks are marked as dirty in the bitmap blocks.

DROPPING TABLESPACE

Bitmaps blocks are cleared (**X$KRCBIT**), bitmap extent headers are cleared (**X$KRCFBH**) as well as datafile descriptor (**X$KRCFDE**). Extents are marked as unused in extent map (**X$KRCEXT**) but not immediately – after some time.

CHANGING BLOCKS

As I mentioned already, changes are tracked for 32K chunks of datafile so any block change in the chunk will render it dirty and bit must be set to 1. If the chunk was already dirty than nothing will change in the BCT file. Basically, if previous block SCN is more than corresponding to the datafile `X$KRCFDE.CURR_LOWSCN`.

DATAFILE INCREMENTAL BACKUP

When RMAN starts incremental backup of a datafile, it needs to create new version in change tracking file.

Current bitmap is marked as historical – `XFLAG` column of `X$KRCFBH` (bitmap headers) is set to 3 and `HIGH` column is set to the checkpoint SCN. Purge bitmaps that are older than 8 versions.

New extents are allocated and formatted. Bitmap extent header block is formatted but no version specific information is written. I.e. querying `X$KRCFBH` we won't see columns `VERCNT`, `VERTIME`, `LOW`, `HIGH` and etc. filled. This is because of version optimization – if blocks in datafile haven't been changed between two incremental backup then new version is not created. Thus, version attributes are updated only when chunks are first marked as dirty.

Datafile descriptor is updated (`X$KRCFDE`) – attributes of current version and previous version are filled (`CURR_%` and `HIST_%` columns).

When some a chunk needs to be marked dirty for the first time after incremental backup, version information in bitmap extent headers (`X$KRCFBH`) is initialized – `XFLAGS` is set to 2, CURR_VERCNT and CURR_VERTIME filled from datafile descriptor. If file had no changes since last incremental backup – nothing is written to the bitmap.

There is no difference in handling different backup levels. Version is created in the same way whether its level 0, level 1 or level 4 backup. By the way, it seems that Oracle 10g documentation officially mentions only support for levels 0 and 1. However, I checked incremental backup levels up to 4 and they do work (10.2.0.2).

DATAFILE RESTORE

If single datafile needs to be restored, BCT file keeps information on versions and it can be successfully backed up later using change tracking optimization to read just dirty chunks.

I did a small test:

- Removed datafile in OS
- Set datafile to offline mode
- Restored from backup using "`RESTORE DATAFILE <#>;`"
- Recover that datafile using "`RECOVER DATAFILE <#>;`"
- Online datafile

All version information was preserved.

8 VERSIONS IMPACT

Let's walk through an example of bi-weekly incremental backup cycle.

2 TB data warehouse database containing 5 years worth of data is backed up every other Sunday with incremental level 0 backup. Full backup is running 20 hours. For the next 13 days incremental level 1 *cumulative* backup is taken. Cumulative level 1 backup means that RMAN will need to copy blocks changed since last level 0 backup.

Backup is running every morning after nightly ETL batch completes. The batch changes about 1% (including new data loaded, updated indexes and changes to the staging tables). Half of changed blocks are in staging area. Another half is new data loaded and indexes updated. This means that first incremental level 1 cumulative backup is 0.5% of the database or 10 GB. The next level 1 cumulative backup adds 0.25% of the database size to previous size so sizes are 10 GB, 15 GB, 20 GB and so on ending with 70 GB on the last level 1 backup before level 0 backup.

Incremental backups take less an hour so they finish before users start their day and hit database with their requests.

Let's assume that we enabled change tracking just before level 0 incremental backup and version number 1 is the current version. Incremental level 0 backup starts and as soon as each datafile is backed up, the current version becomes 2.

Monday – incremental backup kicks off and version 3 is the current version. No backup is purged. RMAN is happily using change tracking file to determine which blocks are needed for backup – RMAN scans the bitmaps since last level 0 backup – version 2 bitmaps.

Tuesday - incremental backup kicks off and version 4 is the current version. No backup is purged. RMAN again scans the bitmaps since last level 0 backup. This time it needs bitmaps for versions 2 and 3. Some blocks might be marked dirty in both versions. In fact, those are blocks in the staging area representing 0.25% of the database size as we stated above.

Backups for the next days until Sunday are working under the same scenario using bitmaps since version 2. Sunday's incremental level 1 cumulative backup does the same but it now purges oldest bitmap version. The current version is switched to number 9 on Sunday's backup and version 1 needs to be purged – Oracle keeps only 8 versions *including current version*. This is not a problem and RMAN still can use versions from 2 till 8 to determine which blocks have been changed and must be backed up.

Second Monday - incremental backup kicks off and version 10 becomes the current version. Bitmaps of version 2 are purged. Now RMAN cannot locate all the required versions to find all the dirty blocks changed since incremental level 0 backup – it misses bitmap version 2 and cannot identify blocks changed between the last level 0 and the first level 1 incremental backup. As a result, RMAN has to fall back to the old incremental backup method and scan the whole database.

The consequences are 10 hours incremental backup, IO subsystem performance degradation, users are unhappy because their requests take few times longer than usual.

PREDICT NUMBER OF BLOCKS READ AND BACKUP SIZE

Based on the bitmap content (`X$KRCBIT`) we can predict how many 32K chunks RMAN would read if incremental backup starts now. However, precise backup size prediction is only possible if database block size is 32K. Otherwise, a 32K chunk consists of multiple database blocks (2, 4, 8 or 16 depending on the block size). It's quite possible that only one 8K block out of 4 in a chunk is changed. In this case RMAN will read the whole chunk but it needs to copy only one changed block. RMAN still can filter out blocks unchanged since last backup SCN based on block SCN just like it does for every block in case change tracking is not enabled.

The query below can be used to calculate how many kilobytes will be read by RMAN to backup datafile 7 based on last level 1 incremental backup (i.e. incremental level 1 or incremental cumulative level 2):

```
SELECT count(distinct bno) * 32
FROM x$krcbit b
WHERE b.fno = 7 AND b.vercnt >=
  (SELECT MIN(ver) FROM
    (SELECT curr_vercnt ver, curr_highscn high, curr_lowscn low
     FROM x$krcfde WHERE fno = 7
     UNION ALL
     SELECT vercnt ver, high, low
     FROM x$krcfbh WHERE fno = 7)
  WHERE (SELECT MAX(bd.checkpoint_change#)
     FROM v$backup_datafile bd
     WHERE bd.file# = 7
     AND bd.incremental_level <= 1) between low and high);
```

Running incremental backups for a while it's possible to collect historical ration between number of blocks read and number and size of the backup. This would as well account for compression.

Note that the query above is just an example and it has the following limitations:

- Chunk size is hard coded to 32K (could it vary on different platforms?)
- First block overhead is not accounted for
- No special case when required bitmap version is not available (purged) and the whole datafile must be read
- No case with backup optimization for level 0 (`v$datafile_backup.used_optimization`)
- No case when no data blocks in datafile is changed (no bitmap version but the first block must be backed up anyway)
- Only single datafile
- No accounting for unavailable base incremental backup

CHANGE TRACKING INSIDE ORACLE INSTANCE

There is a new special process in Oracle 10g – *Change Tracking WRiter (CTWR)*. As name suggests the job of this process is to write to the block change tracking file. In normal Oracle operations there is no other process that is writing to it. RMAN shadow process reads from as well as writes to the change tracking file during incremental backups.

Any process in Oracle can read from change tracking file. When user session queries `X$KRC%` views, it access BCT file behind the scenes. Shadow process itself accesses CTWR file without communications to CTWR process.

WHO DOES WHAT WHEN BLOCKS ARE CHANGED

Let's see what happens when processes change blocks.

Every change to the datafile blocks generates redo entry. The process first produces change vector and than this change vector is applied to the block. The change is not applied directly to the block on disk except for direct path writes. The block is first loaded into buffer cache in SGA and then change vector is applied changing the block content. The block in the buffer cache is now considered dirty and later one of the *Database WRiter processes (DBWx)* will flash it to disk.

Neither of these processes is writing to the change tracking file to mark chunks in the bitmaps as dirty. This is the job of *Checkpointer process (CTWR)* but how does it know which blocks are changed? The process that updates the block in buffer cache is responsible for passing information to the CTWR about which blocks were changed. At the time the process puts redo entry in the log buffer, it also updates special buffer in the SGA that is used by change tracing writer process later to update the bitmaps. The mechanism of asynchronous updates to the BCT file is somewhat similar to log buffer but not entirely the same. It also doesn't require flushing pending changes to the BCT file on commits so foreground processes won't wait on something similar to "log file sync".

When does CTWR update the BCT file? This happens usually during checkpoint. CKPT process signals to CTWR process to flush pending changes to the BCT file. CKPT is waiting for confirmation that CTWR received the instruction. However, CTWR doesn't update bitmaps to the BCT file immediately after the signal from CKPT. Instead, it posts back to the CKPT. As soon as CKPT gets the signal from CTWR, it updates datafile headers and controlfile so checkpoint is completed. CTWR in the meantime proceeds with updating bitmaps in the block change tracking file.

What happens with direct path writes when blocks are updated by shadow processes and written directly on disk? The process is similar. Direct path writes still generate some redo – block invalidations. When these redo entries are written to the log buffer, information is written to the change tracking buffer in SGA and CTWR process will update the bitmaps in BCT file later during the next "*CTWR heartbeat*" or checkpoint.

WHO DOES WHAT DURING RMAN INCREMENTAL BACKUP

When incremental datafile backup is started, RMAN shadow process in Oracle instance reads from the BCT file and then updates the BCT file header as well the block 2176 exposed via `X$KRCCDR` fixed table. The next step for RMAN is to signal to the CTWR process that new version needs to be created. At this point RMAN waits for CTWR to finish its business.

CTWR creates the new version including allocation of new extents and purging old bitmaps as needed. After that, CTWR posts back to the RMAN shadow process.

RMAN shadow process can now read all required past bitmaps and identify which chunks are required for backup. Finally, RMAN gets to its main part – reading dirty chunks and writing changed blocks to the backup piece.

CTWR WAIT EVENTS

CTWR process performs updates to the block 18 of the BCT file about every 20-60 seconds – I call it *"CTWR heartbeat"*. During this heartbeat CTWR reads and writes block 18 posting sequentially two short wait events – "*change tracking file synchronous read*" and "*change tracking file synchronous write*".

When updating the BCT file, CTWR process also performs some IO against controlfile ("*control file sequential read*" and "*control file parallel write*"). Reads and writes to the BCT file itself are instrumented with "*change tracking file synchronous read*" and "*change tracking file synchronous write*". However, updates of the bitmap blocks are *NOT* instrumented. This was puzzling me for long time as I couldn't detect when bitmap blocks are updated until I traced CTWR process with Linux strace. With strace I found system calls to pwrite64 function. CTWR was updating all blocks of a single bitmap extent with a single call. There is wait event "*change tracking file parallel write*" and perhaps wait event supposed to be used for bitmap writes. Could be it's an implementation bug (Linux 32 bit platform).

Bitmap updates after direct path writes done during the CTWR heartbeat are not instrumented either and cause me quite a bit of confusion because I couldn't catch the moment how bitmaps get updated in this case. Again, strace dotted all the i's and crossed the t's.

SHARED POOL AND LARGE POOL AREAS

There are several shared pools areas that I believe are related to change tracking:

- *change tracking state cha* – this area is most probably used when change tracking is enabled and disabled as well as during instance startup when CTWR process is started. It seems that there is latch "*change tracking state change latch*" protecting this shared pool area.
- *pq_krcpx* – this and next 3 areas are unknown to me and perhaps used for parallel loads/direct path writes and/or SCN tracking.
- *scntab_krcpx*
- *krcpx*
- *krccr_hp_krccx*

Two new areas in large pool:

- *krcc extent chunk*
- *CTWR dba buffer*

CTWR BUFFER WAIT

Wait event "*block change tracking buffer space*" seems to be posted when processes need to write to CTWR buffer and it's full. I wasn't able to reproduce and investigate this wait event on my test systems. Perhaps, CTWR was flushing it quicker than processes could write to it. If my assumption is correct – this wait event should be a direct indicator of excessive overhead caused by change tracking.

The size of change tracking buffer can probably be controlled by hidden parameter "*_bct_public_dba_buffer_size*".

LATCHES

Latch "*change tracking state change latch*" is probably there to protect shared pool area "*change tracking sta*". This latch is taken by CTWR process during enabling and disabling change tracking in the database as well as during instance startup when CTWR process starts.

Two other related latches, "*change tracking optimization SCN*" and "*change tracking consistent SCN*", have probably something to do with the way CTWR tracks SCNs and, perhaps, optimization so that CTWR doesn't need to update bits for the blocks that are changed over and over again while the same version of bitmap is current.

ENQUEUES

Several new enqueues related to block change tracking mechanism but so far I can't clearly describe their role:

- *enq: CT - global space management*
- *enq: CT - local space management*
- *enq: CT - change stream ownership*
- *enq: CT – state*
- *enq: CT - state change gate 1*
- *enq: CT - state change gate 2*
- *enq: CT - CTWR process start/stop*
- *enq: CT – reading*

HIDDEN PARAMETERS

- *_bct_public_dba_buffer_size* – total size of all public change tracking dba buffers, in
- *_bct_initial_private_dba_buffer_size* – initial number of entries in the private change tracking dba buffers

- *_bct_bitmaps_per_file* – number of bitmaps to store for each datafile
- *_bct_file_block_size* – block size of change tracking file, in bytes
- *_bct_file_extent_size* – extent size of change tracking file, in bytes
- *_bct_chunk_size* – change tracking datafile chunk size, in bytes
- *_bct_crash_reserve_size* – change tracking reserved crash recovery SGA space, in bytes
- *_bct_buffer_allocation_size* – size of one change tracking buffer allocation, in bytes
- *_bct_buffer_allocation_max* – maximum size of all change tracking buffer allocations, in bytes
- *_bct_buffer_allocation_min_extents* – mininum number of extents to allocate per buffer allocation
- *_bct_fixtab_file* – change tracking file for fixed tables

OVERHEAD AND HIDDEN LIMITATIONS

Let's summarize non-obvious limitations and identify possible bottlenecks and overhead.

NO DISTINCTION BETWEEN BACKUP LEVELS

As I mentioned already, CTWR doesn't make any distinction between backup levels. Regardless of incremental backup level, bitmap version is simply tracking changed blocks between any two consequent backup.

This is a very smart solution. What are the requirements from RMAN to identify the blocks eligible for backups? It simply needs to know all blocks changed between SCN of the base backup and SCN of the current backup. It doesn't matter which backups were taken in between these SCNs – it's only important that all versions between two SCN's are available. Some blocks can be changed in every version and some – only in one version. Recall the query I used to predict number of blocks that RMAN will read. Similar approach is taken by RMAN but it accesses the BCT file directly instead of querying `X$KRCBIT` fixed table.

However, we have already gone though bi-weekly cumulative backup strategy and seen that almost half of incremental backups couldn't actually use change tracking feature.

Similar scenario, though less common, is possible with just incremental backup. I remind you what the differences between normal incremental backup and incremental cumulative backup. Cumulative backup contains blocks changed since last incremental backup of *previous or lower level*. I.e. incremental level 2 cumulative backup needs all blocks changed since last level 1 or level 0 backup even though there might have been one or more level 2 backup taken after that. Non-cumulative backup includes only blocks changed since last incremental back of *at least the same level*. For example, incremental level 2 backup (non-cumulative) requires only blocks updated since last level 2 or lower backup.

Let's consider now fairly uncommon monthly backup cycle. Though it's not often that this kind of backup is used there are use cases when it can be handy. On of such use cases is updating datafile image copies using incremental backup – new feature available in Oracle 10g:

- Monthly – incremental level 0
- Weekly – incremental level 1
- Nightly – incremental level 2
- Hourly – incremental level 3

23 hourly backups (actually 9th backup) will cause CTWR to purge versions of bitmaps with changes since last level 2, 1 and 0 backups. This will cause all level 1 and 2 backups to scan the whole database to identify the blocks eligible for backup.

There is a hidden parameter *_bct_bitmaps_per_file* set to 8 by default but no officially supported way to change number of versions.

POSSIBLE BOTTLENECKS

I didn't observe any noticeable and measurable impact of enabling block change tracking. However, it doesn't mean there are none.

Since all IO performed by CTWR asynchronously in regard to the foreground sessions and number of IO's is minimal, there are little chances that this would be a bottleneck for the users sessions. The only problem would be when change tracking

buffer becomes full and CTWR cannot keep up flushing it to the BCT file. However, this would probably be a side-effect of other issues on the system such as extreme CPU capacity shortage, which is itself often not a root cause, or very poor IO performance. In both cases chances are that more sensitive processes of Oracle instance are impacted first.

IN CASE THE MURPHY ISN'T SLEEPING

In the unlikely circumstances when enabling change tracking causes performance issues – fallback is simple. Disabling change tracking is an online action. The only consequence is that incremental backups will take longer.

In case the BCT file gets corrupted due to whatever reason (bug, media corruption, human factor), it can be easily recreated online by disabling and re-enabling block change tracking. Again, don't forget that all versioning information is lost in this case and next incremental backup will have to scan all blocks.

REFERENCES

1. Advanced Research Techniques by Tanel Põder
2. Backup and Recovery Basics, Section 4.4
3. Metalink Notes 262853.1 and 306112.1