Andrzej Barczak[1]
Dariusz Zacharczuk[1]
Damian Pluta[1]

[1] University of Natural Sciences and Humanities, Institute of Computer Science, 3 Maja 54, 08-110 Siedlce, Poland

Tools and methods for optimization of databases in Oracle 10g. Part 3 – theory in practice

Abstract. The article base on knowledge from earlier two parts. It shows how to effectively use the knowledge about database optimization methods, and how to verify and confirm the validity of the decisions in the process.

Keywords. database optimization, Oracle 10g, tools and methods

1. Introduction

The problem of increasing the efficiency applies to any database and is very complex. Optimizing the database must be done comprehensively. For optimum performance, you can not concentrate on only one part of the system. It is necessary to analyze the application and queries involved in the process of data collection. Only then it is possible to effectively carry out the optimization process. The author assumes that the reader has a basic knowledge of the management of memory, working with indexes, clusters and partitions in Oracle.

2. Database Optimization

The database, which will be optimized is base of online store selling tools. It contains about 10 000 customers and about 400 000 contracts, that have been accumulated over a period of four years. Estimated that yearly comes around 2 500 new customers and 100 000 orders. In store is about 1 000 products supplied by 10 different suppliers. Number of employees and product category is not specified. ERD diagram can be seen below.

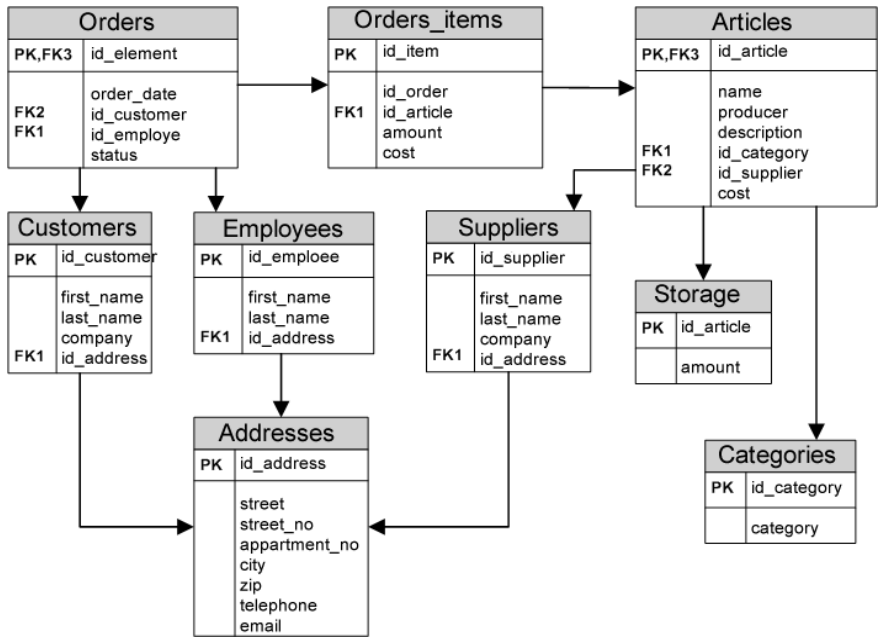

Figure 2.1 Datebase logical model

2.1. Configuring of SGA and PGA areas

The hardware platform on which the Oracle instance is running has 1024 MB of RAM. 512 MB of memory is necessary for a stable and efficient operation of the operating system and other applications other than Oracle. Thus for instance of Oracle (SGA and PGA areas) remains 512 MB of RAM. The suggested size of the PGA area is 20% of the SGA. Taking into account the available memory and the recommended proportions, the SGA area has been assigned 428 MB RAM, and 85MB for PGA area.

In accordance with the recommendation of the Oracle SGA and PGA memory is managed automatically by Oracle, which dynamically adjusts the size of the memory area to the changing needs and requirements during system operation.

2.2. Use of indexes

First, the indexes will be used to Suppliers and Employees tables. Due to the fact that these tables have a similar internal structure, store approximately equal amounts of records, and the nature of queries targeted to them are approximate the same, we can assume that the optimization process each table will be conducted in an identical manner. Optimization of these tables will be made on the example of the Suppliers table.

Suppliers table holds only 10 records. Each supplier has its own unique identifier id_supplier. This identifier is often mentioned in the WHERE clause and joins the Suppliers table with other tables. Therefore, on the column id_supplierr unique index was created:

CREATE UNIQUE INDEX suppliers_unique_index ON suppliers (id_supplier);

Queries that are usually addressed to the Suppliers table, choosing a single record base on id_supplier in equivalence condition specified in the WHERE clause:
SELECT * FROM suppliers WHERE id_supplier = 1;

In order to verify that for such a small table, which is the Suppliers table, access to data via the index will be more efficient than a full table scan, the performance of this query was analyze. TKPROF utility report shows the hardware resource consumption during execution of a query:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	l	0.00	0.00	0	0	0	0
Fetch	2	0.00	0.02	9	2	0	1
total	4	0.00	0.02	9	2	0	l

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS

Rows	Row Source Operation
1	TABLE ACCESS BY INDEX ROWID suppliers (cr=2 pr=9 pw=0 time=21478 us)
1	INDEX UNIQUE SCAN suppliers_unique_index (cr=1 pr=1 pw=0 time=11760 us)

We read that the optimizer choose an existing index as the optimal path to data. The first time you run the query, the required data was not in the data buffer, so the data must first be read from disk. Number of blocks read from disk was 9, and reference to the buffer was only 2. Execution time is .02s and it is mainly because of delays that have arisen during a physical read data from the disk. Analyzing the number of physical and logical reads, we can conclude, that the optimizer retrieving data from disk, downloaded index and collected not only the required record, but all the Suppliers table. In turn from the buffer has been read only index and search the record.
Report shows the consumption of resources by the same query at a 100-fold run:

call	count	cpu	elapsed	disk	query	current	rows
...							
total	400	0.01	0.05	9	200	0	l00

…

1	TABLE ACCESS BY INDEX ROWID suppliers
1	INDEX UNIQUE SCAN suppliers_unique_index

It should be noted that the number of physical reads is not increased, the next time

you run the query, only two logical reads were necessary. 100-fold time was 0.05s, 0.02s of with is the time of the first execution of a query. CPU time is 0.01s.
In order to compare the query using the index and no, the next query use FULL optimizer hint, which instructs the optimizer to use full table scan:
SELECT /*+ FULL(suppliers) */ * FROM suppliers WHERE id_supplier=7

TKPROF reports prepared after a single run and a 100+fold say, that the first time (full table scan), 6 physical reads from disk was necessary, and then 8 references to the data buffer. Execution time was .01s In the second case, the number reference to the drive has not increased, and each query subsequent run require 8 readings from the data buffer. 100-fold performance time was .07s, while the CPU time is used up .09 s. High values of the time is related to a very large number of references to the data buffer.

If the requested data is not in the buffer, full scan is more effectively solution, since it generates less physical reads from the disk and is carried out in a shorter time. This is due to the fact, that the access to the data using an index, the optimizer in the same way as it was for a full table scan, decided to collect all the Suppliers table from the disk, due to its small size, and in addition index had to be also downloaded. However, after 100 runs, much less resources absorbent solution becomes is the use the created index. It generates up to 600 fewer access to the buffer so it uses less CPU time and is faster. After loading all the required data from the disk, each subsequent start-up requires you to download only 2 blocks of data from the buffer: one block with the index and one block from the searched record. However, the next run of the query performing a full table scan, every time draw from the buffer all the Suppliers table. In the case of a small table, the use of full scan is a more efficient solution, if this query is performed sporadically and there is a high probability that data is not required in the data buffer.

Another table in the queue to optimize is Customers table, which consists about 10 000 records. Each client has its own unique identifier id_client. (high selectivity data), which often occurs in the WHERE clause and joins the Customers table to other tables. Therefore, for the Customers table a unique index on the column id_client was created. The effectiveness of this approach has been tested using the following query:

```
SELECT * FROM clients WHERE id_client = 6951;

results are as follows:
call     count       cpu    elapsed       disk      query    current        rows
...
total        4      0.00       0.00          0          3          0           1

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Rows     Row Source Operation
------  ---------------------------------------------------
1  TABLE ACCESS BY INDEX ROWID clients
1  INDEX UNIQUE SCAN clients_unique_index
```

Reading data from disk is not necessary – all the required data were in the buffer, which of occurred 3 readings. It was made one parsing queries – parsed form of query was not in the buffer library.

With a full scan (FULL hint optimizer) CPU time and execution time was of .01s, there was non reading from the disk and the number of references to the buffer was as high as 185. The use of the index largely eliminates the number of I/O. In this case, buffer contained all the required data. Reading from disk would cause even more noticeable difference.

Optimizing the table addresses, which consists more than 10 000 records, the situation is similar as in the previous cases. For a table of addresses a unique index on the column id_address was created. Most references to this table, are queries pursuing join on Customers, Employees and Suppliers tables. The effectiveness of the created index has been tested using the following query:

```
SELECT * FROM addresses a, suppliers d
WHERE d.id_supplier = 1 AND d.id_address = a.id_address;
```

TKPROF tool illustrates resources consumed during the execution of a query using the index and its omission. The query optimizer chose to use an existing index, as the more efficient access paths to the data. Query, in which access to the data held with the index, generated only 5 references to the data buffer, and the the one with full table scan generated up to 141. From information stored in the lines of query execution plans, it can be seen, that the times of query execution reached respectively 1709 μs and 2800 μs.

Table contract includes 400 thousand records, and a composite index on columns id_client and order_date, for the query:

```
SELECT * FROM orders
WHERE id_client = 2345
AND order_date > TO_DATE('01/01/2010', 'DD/MM/YYYY')
ORDER BY order_date DESC;
```

total execution time and consumed CPU time required for the query was less than .01s. There were no physical reads from the disk and only 23 references to the data buffer. Using optimizer hints NO_INDEX (omitting the existing index) CPU time and the total time of a query are .04s and .05s. To the results were comparable, all the necessary data were already in the data buffer, to which 2518 requests appeals. In addition, the fact in favor of the index is that, the values stored in the index are already sorted, and there is no need to re-sort the selected records, as is the case of full scan. When optimizing the orders_items table containing 1.2 million records that difference is even greater.

Let us return to the table articles. Due to the fact that data are often filtered by category and producer, bitmap index was used, which includes columns id_category and producer. Create index is a bitmap because the data stored in the indexed columns have a low selectivity (each column contains only about 10 unique values). And in these cases, the preferred type of the index is the bitmap index:

```
CREATE BITMAP INDEX cat_prod_index
ON towary(id_category, lower(producer));
```

Performance of the index was tested for the following query:

```
SELECT * FROM articles
WHERE id_category = 1 AND LOWER(producer) = LOWER('Hilti');
```

This query has been executed 100 times. At first run the required data was not in the data buffer – it required 16 physical reads from disk. Each subsequent execution require only four readings from the data buffer. The total CPU time used by the query was .18seconds, and the query execution time is .10s.
For comparison, is the bitmap index actual efficient than the basic type of index, the same request was made an identical number of times using a B-tree index. In this case, more efficient proved to be bitmap index. Number of physical reads from disk for both indices are identical, while a much larger number of references to the data buffer generated a query using the B-trees that also consumed more CPU time. Query execution times using the two indices are comparable.

For comparison, how will the bitmap index performance deal with increasing selectivity data in indexed columns, table articles was modified. Selectivity data in columns id_category and producer raised five times for each column. The TKPROF report shows that 100-fold query run using a bitmap index, required 7 physical reads from disk, 402 references to the data buffer, .32s CPU time and took .29s. However, the same query which use B-tree index also generated 7 readings from the disk, the references to the data buffer was about 100 more, consumed .21s CPU time and continued .14s.

For information with low selectivity, bitmap index takes up much less space. Bitmap index immediately obtain the appropriate ROWID address, in the case of B-tree index, it is necessary to go through the whole tree, which involves one I/O operations on each level of the tree. Therefore, in the first case, better results were obtained for a bitmap index.

In the second case, the increased selectivity also increased the number of bitmaps, on which base the ROWID addresses records are determined. The increase number of bitmaps did not cause increase number of I/O, but increased demand for CPU time required to determine the appropriate address records, which resulted in a significantly longer overall execution time. Thus, we confirmed correct selection of the index type: the bitmap index.

2.3. Uses of clustered table

Due to the fact that the tables orders and order_items are related by data and query that select data from them often make their concatenation, they are saved together in a cluster:

```
CREATE CLUSTER ord_orditem_cluster (id_order number(6));
CREATE INDEX idx_ord_orditem_cluster ON CLUSTER ord_orditem_cluster;

CREATE TABLE orders (…)
CLUSTER ord_orditem_cluster (id_order);
```

Common column for both tables is id_order, so this column is a cluster key that is indexed and is a clustered index. Cluster index replaced two separate indexes, which included columns id_order when tables were stored separately. For table orders, besides saving data in the cluster, composite index covering the columns id_client and order_date has been created.
In order to check whether an orders table and order_items in the form of a cluster will be more effective solution than the standard separable tables, below query was execute and analyzed in both cases:

```
SELECT z.id_order, z.order_date, z.id_client, z.id_emploee,
  (SELECT SUM(oi.amount * oi.cost)
   FROM order_items oi
   WHERE oi.id_order = z.id_order) AS suma
FROM orders z
WHERE z.id_client = 4649
AND z.order_date > TO_DATE('01/01/2010', 'DD/MM/YYYY')
ORDER BY z.order_date DESC;
```

The query returns the customer's orders with a given customer ID and the date of an order greater than the reference. As additional information for each of the selected orders, query displays the total amount of the order, which is calculated by the sub-query. TKPROF reports for the 50-fold run using the cluster:

call	count	cpu	elapsed	disk	query	current	rows
Parse	50	0.00	0.00	0	0	0	0
Execute	50	0.00	0.00	0	0	0	0
Fetch	100	0.17	0.22	27	1900	0	400
total	200	0.17	0.23	27	1900	0	400

```
Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Rows     Row Source Operation
------   ---------------------------------------------------
8        SORT AGGREGATE (cr=26 pr=16 pw=0 time=1379 us)
28       TABLE ACCESS CLUSTER order_items
         (cr=26 pr=16 pw=0 time=1379 us)
8 INDEX UNIQUE SCAN IDX)ord_orditem_cluster
         (cr=18 pr=16 pw=0 time=1190 us)
8 TABLE ACCESS BY INDEX ROWID orders
         (cr=12 pr=11 pw=0 time=638 us)
8 INDEX RANGE SCAN DESCENDING idcustomer_orderdate_index
         (cr=4 pr=3 pw=0 time=567 us)
```

and for separate tables:

call	count	cpu	elapsed	disk	query	current	rows
Parse	50	0.01	0.00	0	0	0	0
Execute	50	0.00	0.00	0	0	0	0
Fetch	100	0.23	0.57	33	2000	0	400
total	200	0.25	0.58	33	2000	0	400

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS

Rows	Row Source Operation
8	SORT AGGREGATE (cr=26 pr=20 pw=0 time=295585 us)
28	TABLE ACCESS BY INDEX ROWID order_items (cr=26 pr=20 pw=0 time=295419 us)
28	INDEX RANGE SCAN idorder_index (cr=18 pr=12 pw=0 time=259980 us)
8	TABLE ACCESS BY INDEX ROWID orders (cr=14 pr=12 pw=0 time=26135 us)
8	INDEX RANGE SCAN DESCENDING idcustomer_orderdate_index (cr=5 pr=4 pw=0 time=21826 us)

In both cases, by the first execution of a query, the blocks of the required data was not in the data buffer. Cluster request generated 27 physical reads from disk, 1900 references to buffer data, consumed 0.17s CPU time and returned results after .23s. For separate tables query has 33 reading from the disk, 2000 requests to the buffer, consumed .25 seconds of CPU time and returned results after .58s.
Query that uses cluster was more efficient in every way. To find the cause of this advantage, we need to look more closely at the implementation of the plans of the two queries.

The first two steps (as viewed from the bottom) of the two execution plans are identical - composite index on columns id_client and order_date is scanned. Then, based on designated by ROWID addresses, relevant data blocks are collected. However, steps 3 and 4 are now different for each execution plans. For query operates on tables stored separately, step 3 is scanned index on column id_order, and the result are 28 assigned addresses ROWID. In step 4, on the basis set out in the previous step, records are retrieved from the table order_items. On the other hand, for the query using the cluster, step 3 scans the clustered index and is set 8 ROWID addresses. In step 4 appropriate data blocks are taken. Because the data in the cluster associated with both tables are stored in the same block, the required data has been obtained in step 2 and there is no need to download any additional data blocks. The final step for both execution plans are identical, namely the aggregate function SUM() sums the appropriate values.

As can be seen from the foregoing, write tables orders and orders_items as the cluster is an effective solution. The data from these tables usually are taken in the junction, which is why the presence in the same block of related data records from both tables, can reduce the number of costly I/O operations, which entails a reduced memory and CPU time usage and less time waiting for results.

2.4. The use of partitioning table

Data from the table articles are usually selected by category and producer, the earlier in this paper, on the columns id_ category and producer of this table, composite bitmap index covering the columns was created. An alternative to the created index, listed partitioning by category in table orders can be applied. The efficiency of this solution will first be tested on a query that selects all records in this category. Here is an example of such a query:

```
SELECT *  FROM articles  WHERE id_category = 1;
```

These listings provide reports to the query using a solution based on partitioning:

```
call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.00       0.00          0          1          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        3      0.00       0.00          0          6          0          31
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        5      0.00       0.00          0          7          0          31

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Rows     Row Source Operation
-------  ---------------------------------------------------
     31  PARTITION LIST SINGLE PARTITION: KEY KEY
         (cr=6 pr=0 pw=0 time=23 us)
     31  TABLE ACCESS FULL articles PARTITION: 2 2
         (cr=6 pr=0 pw=0 time=17 us)
```

and bitmap index:

```
call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        3      0.00       0.00          0         12          0          31
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        5      0.00       0.00          0         12          0          31
```

```
Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Rows     Row Source Operation
------   ---------------------------------------------------
31       TABLE ACCESS BY INDEX ROWID articles
         (cr=12 pr=0 pw=0 time=79 us)
31       BITMAP CONVERSION TO ROWIDS (cr=3 pr=0 pw=0 time=156 us)
7 BITMAP INDEX RANGE SCAN cat_prod_index
         (cr=3 pr=0 pw-0 time=132 us)
```

For query choosing all records for one category, more efficient solution for data access is based on the partition, because it was only 7 necessary references to the data buffer, and in the case of a bitmap index it was 12. In both cases, all the required data contained in the data buffer. The reason for better performance using partitions, is a full scan of the partition, which contains only the records from the desired category, does not involve any additional I/O operations associated with the reading of data blocks storing the index and processing the index, as it is place for the solution using the index.

However, most data in the table articles are selected based on a double criteria, consisting of the category and producer. Therefore, the next performance comparison was carried out for the query filter records according to this criteria, with use of partitioning and then the index:

```
SELECT * FROM articles WHERE id_category = 1
AND LOWER(producer) = LOWER('Hilti');
```

Solution based on partitioning generated 9 references to buffer, all the data required were already in it. However, a query using an index generated only 4 references, and the other values were the same. Advantage of query, used as data path the existing complex bitmap index, is due to the fact, that the conditions in the WHERE clause coincide with columns, which includes an index. Based on the index ROWID addresses of relevant records were determined directly.
While in the case of partition-based solution, to determine the records, a full scan of the partition stored the records from the appropriate category, was needed, in order to select records of a certain producer.

From follows appears, that the partition table articles by category, is more efficient solution than accessing data from a complex bitmap index on columns id_category and producer only, if the data are selected according to a single criteria, which agrees with the partitioning scheme.

2.5. The use of index-organized tables

Table storage consists of about 1000 records, which store information about the availability of articles in stock. The primary key column in the table is id_article. Queries directed to that table often select individual records of a certain id_article, that's why id_article column should be indexed. Due to the fact that the table storage has only two columns: amount and id_article, creating an index for the column

id_article duplicate the half of the data, so the table storage was created as a index-organized table. The efficiency of this solution was compared to the yield of the standard solution, in which the index and table exist as two separate objects. The effectiveness of these two solutions was checked on the basis of the following query: SELECT amount FROM storage WHERE id_towaru = 837;

TKPROF report generated using index-organized table:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	0.00	0.00	0	2	0	1
total	4	0.00	0.00	0	2	0	1

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS

Rows	Row Source Operation
1	INDEX UNIQUE SCAN PK_INDEX (cr=2 pr=0 pw=0 time=32 us)

and ordinary table:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	0.00	0.00	0	4	0	1
total	4	0.00	0.00	0	4	0	1

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS

Rows	Row Source Operation
1	TABLE ACCESS BY INDEX ROWID storage(cr=4 pr=0 pw=0 time=31us)
1	INDEX RANGE SCAN id_article_index (cr=3 pr=0 pw=0 time=22 us)

Analyzing listings, we see, that query for a solution based on index-organized tables generated only 2 references to the data buffer, and the solution based on the standard notation table and index generated twice much. In both cases, the data contained in the data buffer, hence the very low execution time. Advantage of the first solution is due to the fact, that to obtain the desired data all we need is just scan of the index, which is stored as a table. In the second case (duplicated

the primary key), first it was necessary to scan index, and then access to a table based on the address obtained by ROWID.

For table storage, more efficient solution is to index-organized table, because it does not lead to duplicate primary key, which in the case of table storage, accounts for 50% of the data.

3. Summary

Based on the available methods and tools, optimize process were developed to the database and the obtained results were evaluated. The results of the research on the impact of individual solutions to increase the performance-optimized database allow us to conclude that:

- the use of an index on the column (or columns), which often occurs in the WHERE clause of queries and tables merging conditions, significantly reduces the number of generated I/O operations and allows to obtain results faster;
- for columns that contain data with high selectivity (eg unique identifiers records) efficient type of index is a B-tree index, while with the low selectivity - bitmap index;
- additional advantage of using indices is that the returned data using indexes are sorted by the index key;
- created tables with the related data as a cluster, reduces the number of I/O, the data in these tables are taken at the junction;
- divide table into partitions increases the efficiency of queries directed against a partitioned table, contains in the WHERE clause condition corresponding to the applied partitioning;
- created index-organized table provides faster random access to data based on the master key, than access using the standard separable index and table;
- index-organized tables use less disk space because the column which is the primary key, is not duplicated, which translates to a reduced number of I/O operations.

Presented optimization process has been carried out, for the current amount and specificity of the data. According to the assumptions, new records in the database were coming, which can also cause a change in characteristics of the stored data. Therefore, if one observed decreasing of productivity in the database, presented optimization process should be repeated to obtain best results once again.

With the knowledge contained in parts 1 and 2 and in this one, we are able to reduce the speed of access to data. The resulting time differences are dependent on the complexity of the query or the amount of collected data. Ranged from several microseconds to several IO operations to the tenth of a second and thousands of IO operations. Can performance improve at this level is actually important and worth the work involved? It depends primarily on who is the recipient of such data.

At the beginning take into consideration one of the popular web portal NK.pl, where the daily average logs on are about 500 thousand (data from 2011). In this case, the daily saving of time required to retrieve data from the database, can be up to 30 hours. This time translates into real savings e.g. purchase fewer servers.

If you're talking about cost much better example would be stock exchange. In a world where time is money fractions of a second count... literally. On Wall Street, decisions are taken in milliseconds. Speed ??is the key to success. United States have, however, several exchanges. In 2010, a company from Chicago dug under the cities, to put their own fiber to NY. Thanks to it, connection time between the cities decreased from 14,5ms to 13,1ms, which cost more than $100 million. Two years later, the fiber was already outdated and they built 22 ovens towers. Transferring data in this way is faster than fiber. The cost of relays was $30 million and reduced latency of 5ms. So every 1ms cost of $6 million. Now you see, what the speed might be worth.

References

1. A. Barczak, D. Zacharczuk, D. Pluta: Tools and methods of databases optimization in Oracle Database 10g. Part 1 – tuning instance, Publishing House of University of Natural Sciences and Humanities, 2012.
2. Alapati S.R.: Expert Oracle Database 10g Administration, Apress, 2005
3. Barczak A., Florek J., Sydoruk T.: Bazy danych, Wydawnictwo Akademii Podlaskiej, Siedlce 2007.
4. Greenwald R., Stackowiak R., Stern J.: Oracle Essentials: Oracle Database 10g, 3rd Edition, O'Reilly, 2004.
5. Lonley K., Bryla B.: Oracle Database 10g Podręcznik administrator baz danych, Helion, Gliwice 2008.
6. Taniar D., Rahayu J. W.: A Taxonomy of Indexing Schemes for Parallel Database Systems. Distributed and Parallel Databases, Volume 12, Number 1, Kluwer Academic Publishers, pp. 73-106, 2002.
7. Liebeherr J., Omiecinski E., Akyildiz I.F.: The Effect of Index Partitioning Schemes on the Performance of Distributed Query Processing. IEEE Transactions on Knowledge and Data Engineering archive, Volume 5, Issue 3, 1993, pp. 510-522.
8. Helmer S., Moerkotte G.: A performance study of four index structures for set-valued attributes of low cardinality. VLDB Journal, 12(3): pp. 244-261, October 2003.
9. Bertino E. et al.: Indexing Techniques for Advanced Database Systems. Kluwer Academic Publishers, Boston Dordrecht London, 1997.
10. Oracle: Oracle Database Administrator's Guide, 10g Release 2 (10.2), Dokumentacja techniczna, 2006.
11. Oracle: Oracle Database 10g Administration Workshop I, Dokumentacja techniczna, 2004.
12. Oracle: Oracle Database 10g Administration Workshop II, Dokumentacja techniczna, 2004.
13. Oracle: Oracle Database Concepts, 10g Release 2 (10.2), Dokumentacja techniczna, 2005.
14. Oracle: Oracle Database Data Warehousing Guide, 10g Release 2 (10.2), Dokumentacja techniczna, 2005.

15. Oracle: Oracle Database Performance Tuning Guide, 10g Release 2 (10.2), Dokumentacja techniczna, 2008
16. Oracle: Oracle Database SQL Reference 10g Release 2 (10.2), Dokumentacja techniczna, 2005.
17. Oracle PL/SQL Database Code Library and Resources, [online] http://psoug.org/reference/library.html.
18. Tow D.: SQL. Optymalizacja, Helion, Gliwice 2004.
19. Urman S., Hardman R., McLaughlin M.: Oracle Database 10g. Programowanie w języku PL/SQL, Helion, Gliwice 2007.
20. Whalen E., Schroeter M.: Oracle Optymalizacja wydajności, Helion, Gliwice 2003.