Oracle DB-Tuning Essentials

Agenda

1. The DB server and the tuning environment
2. Objective, Tuning versus Troubleshooting, Cost Based Optimizer
3. Object statistics
4. Access paths I
5. Access paths II
6. Monitoring Performance I
7. Monitoring Performance II
8. Application setup

Access Path

- Table Access
- Index Access
- Join Methods
- Sorts and hash
- Hints
- Parallel Query

Access paths
Efficient methods to access data

Oracle provides several methods to access data. E.g.

- Full Table Scan
- Table access by rowid
- Index/Hash Cluster
- B*-Tree Index
- Bitmap Index
- Materialized View

All have their pro's and con's. For a performing system it's important to use the most efficient access path to the data.

Table Access
Full Table Scan

Heap Tables (the default)

- Can store a row anywhere within the segment
- Pushes the 'High Water Mark' when formatting new blocks
- Segment header has the allocation map (and the HWM)

Full Table Scan has to read all blocks up to the HWM

- Even if there is no row (had rows that have been deleted)
- Uses efficient multiblock reads (large I/O)
- Good when we need rows from most of the blocks

Inserts
Deletes
Full Table Scan

Table Access
Demo High Water Mark

1. Select (Full Table Scan) on large table
2. Delete all rows from the table and gather stats
3. Same select again: all the blocks are still read

```
-------------------------------------------------------------------------
|Id | Operation          | Name | Starts | E-Rows | A-Rows | Buffers |
|-----------------------------------------------------------------------
| 0 | SELECT STATEMENT   |      |      1 |        |      1 |     233 |
| 1 |  SORT AGGREGATE    |      |      1 |      1 |      1 |     233 |
| 2 |   TABLE ACCESS FULL| DEMO |      1 |      1 |      0 |     233 |
|-----------------------------------------------------------------------
```

4. After a truncate table:

```
-------------------------------------------------------------------------
|Id | Operation          | Name | Starts | E-Rows | A-Rows | Buffers |
|-----------------------------------------------------------------------
| 0 | SELECT STATEMENT   |      |      1 |        |      1 |       3 |
| 1 |  SORT AGGREGATE    |      |      1 |      1 |      1 |       3 |
| 2 |   TABLE ACCESS FULL| DEMO |      1 |      1 |      0 |       3 |
|-----------------------------------------------------------------------
```

Table Access
How to lower the High Water Mark?

ALTER TABLE SHRINK SPACE COMPACT

ALTER TABLE SHRINK SPACE or ALTER TABLE MOVE

TRUNCATE TABLE

Table Access
Alter table SHRINK SPACE

Moves rows near the HWM to an area with free space

- Need to ENABLE ROW MOVEMENT before (lock)
- SHRINK SPACE COMPACT needs no lock
- SHRINK SPACE needs lock to lower HWM but quick when after the COMPACT

```
SQL> ALTER TABLE … ENABLE ROW MOVEMENT;
SQL> ALTER TABLE … SHRINK SPACE COMPACT;
SQL> ALTER TABLE … SHRINK SPACE;
SQL> ALTER TABLE … DISABLE ROW MOVEMENT;
```

Takes longer than MOVE or dbms_redefinition

- Useful if you have are low on free space (does not need twice the space)

Can change the clustering factor of some indexes

- Don't do it too frequently
- Be careful if you rely on good clustering for indexes that follow the insertion date (range scans on sequence, on dates,…)

Table Access
Alter table SHRINK SPACE

How to plan a shrink?

- ENABLE ROW MOVEMENT before, when low DML activity
- ALTER TABLE ... SHRINK SPACE COMPACT during the day (not at peak activity because of additional redo)
- ALTER TABLE ... SHRINK SPACE later when low activity

We can keep ENABLE ROW MOVEMENT enabled

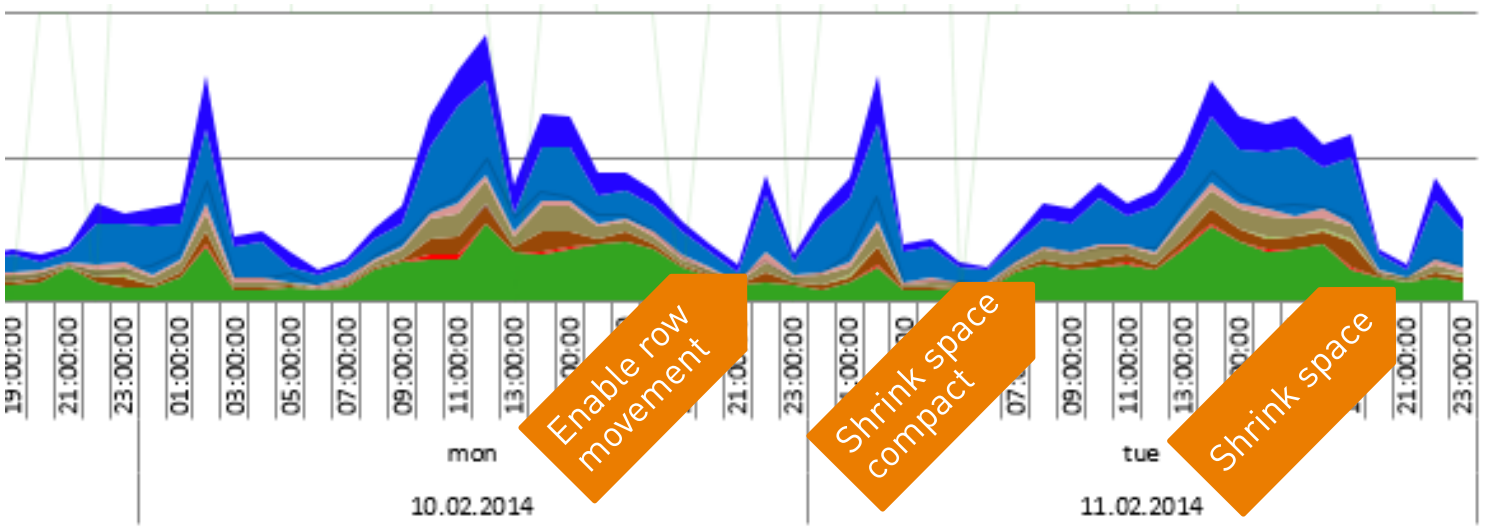


Table Access
Demo: SHRINK TABLE

- Connect as demo/demo
- Create a table and gather stats:

```
SQL> create table DEMOHWM as select * from CUSTOMERS;
SQL> exec dbms_stats.gather_table_stats(user,'DEMOHWM');
```

- Do a full table scan with autotrace:

```
SQL> set autotrace on
SQL> select count(*) from DEMOHWM;
```

  - how many blocks are read?

- Delete half of the rows and do the same select

```
SQL> delete from DEMOHWM where cust_year_of_birth<1960;
```

  - how many blocks are read?

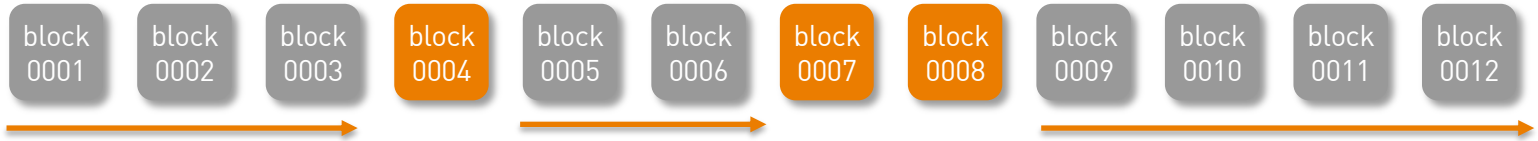
- Shrink the table, and do the same select
  - how many blocks are read?

Table Access
Multiblock read

When reading several contiguous blocks

- Read all in one I/O call (db file squattered read)
- Up to db_file_multiblock_read_count (default: OS max I/O size)
- Only blocks that are not already in buffer cache

Blocks 0004, 0007 and 0008 already in cache:



All blocks read in 3 I/O calls

Recommendation

- Don't set the db_file_multiblock_read_count
- If it needs to be set (I/O size is smaller), be sure to have workload system statistics for the optimizer to use them

Table Access
Demo: Multiblock Read

Full table scan wait events

1. With db_file_multiblock_read_count=16

```
EVENT                          P3TEXT               P3 WAIT_TIME_MICRO
------------------------------ ---------- ---------- ---------------
direct path read               block cnt          16              17
direct path read               block cnt          16              17
```

2. With db_file_multiblock_read_count=64

```
direct path read               block cnt          62             143
direct path read               block cnt          64              73
```

3. Reading to buffer cache

```
db file scattered read         blocks             64             353
db file scattered read         blocks             62             321
```

4. Reading only single blocks

```
db file sequential read        blocks              1               4
db file sequential read        blocks              1               5
```

Table Access
Table access by rowid

ROWID identifies a table row

- The segment has an id (data_object_id)
- The extent is in a file (relative_fno)
- The block has an offset in the file (block_id)
- The row has a number in the block (the row directory entry)

Access by ROWID

- Is the fastest access to retrieve one row
- Is the slowest access to retrieve all the rows

ROWID can come from

- User provided rowid (not consistent across transactions)
- Index entries that address the table row with a ROWID
- Hash cluster that calculates ROWID from hash function

Table Access
Demo: table access by rowid

- Connect as demo/demo
- Set autotrace
- Get rowid for customer 100

```
SQL> set autotrace on
SQL> select rowid,cust_id from CUSTOMERS where cust_id=100;
```

  - how many blocks are read?

- Get row from this rowid

```
SQL> select * from CUSTOMERS where rowid='AAAXYMAAGAAABZaAA7';
```

  - how many blocks are read?

- View ROWID information

```
SQL> select dbms_rowid.rowid_object(rowid) ,dbms_rowid.rowid_relative_fno(rowid)
           ,dbms_rowid.rowid_block_number(rowid) ,dbms_rowid.rowid_row_number(rowid)
```

Table Access
Single Table Hash Cluster

A heap table allocates space as needed

- There is no defined place for a row
- Need to full scan if we don't have the ROWID

A hash cluster allocates blocks for specific values

- We define the number of hash keys
- We define how much space to allocate for each key (default 1 block)
- The hash key is calculated from a column value (not necessary the PK)

Quick access to the hash entry (one block only)

- Quick access to the row if few collisions only and no overflow
- Longer access if the row has been allocated in another block
- Good only for static tables (lookup tables) where the volume is known
- In 12c can be partitioned by range: makes it less static.

Table Access
Exercise: fetch size

- Connect as demo/demo
- Create the following table and check number of blocks:

```
SQL> create table DEMO as select CUST_ID from CUSTOMERS;
SQL> select blocks from user_tables where table_name='DEMO';
```

  - how many blocks under high water mark?

- Set autotrace and select all rows:

```
SQL> set autotrace on
SQL> select * from DEMO;
```

  - how many blocks are read (consistent gets)?
  - Why? How to read the minimum of blocks?

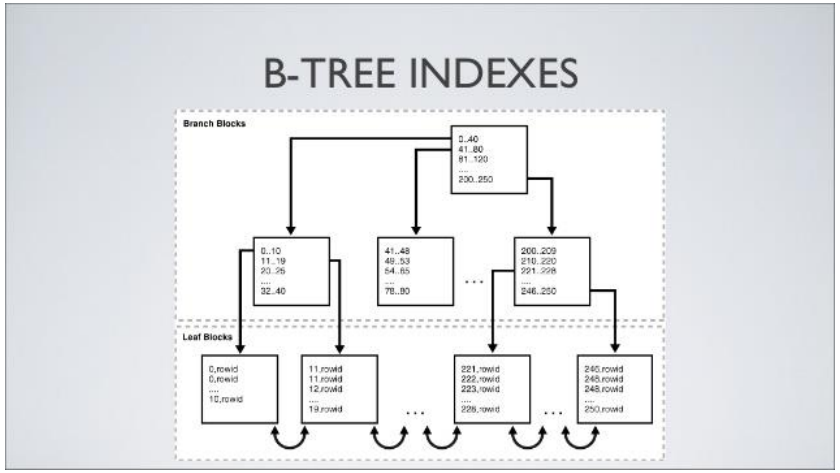
Hints:

1. Multiblock read is good to retrieve lot of rows in one call
2. Check also 'SQL*Net roundtrips to/from client'

Access Path

- Table Access
- Index Access
- Join Methods
- Sorts and hash
- Hints
- Parallel Query

Index Access
Index B-Tree



An index is a sorted structure

- Quick access to a value, or a range of values
- Several levels: branches are linked to lower branches or leaves
- Leaves have all entries and links (rowids) to table rows
- Leaves are a doubly linked list

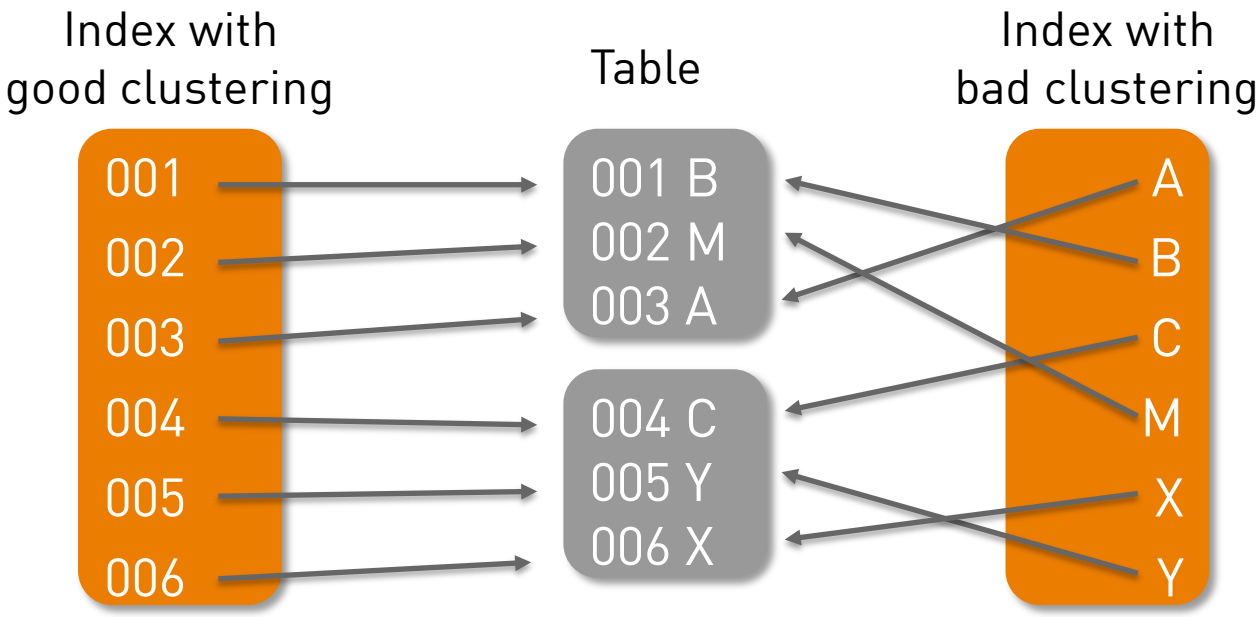
Index is maintained

- New rows have new entries in the structure
- Leaf or branch blocks may split when full
- When root is full, a split adds a new level
- Updating index value deletes the old value and adds the new one

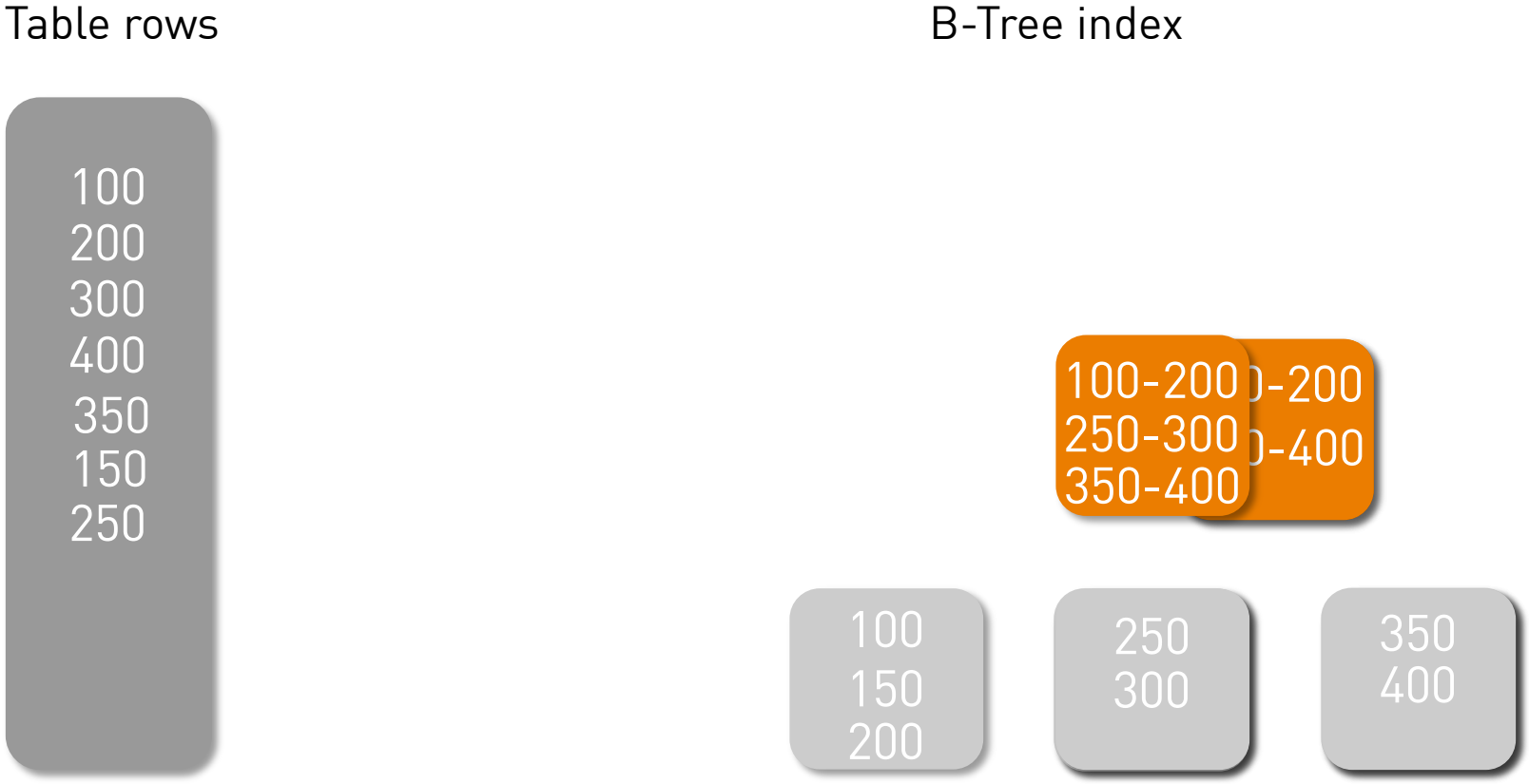
Index Access
Index Clustering factor

Table access by index rowid

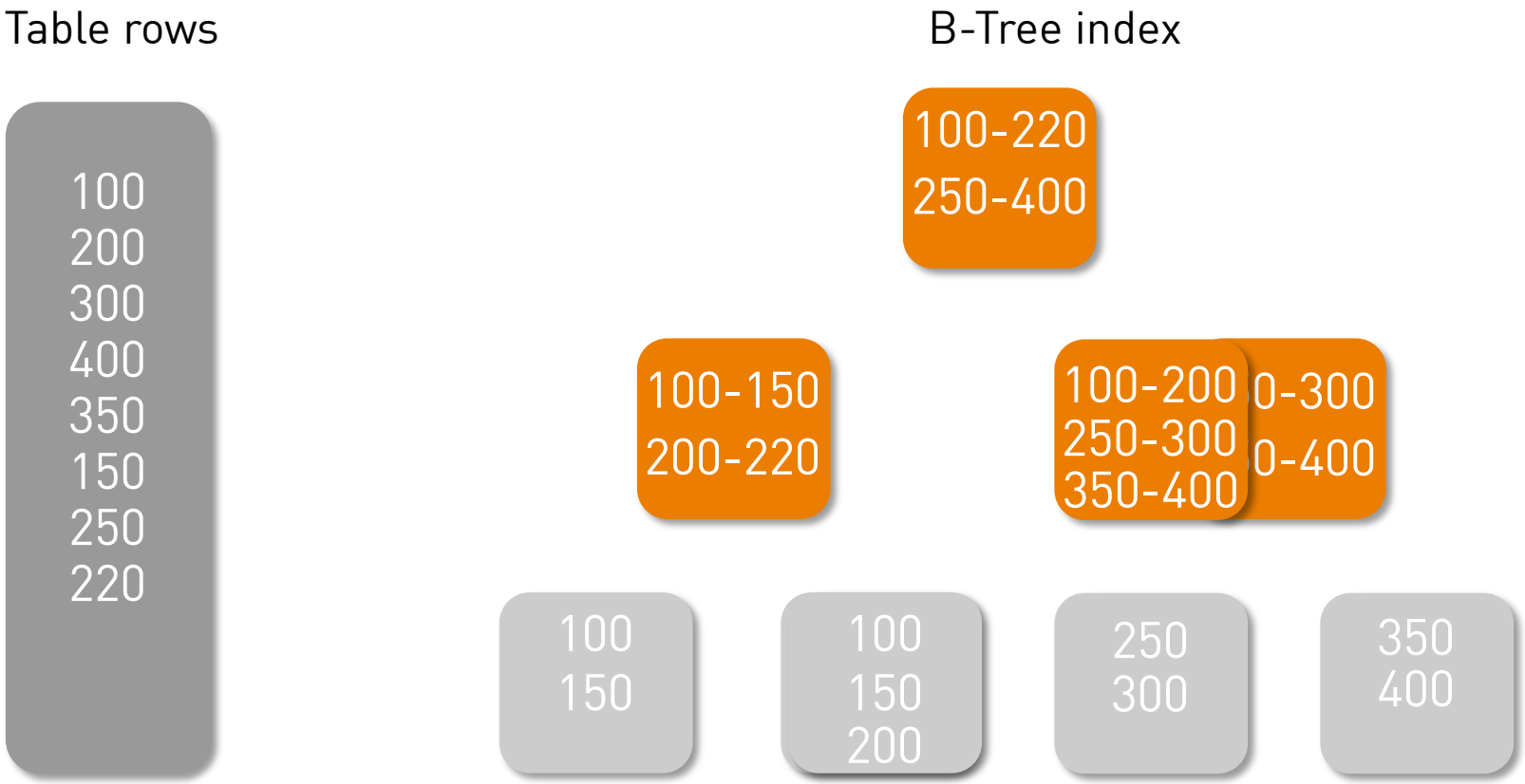
- Table rows coming from index entries may be scattered in the table
- Then each access is a new logical read
- If rows have been inserted from increasing values (sequence, date) they are probably better clustered on that value



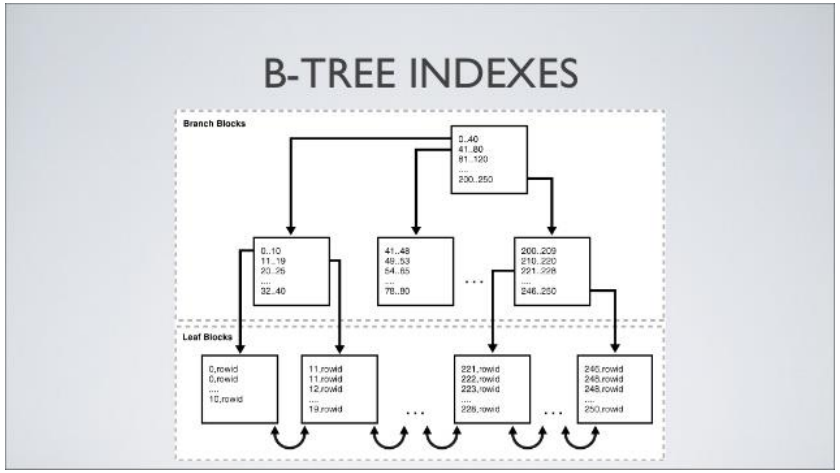
Index Access
Index B-Tree (leaf split)



Index Access
Index B-Tree (root split)



Index Access
Index efficiency



An index is good to get a few rows

- Quick access to the values we want
- Get required columns from the index (covering index)

But

- Getting additional columns from the table is expensive
- Especially when the clustering factor is bad

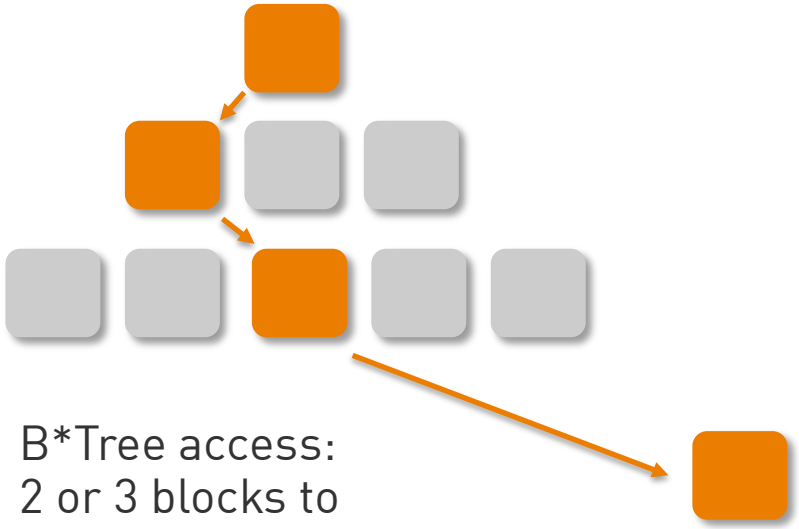
Look at the access predicate

- Functions prevent index access (except with Function Based Index)
- Implicit conversions are like functions. See the execution plan.
- Index order is binary – not ordered for non US7ASCII character set
- Null entries are not indexed (when all columns are null)

Index Access
Cost of index access

Index access vs. Table full Scan

Index access



- B*Tree access:
  2 or 3 blocks to access to an entry
  + one for the row

Table scan

- Full table scan:
  Reads all blocks
  efficient for lot of rows

Index Access
Demo: Index access

Run the following

```
SQL> set arraysize 5000
SQL> select /*+ gather_plan_statistics */ cust_id from CUSTOMERS where
cust_id=100;
SQL> select /*+ gather_plan_statistics */ cust_id from CUSTOMERS where
cust_id between 100 and 1000;
SQL> select /*+ gather_plan_statistics */ cust_id from CUSTOMERS where
cust_id between 1 and 1000 and cust_first_name='Amy';
```

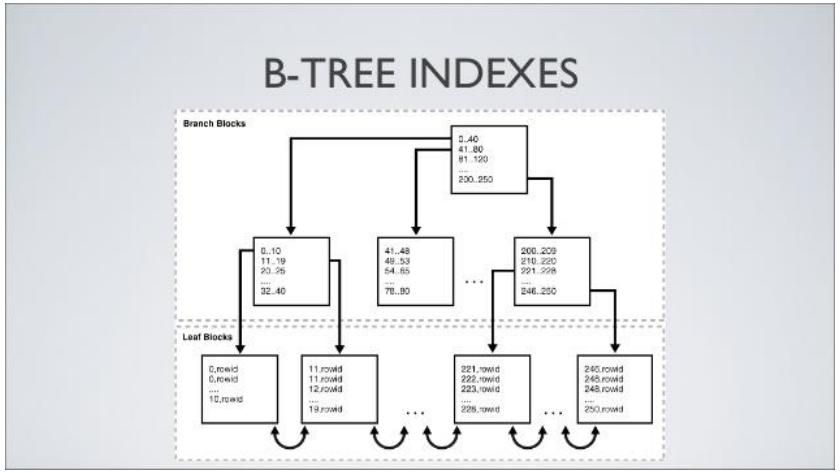
- And get the execution plan after each one:

```
SQL> select * from
table(dbms_xplan.display_cursor(format=>'allstats last'));
```

- Replace the CUSTOMER_PK(CUST_ID) index by CUSTOMER_DEMOINDEX(CUST_ID,CUST_FIRST_NAME) and run the last select

Optimization is done by filtering as soon as possible.

Index Access
Performance with proper Indexes



A good Indexing strategy is key to good performance

- Strategy: Index all that should be, and no more.
- Index Primary Keys and all Foreign Keys
- Remove duplicate indexes (Index on “Surname, Firstname, Age” makes the Indexes “Surname”,”Surname, Firstname” redundant)
- Column order is crucial (exception below)
- If all columns of the composite index are always used in predicates then the least selective may be first in list and compressed
- Consider Index-only-scan for very short select lists
- Check often used predicates for top SQL in the shared pool and AWR

- https://www.doag.org/formes/pubfiles/6663723/2015-01-News-Lothar-Flatz-Faktenbasierte-Indexierung-ein-Erfahrungsbericht.pdf

Index Access
Index alternatives

Hash cluster

> The rowid is derived from the value

Index Organized Tables

> Whole row is stored in the index structure

Bitmap indexes

> Single column indexes store bitmap of rowid

Storage indexes (exadata)

> Maps the blocks with their min/max values

In-Memory option

> Single column stored in memory for quick vector scan

Index Access
IOT and bitmap indexes

Index Organized Tables

> Whole row is stored in the index structure

> Except columns defined in overflow

> Secondary indexes are more expensive

> Use case: association table for many-to-many

Bitmap indexes

- Single column indexes store bitmap of rowid
- Cheap combination of several indexes
- But not suited for OLTP (degradation, locks)

Index Access
Row storage vs. Columnar storage

Row store vs Column store

> Analytic queries don't need all columns -> column store

> DSS Data Marts have denormalized columns -> compression

> Bitmap indexes are not for OLTP

Hybrid Columnar compression

> For oracle storage only (ZFS storage appliance, Exadata)

> Exadata can do SmartScan on it (storage index, predicate/projection offloading, decompression)

> **HCC is not for OLTP**

In-Memory option

> Allows columnar optimized operations (scan & update)

Index Access
Demo: Single Table Hash Cluster

Create cluster and table

```
SQL> connect demo/demo
SQL> create cluster DEMO_CLUSTER(CUST_ID number)
     size 5000 single table hashkeys 1000 ;
SQL> create table DEMO cluster DEMO_CLUSTER(CUST_ID) as
select * from CUSTOMERS;
```

- We have 50000 customers and no index

Select one customer by id

```
SQL> select /*+ gather_plan_statistics */ rowid,cust_id from
DEMO where cust_id=110;
SQL> select * from
table(dbms_xplan.display_cursor(format=>'allstats last'));
```

- Check how many logical reads
- You can try the same with smaller cluster size