Andrzej BARCZAK,
Dariusz ZACHARCZUK,
Damian PLUTA

Siedlce University of Natural Sciences and Humanities,
Institute of Computer Science,
ul. 3 Maja 54, 08-110 Siedlce, Poland

Methods of optimization of distributed databases in oracle – part 2

Abstract. The second part of the paper devoted to optimization of distributed databases. This part presents tests which confirm the efficiency of database tuning methods, described in part one. Analysis of tests results based on the developed database is presented.

Keywords: optimization, distributed databases, Oracle, tests

1. Introduction

Part one of the paper describes the steps to optimally tune a distributed database and presents methods and tools, and how to use them. Part two applies theoretical knowledge to practice and analyses the results of performed tests.

In the process of testing the efficiency of the database for dedicated and shared service processes, the main node of a distributed database ORADB1 was used. In this node, two service names were defined:

- ora_db1 – enables connection using a dedicated server process;
- ora_db1shared – enables connection in the shared server process mode;

To simulate the load on a database generated by connected clients, a client application named ConnectOracle.jar was implemented in Java, which, at regular intervals, randomly

executes one of four queries with random parameters. A specified number of clients is activated by a batch program named run_klient.bat that activates ConnectOracle.jar. The use of two batch programs allows for simultaneous activation of any number of applications. With only one program, the batch script would wait for the running application to terminate. Load measurements were performed immediately after all clients started running, and lasted 1 hour. Each test was repeated thrice, and the results of the most representative tests were taken into account.

Fig. 1 below shows a logical model of the TELEKOM distributed database.

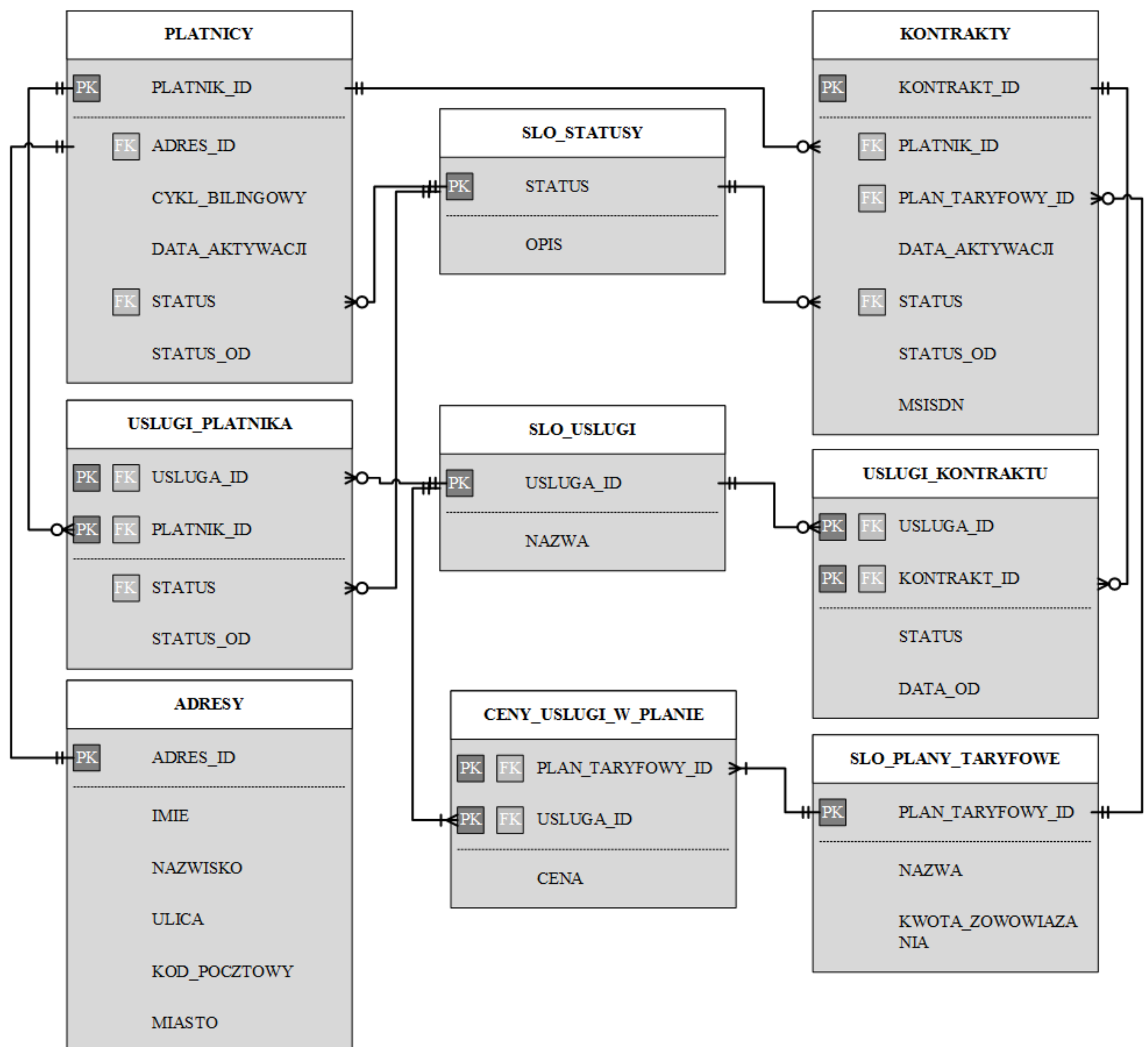

Figure 1. The TELEKOM database model.

The queries are of the following form: [2],[3],[4],[5]

1. selects contracts that activated a random service (service id from 1 to 10) on a random day during the year (range from „today”-2*365 days till „today”):

```
SELECT
    K.MSISDN, UK.DATA_AKTYWACJI, 'AKTYWNA' AS STATUS,
    (SELECT U.NAZWA FROM TELEKOM.SLO_USLUGI U WHERE U.USLUGA_ID = UK.USLUGA_ID) AS
NAZWA_USLUGI
FROM TELEKOM.KONTRAKTY K, TELEKOM.USLUGI_KONTRAKTU UK
WHERE
    K.KONTRAKT_ID = UK.KONTRAKT_ID AND
    UK.USLUGA_ID = (SELECT CEIL(DBMS_RANDOM.VALUE(0,10))
       FROM DUAL) AND
    TRUNC(UK.DATA_AKTYWACJI, 'DD') = (SELECT TRUNC(SYSDATE -
       DBMS_RANDOM.VALUE(0,2*365), 'DD') FROM DUAL) AND
    UK.STATUS = (SELECT S.STATUS FROM TELEKOM.SLO_STATUSY S
       WHERE S.OPIS = 'AKTYWNY');
```

2. returns the number of times services were activated on a random day (range from „today”-2*365 days till „today”) broken into separate services:

```
SELECT
 A.DATA_AKTYWACJI,
 U.NAZWA AS NAZWA_SULUGI,
 A.LICZBA
FROM
    (SELECT
        UK.DATA_AKTYWACJI,
        UK.USLUGA_ID,
        COUNT(*) AS LICZBA
     FROM
        TELEKOM.USLUGI_KONTRAKTU UK
     WHERE
          TRUNC(UK.DATA_AKTYWACJI, 'DD') =
        (SELECT TRUNC(SYSDATE - DBMS_RANDOM.VALUE(0,2*365),
           'DD') FROM DUAL)
    GROUP BY
        UK.DATA_AKTYWACJI, UK.USLUGA_ID
    ORDER BY
        UK.DATA_AKTYWACJI, UK.USLUGA_ID) A,
    TELEKOM.SLO_USLUGI U
WHERE
    A.USLUGA_ID = U.USLUGA_ID;
```

3. returns all services for random MSISDN (range from 5000000 to 6000000):

```
SELECT
    A.KONTRAKT_ID,
    A.MSISDN,
    U.NAZWA AS USLUGA,
    S.OPIS AS STATUS_USLUGI,
    A.STATUS_OD,
```

```
    A.DATA_AKTYWACJI
FROM
    (SELECT
        K.KONTRAKT_ID,
        K.MSISDN,
        K.PLAN_TARYFOWY_ID,
        UK.USLUGA_ID,
        UK.STATUS,
        UK.STATUS_OD,
        UK.DATA_AKTYWACJI
    FROM
        TELEKOM.KONTRAKTY K,
        TELEKOM.USLUGI_KONTRAKTU UK
    WHERE
        K.KONTRAKT_ID = UK.KONTRAKT_ID AND
        K.MSISDN = (SELECT CEIL(DBMS_RANDOM.VALUE(4999999,6000000))
           FROM DUAL)) A,
    TELEKOM.SLO_USLUGI U,
    TELEKOM.SLO_STATUSY S
WHERE
    A.USLUGA_ID = U.USLUGA_ID AND
    A.STATUS = S.STATUS;
```

4. returns all contracts of a random payer (range from 1 to 200000):

```
SELECT
    A.PLATNIK_ID,
    A.CYKL_BILINGOWY,
    A.MSISDN,
    PT.NAZWA AS PLAN_TARYFOWY,
    S.OPIS AS STATUS,
    A.DATA_AKTYWACJI
FROM
    (SELECT
        P.PLATNIK_ID,
        P.CYKL_BILINGOWY,
        K.MSISDN,
        K.PLAN_TARYFOWY_ID,
        K.STATUS,
        K.DATA_AKTYWACJI
    FROM
        TELEKOM.PLATNICY P,
        TELEKOM.KONTRAKTY K
    WHERE
        P.PLATNIK_ID = K.PLATNIK_ID AND
        P.PLATNIK_ID = (SELECT CEIL(DBMS_RANDOM.VALUE
           (0,200000)) FROM DUAL)) A,
    TELEKOM.SLO_PLANY_TARYFOWE PT,
    TELEKOM.SLO_STATUSY S
WHERE
    A.STATUS = S.STATUS AND
    A.PLAN_TARYFOWY_ID = PT.PLAN_TARYFOWY_ID;
```

2. Tests of service processes

The database load tests involved simulating load on the database by connecting 100 clients using dedicated or shared service processes. Each client connected to the database every 20 seconds. In time intervals between query execution, clients' sessions were idle. Successive instances of client applications were run sequentially with 2-second intervals in order to balance the load on the database.

2.1. Tests of dedicated service processes

Tests of dedicated service processes involved connecting clients to ORADB1 database using ora_db1 name service. The results of tests are presented below.

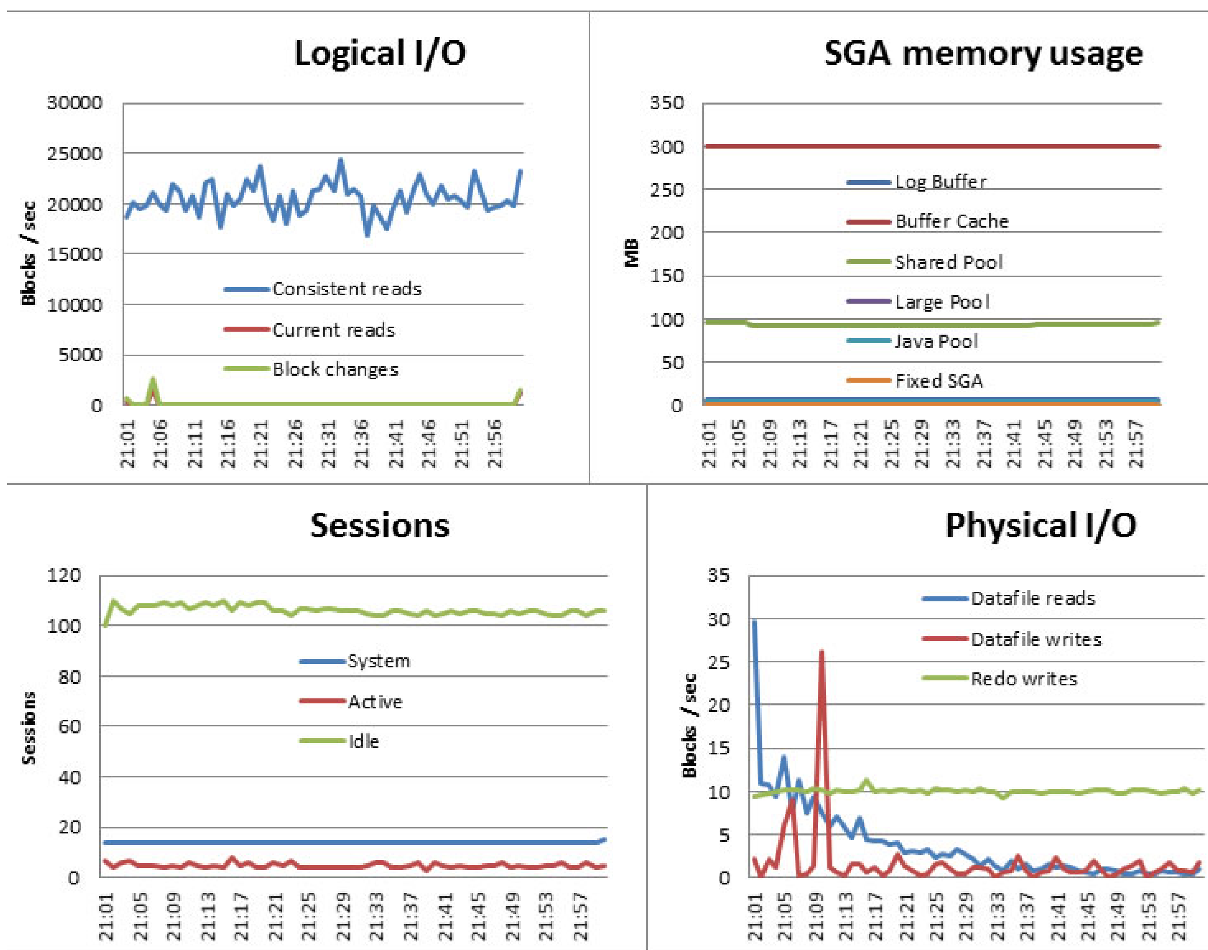

Figure 2. Tests of dedicated service processes

The buffer hit rate was 99.13%:

```
SELECT
    ROUND((1-(A.PHYSICAL_READS /(A.DB_BLOCK_GETS + A.CONSISTENT_GETS)))*100 , 2)
FROM
    (SELECT
        (SELECT VALUE FROM V$SYSSTAT WHERE NAME IN ('db block gets')) AS DB_BLOCK_GETS,
        (SELECT VALUE FROM V$SYSSTAT WHERE NAME IN ('consistent gets')) AS CONSISTENT_GETS,
        (SELECT VALUE FROM V$SYSSTAT WHERE NAME IN ('physical reads')) AS PHYSICAL_READS
    FROM DUAL) A;
```

PGA memory usage was as follows:

```
SELECT
    'TELEKOM' AS SCHEMA, ROUND(SUM(VP.PGA_USED_MEM)/1000000, 2) AS PGA_USED_MEMORY,
    ROUND(SUM(VP.PGA_ALLOC_MEM)/1000000, 2) AS PGA_ALLOCATED_MEM
FROM V$SESSION VS, V$PROCESS VP
WHERE VS.PADDR = VP.ADDR AND VS.USERNAME = 'TELEKOM'
UNION ALL
SELECT  'ALL_SCHEMA' AS SCHEMA,  ROUND(SUM(VP.PGA_USED_MEM)/1000000, 2)AS PGA_USED_MEMORY,
    ROUND(SUM(VP.PGA_ALLOC_MEM)/1000000, 2) AS PGA_ALLOCATED_MEM
FROM V$SESSION VS, V$PROCESS VP
WHERE VS.PADDR = VP.ADDR;
```

SCHEMA	PGA_USED_MEMORY [MB]	PGA_ALLOCATED_MEM [MB]
TELEKOM	64.81	101.59
ALL_SCHEMA	84.74	137.13

Average CPU usage was approx. 45%, while average query execution times were 0.41, 0.58, 0.11, 0.12 seconds, respectively.

2.2. Tests of shared service processes

Tests of shared service processes involved connecting clients to the ORADB1 database using ora_db1shared service name. Fig. 2 presents printscreens of key database performance parameters.

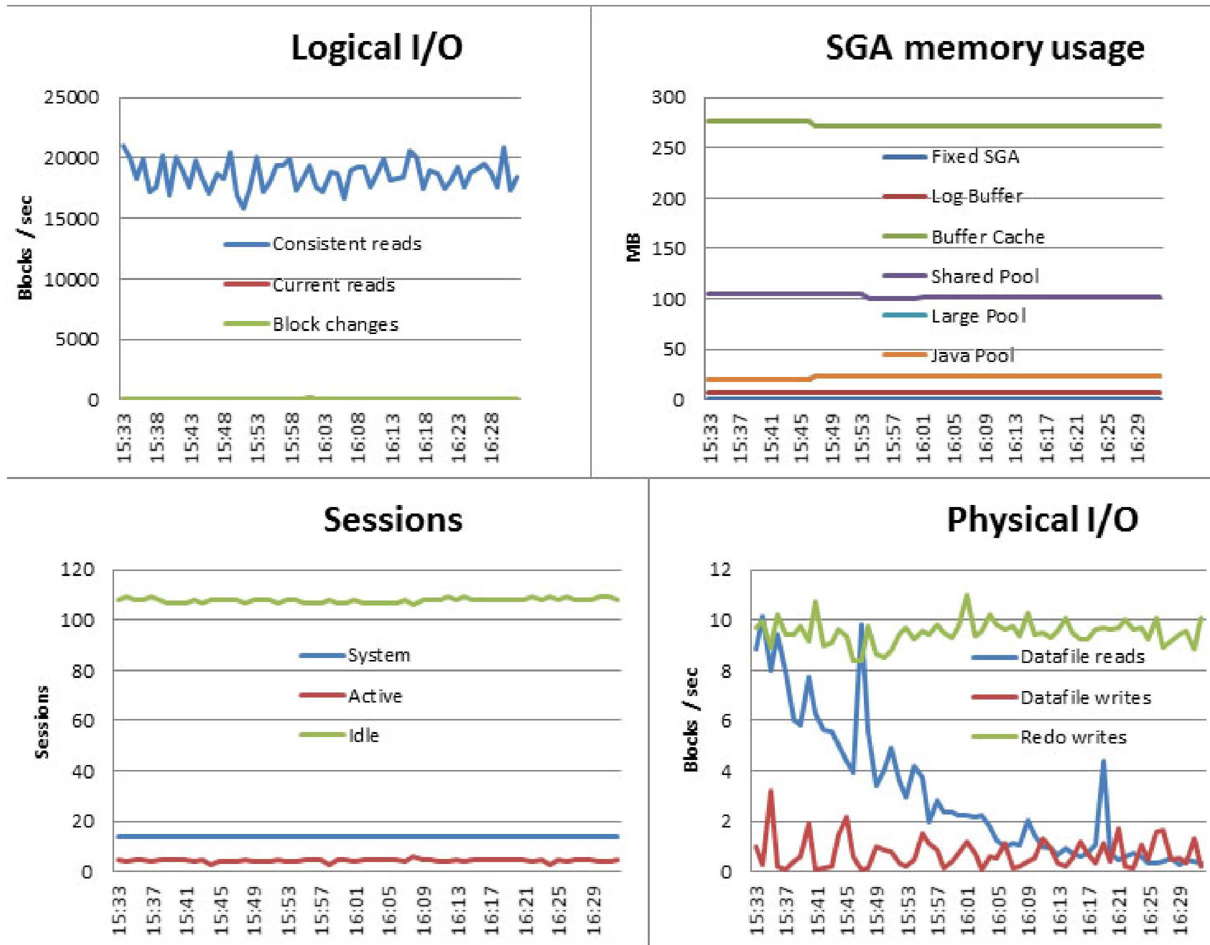

Figure 3. Test of shared service processes

Average CPU usage was approx. 40%, buffer hit rate 99.95%, and average query execution times were 0.98, 1.22, 0.20 and 0.29, respectively. PGA memory usage was as follows:

SCHEMA	PGA_USED_MEMORY [MB]	PGA_ALLOCATED_MEM [MB]
TELEKOM	1.01	1.74
ALL_SCHEMA	26.82	47.91

Dispatcher processes busy rate:

```
SELECT D.NAME AS DISPATCHER_NAME,
    ROUND((D.BUSY/(D.BUSY+D.IDLE))*100, 2) AS TOTAL_BUSY_RATE,
    ROUND(Q.WAIT/Q.TOTALQ, 2) AS AVERAGE_WAIT
FROM V$QUEUE Q, V$DISPATCHER D
WHERE Q.TYPE = 'DISPATCHER' AND Q.PADDR = D.PADDR;
```

DISPATCHER_NAME	TOTAL_BUSY_RATE [%]	AVERAGE_WAIT [sec.]

D000	1.12	0.04

Load on the queue of tasks to be handled:

```
SELECT Q.PADDR AS QUEUE_NAME, Q.TYPE, Q.QUEUED AS ITEMS_QUEUED, Q.WAIT AS TOTAL_TIME_WAITED,
    Q.TOTALQ AS TOTAL_ITEMS_PROCESSED,ROUND((Q.WAIT/Q.TOTALQ)/100, 2) AS AVERAGE_WAIT
FROM V$QUEUE Q
WHERE Q.TYPE = 'COMMON';
```

QUEUE_NAME	TYPE	ITEMS_QUEUED	AVERAGE_WAIT [sec.]
00	COMMON	4	0.21

Virtual circuits status:

```
SELECT C.QUEUE AS QUEUE_STATUS,COUNT(*) AS ITEMS
FROM V$CIRCUIT C GROUP BY C.QUEUE;
```

QUEUE_STATUS	ITEMS
SERVER	1
NONE	95
COMMON	4

Load on shared server processes:

```
SELECT SS.NAME AS SERVER_NAME, ROUND((SS.BUSY/(SS.BUSY+SS.IDLE))*100, 2) AS TOTAL_BUSY_RATE
FROM V$SHARED_SERVER SS;
```

SERVER_NAME	TOTAL_BUSY_RATE [%]
S000	53.97

2.3. Analysis of the results of service processes tests

- Dispatcher processes busy rate: the database was running with one dispatcher process (D000), for which busy rate (TOTAL_BUSY_RATE) was only 1.12%, and average wait time (AVERAGE_WAIT) equalled 0.04 seconds, which means that the dispatcher process was idle most of time, and incoming tasks were handled in a flash. Therefore, the configuration of dispatcher processes was correct.
- Load on the queue of tasks to be handled: the database was running with one queue (00), for which the average wait time for process assignment was 0.21 seconds and the number of items queued was 4. The values of both parameters are acceptable.

Therefore, a conclusion might be drawn that the current number of shared service processes is sufficient.

- Virtual circuits status: V$CIRCUIT view presents the following information: at the moment of taking measurements one virtual circuit is being handled by the server process, 4 circuits, 4 circuits wait to be handled by the server process and 95 are idle. The above information, gathered using this view, also confirms the correctness of the configuration of shared service processes in the database.
- Load on shared server processes: the database was running with one shared server process (S000), for which the TOTAL_BUSY_RATE was 53.97%, i.e. near the value for which, depending on the expected reaction of the database, one might consider increasing the number of shared server processes.
- CPU usage: CPU usage for the test of dedicated and shared server processes was approx. 45% and 40%, respectively. For the test of dedicated server processes, a separate server process utilizing hardware resources was initiated for each session. In contrast, the test of shared processes involved handling all clients' requests by one process using such mechanisms as dispatchers, virtual circuits and queue of tasks to be handled.
- SGA memory usage: in both tested cases, automatic SGA memory management mechanism used the whole space available for this area (424 MB), and divided it between each subarea in similar proportions – reallocation of memory between individual subareas result from different modes of database operation.
- PGA memory usage: in this area of performance, the difference is most visible, and results from the operation of database in different modes. For dedicated server processes, the memory allocated by the processes handling the session of user TELEKOM was approx. 102MB of PGA memory, while for shared processes, all sessions of the user were handled by one process which allocated less than 2MB – 50 times less memory than in the first case.
- Logical I/O operations: for the test of dedicated processes, the number of logical reads (consistent reads) was approx. 2 thousand blocks per second larger as a result of shorter query execution times, and consequently, the execution of a larger number of queries in the same time.
- Physical I/O operations: the number of physical operations for both tests was close to zero, which results from the fact that all required data had been loaded to the data buffer.
- Number of sessions: for both tests the number of active and idle sessions was the same, which indicates that the simulated load on the database was identical for both tests.
- Buffer hit rate: was greater than 99% for both tests, which indicates that all data were in the data buffer.
- Query execution times in the test of shared processes lasted approximately two times longer than execution times of the same queries in the test of dedicated service processes. Longer query execution times were caused by the fact that more than one task waited to be handled at a given moment. Contrastingly, in the case of dedicated processes, each client (process) is assigned a separate server process that immediately handles its request.

2.4. Summary for service processes

Based on the conducted tests, it may be concluded that 100 clients handled by 100 dedicated server processes may be handled by only one shared server process for the base operating in shared server processes mode. The above is true for the assumptions made for the purpose of the tests, with the fundamental one being that the client performs a random query every 20 seconds, and his session remains idle for the rest of the time.

Replacing 100 dedicated processes with only one shared process resulted in 50-fold decrease in PGA memory usage and 5% decrease in CPU time. The remaining database performance parameters for both tests were similar.

Replacing 100 processes waiting for incoming requests with only one resulted in longer query execution times, which, for shared server process, lasted twice as long as for the dedicated processes, which is a direct consequence of incoming requests queuing. Therefore, depending on the expected reaction of the database, one should use, e.g. 2 or 5 shared server processes instead of 1, which also results in significant reduction of database hardware resources use while maintaining required request handling times. [8],[10]

3. Tests of database links

The main node ORADB1, at which all tables of the test database are located, and ORADB2, from which clients referred using links, were used in the process of testing links. Two name services, analogous to the name services in the ORADB1 node, were defined in the ORADB2 node:

- ora_db2 – connection with the base using a dedicated server process;
- ora_db2shared – connection with the base in the shared server process mode.

The following database links, defined in the ORADB2, enabled connection between the databases:

- db1_public_link – a public link that enables the connection with the ORADB1 base in the dedicated service processes mode as a user named TELEKOM;
- db1_shared_link – a public shared link that enables the connection with the ORADB1 base in the dedicated service processes mode as a user named TELEKOM;
- db1sh_public_link – a public link that enables the connection with the ORADB1 base in the shared service processes mode as a user named TELEKOM;

- db1sh_shared_link – a public shared link that enables the connection with the ORADB1 base in the shared service processes mode as a user named TELEKOM.

To simulate load on the databases, generated by connected clients, a modified version of the previous application was used. As in the previous case, batch programs were used to activate the application in this test, and a set of the same four queries, which now use links, e.g.:

```
… FROM
    KONTRAKTY@DB1SH_PUBLIC_LINK K,
    USLUGI_KONTRAKTU@DB1SH_PUBLIC_LINK UK
…
```

Tests of database links were performed using a sample of 5 clients – a small number of clients allowed for detailed analysis of their sessions and related server processes. Each client connected with the ORADB2 database and performs a random query in an infinite loop. Queries are executed with no time intervals between them.

3.1. Tests of public and shared database links.

Tests of public database links involved clients connecting in shared processes mode. Queries referred to tables located in a remote database ORADB1 via a public database link *db1sh_public_link*, indicating the scheme of a user named TELEKOM and a name service *ora_db1shared* in the ORADB1 base. Use of *ora_db1shared* allowed for connecting to the ORADB1 base in the shared server processes mode.

Tests of shared database links were performed analogically, with the difference being the type of the database link used – shared link named *db1sh_shared_link* was used.

The results of the tests are as follows:

Dispatchers processes busy rate:

LINK TYPE	DISPATCHER_NAME	TOTAL_BUSY_RATE [%]	AVERAGE_WAIT [sec.]
Public	D000	34.01	0.03
Shared	D000	25.13	0.03

Load on the queue of tasks to be handled:

LINK TYPE	QUEUE_NAME	TYPE	ITEMS_QUEUED	AVERAGE_WAIT [sec.]
Public	00	COMMON	2	0.04
Shared	00	COMMON	0	0.02

Virtual circuits status:

QUEUE_STATUS	ITEMS – public links	ITEMS – shared links
SERVER	2	3
NONE	2	2
COMMON	1	n/a

Shared processes busy rate:

SERVER_NAME	TOTAL_BUSY_RATE [%]	
	Public Links	Shared Links
S000	28.15%	50.94%
S002	31.94%	51.67%
S003	28.30%	53.28%
S004	26.21%	51.58%
S005	35.34%	49.35%

Graphs of event waits:

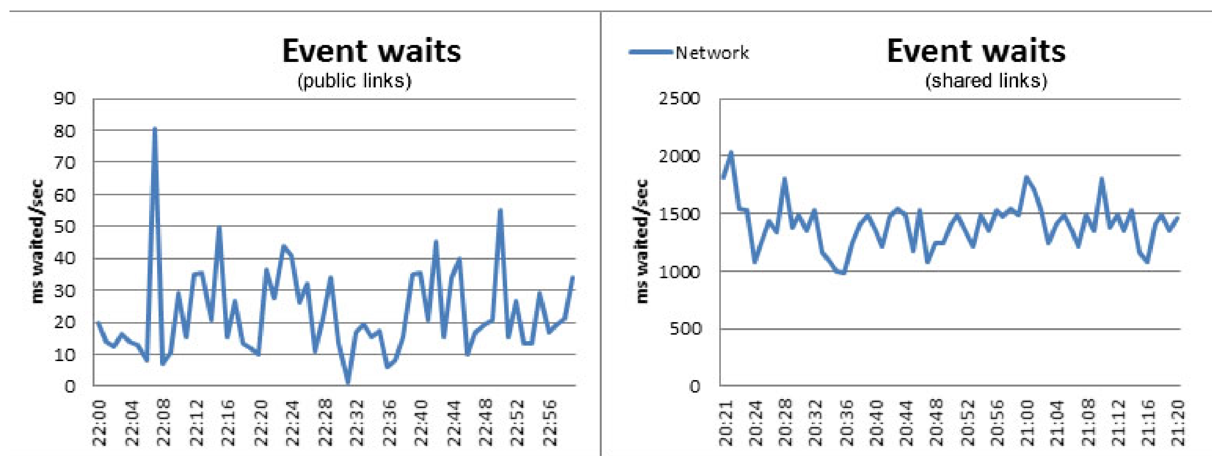

Figure 4. Test of links – event waits

Average query execution times [sec]

a) Public links: 5.08, 10.21, 26.34 and 17.34, respectively.
b) Shared links: 3.48, 13.66, 41.53 and 20.90, respectively.

Sessions and assigned server processes for public links in the *ORADB2* base:

```
SELECT
    VS.SID, VS.SERIAL#,VS.SERVER, VS.SERVICE_NAME, VS.STATUS, VS.USERNAME, VP.PID, VP.SPID,
    VP.PROGRAM
FROM
    V$SESSION VS, V$PROCESS VP
WHERE
```

```
VS.PADDR = VP.ADDR AND VS.USERNAME = 'TELEKOM';
```

SID	SERIAL#	SERVER	SERVICE_NAME	STATUS	USERNAME	PID	SPID	PROGRAM
SID	SERIAL#	SERVER	SERVICE_NAME	STATUS	USERNAME	PID	SPID	PROGRAM
133	149	SHARED	ORA_DB2SHARED	ACTIVE	TELEKOM	18	3848	ORACLE.EXE (S000)
130	244	SHARED	ORA_DB2SHARED	ACTIVE	TELEKOM	21	3404	ORACLE.EXE (S001)
138	92	SHARED	ORA_DB2SHARED	ACTIVE	TELEKOM	23	3024	ORACLE.EXE (S002)
139	31	SHARED	ORA_DB2SHARED	ACTIVE	TELEKOM	27	3744	ORACLE.EXE (S003)
140	20	SHARED	ORA_DB2SHARED	ACTIVE	TELEKOM	28	1812	ORACLE.EXE (S004)

Sessions and assigned server processes for public links in the *ORADB1* base:

SID	SERIAL#	SERVER	SERVICE_NAME	STATUS	USERNAME	PID	SPID	PROGRAM
149	56	NONE	ORA_DB1SHARED	INACTIVE	TELEKOM	18	3848	ORACLE.EXE (S000)
134	18	NONE	ORA_DB1SHARED	INACTIVE	TELEKOM	21	3404	ORACLE.EXE (S001)
138	30	SHARED	ORA_DB1SHARED	INACTIVE	TELEKOM	23	3024	ORACLE.EXE (S002)
133	24	SHARED	ORA_DB1SHARED	ACTIVE	TELEKOM	27	3744	ORACLE.EXE (S003)
156	42	SHARED	ORA_DB1SHARED	INACTIVE	TELEKOM	28	1812	ORACLE.EXE (S004)

Sessions and assigned server processes for shared links in the *ORADB2* base:

SID	SERIAL#	SERVER	SERVICE_NAME	STATUS	USERNAME	PID	SPID	PROGRAM
141	28	SHARED	ORA_DB2SHARED	ACTIVE	TELEKOM	14	3216	ORACLE.EXE (S000)
137	1	SHARED	ORA_DB2SHARED	ACTIVE	TELEKOM	26	3108	ORACLE.EXE (S001)
139	1	SHARED	ORA_DB2SHARED	ACTIVE	TELEKOM	27	3220	ORACLE.EXE (S002)
138	2	SHARED	ORA_DB2SHARED	ACTIVE	TELEKOM	28	2552	ORACLE.EXE (S003)
134	6	SHARED	ORA_DB2SHARED	ACTIVE	TELEKOM	29	2368	ORACLE.EXE (S004)

Sessions and assigned server processes for shared links in the *ORADB1* base:

SID	SERIAL#	SERVER	SERVICE _NAME	STATUS	USER NAME	PID	SPID	PROGRAM
139	57	NONE	ORA_ DB1SHARED	INACTIVE	TELEKOM	13	3080	ORACLE.EXE (D000)
137	13	SHARED	ORA_DB1SHARED	INACTIVE	TELEKOM	17	4040	ORACLE.EXE (S001)
135	4	SHARED	ORA_ DB1SHARED	ACTIVE	TELEKOM	17	4040	ORACLE.EXE (S001)
132	5	SHARED	ORA_ DB1SHARED	ACTIVE	TELEKOM	20	3876	ORACLE.EXE (S000)
130	12	SHARED	ORA_ DB1SHARED	INACTIVE	TELEKOM	20	3876	ORACLE.EXE (S000)
136	1	SHARED	ORA_ DB1SHARED	INACTIVE	TELEKOM	29	2200	ORACLE.EXE (S004)
133	1	SHARED	ORA_ DB1SHARED	ACTIVE	TELEKOM	29	2200	ORACLE.EXE (S004)
158	9	SHARED	ORA_ DB1SHARED	INACTIVE	TELEKOM	30	1496	ORACLE.EXE (S003)
134	1	SHARED	ORA_ DB1SHARED	ACTIVE	TELEKOM	30	1496	ORACLE.EXE (S003)

3.2. Analysis of shared database links

The results of database links provide the following information about the performance of the database and shared service processes work: [7],[10]

- Busy rate of shared service processes of the local database (ORADB2) – this aspect was examined for the remote database (ORADB1). As for the local base, it was assumed that the request rate during each test was at the same level. During each of the tests, 5 test clients were handled by 5 shared server processes, which results from the fact that clients' sessions were active all the time.
- Busy rate of shared service processes of the remote base – the incoming requests sent to the base using a database link were also handled by 5 shared server processes. It ought to be noted that busy rate of shared server processes was higher for the shared link, which is indicated by approx. 20% higher busy rate.
- Query execution times – average query execution times were higher by approx. 20% for shared database links.
- Event waits – rate of event waits of all sessions for the network was approx. 30 ms per second while for the shared link it was almost 50-fold higher and equalled 1400ms.

3.3. Summary for database links

Query execution times differ by one order of magnitude. Such a dramatic increase results from the fact that the queries were not optimized as regards the execution in the distributed database, which resulted in multiple references to the same remote database within one query. Data obtained in this way are then processed in the local database, which results in transferring large amounts of redundant data over the network.

During tests of database links, it was observed that query execution times for shared database link are 20% longer than for public (non-shared) link. Moreover, the busy rate of shared server processes in a remote database was also approx. 20% higher despite the identical traffic volume generated by test clients in both cases. The above observation may be explained by event waits, whose values were almost 50-fold higher for shared link, which caused shared processes of a remote database server to wait for the network to send the results.

Such a great difference in time waits is a result of sharing network connections, which were not capable of providing the required capacity in the case of a shared link. For non-shared link, however, each time a table in the remote database was referred, a new network connection, dedicated for this particular request was created.

Based on the performed tests, it may be observed that a shared database link allows for limiting the number of network connections between databases when large amounts of data are transferred between nodes of a distributed database. This is done at the cost of network capacity between these bases, which, in turn, has a direct impact on users' requests handling times.

3.4. Tests of distributed queries

3.4.1. Tests of collocated inline views and cost-based optimization

In the process of testing, nodes ORADB1 and ORADB2 of a distributed database were used. The tests involved execution of a distributed query, which, depending on the type of test, was written using different SQL language constructions, which allow for obtaining the same results. The distributed test query was run from the ORADB2 node, and referred to two remote tables located in the ORADB1 node and two local tables.

Test of explicit use of a collocated inline view.

During the test, a construction of collocated inline views, with an alias "A" was used explicitly in the distributed query:

```
SELECT
    A.KONTRAKT_ID, A.MSISDN, U.NAZWA AS USLUGA, S.OPIS AS STATUS_USLUGI, A.STATUS_OD,
```

```
    A.DATA_AKTYWACJI
FROM
    (SELECT K.KONTRAKT_ID, K.MSISDN, UK.STATUS_OD, UK.DATA_AKTYWACJI, UK.STATUS, UK.USLUGA_ID
    FROM
        KONTRAKTY@TELEKOM.DB1SH_PUBLIC_LINK K, USLUGI_KONTRAKTU@TELEKOM.DB1SH_PUBLIC_LINK UK
    WHERE
        K.KONTRAKT_ID = UK.KONTRAKT_ID AND UK.USLUGA_ID = 5) A,
    TELEKOM.SLO_USLUGI U,
    TELEKOM.SLO_STATUSY S
WHERE
    A.USLUGA_ID = U.USLUGA_ID AND A.STATUS = S.STATUS;
```

A detailed plan of query execution together with the values estimated by the Oracle query optimizer at the stage of optimal query execution plan selection based on available statistics is presented below. The query execution plan was generated using DBMS_XPLAN.DISPLAY function:

```
------------------------------------------------------------------------------------
| Id  | Operation              | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |              |  119K|  9723K|  4557   (-1)| 00:00:55 |
|*  1 |  HASH JOIN             |              |  119K|  6677K|  1191    (5)| 00:00:15 |
|   2 |   MERGE JOIN CARTESIAN|              |     2 |    56 |     6    (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL   | SLO_USLUGI   |     1 |    17 |     3    (0)| 00:00:01 |
|   4 |    BUFFER SORT         |              |     2 |    22 |     3    (0)| 00:00:01 |
|   5 |     TABLE ACCESS FULL  | SLO_STATUSY  |     2 |    22 |     3    (0)| 00:00:01 |
|   6 |   REMOTE               |              |  119K|  3397K|  1183    (5)| 00:00:15 |
------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("USLUGI_KONTRAKTU"."USLUGA_ID"="SLO_USLUGI"."USLUGA_ID" AND
              "USLUGI_KONTRAKTU"."STATUS"="SLO_STATUSY"."STATUS")
   3 - filter("SLO_USLUGI"."USLUGA_ID"=5)

Remote SQL Information (identified by operation id):
----------------------------------------------------

   6 - SELECT "A1"."USLUGA_ID","A1"."STATUS","A1"."STATUS_OD","A1"."DATA_AKTYWACJI",
       "A1"."KONTRAKT_ID","A1"."USLUGA_ID","A2"."KONTRAKT_ID","A2"."MSISDN","A2"."KO
       NTRAKT_ID" FROM "USLUGI_KONTRAKTU" "A1","KONTRAKTY" "A2" WHERE "A2"."KONTRAKT
       _ID"="A1"."KONTRAKT_ID" AND "A1"."USLUGA_ID"=5 (accessing 'TELEKOM.DB1SH_PUBL
       IC_LINK' )
```

The real time of query execution based on SQL Trace and TKPROF was 1.86 secs.

It may be observed from the above query execution plan that despite the fact that the query refers to two remote tables located in the ORADB1 base, access to this database was realized in one request, which was assigned identifier “6” in the execution plan. This request considered restrictions imposed on both remote tables in the WHERE clause of the collocated inline view, which allowed to perform it in a remote database using the indexes put on the tables, with the final result being returned to the local database. It allowed for limiting the amount of data

transferred between the nodes of the distributed database using the network to minimum. [1],[3],[4],[5]

Test of implicit use of a collocated inline view.

During the test, a construction of collocated inline views, with an alias "A" was not used explicitly in the distributed query:

```
SELECT … FROM
    KONTRAKTY@TELEKOM.DB1SH_PUBLIC_LINK K,
    USLUGI_KONTRAKTU@TELEKOM.DB1SH_PUBLIC_LINK UK, …
WHERE
    K.KONTRAKT_ID = UK.KONTRAKT_ID AND
    UK.USLUGA_ID = 5 AND …
```

Query execution plan:

```
-----------------------------------------------------------------------------------
| Id  | Operation              | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |             |  119K|  9723K|  4565  (-1)| 00:00:55 |
|*  1 |  HASH JOIN             |             |  119K|  6677K|  1199   (6)| 00:00:15 |
|   2 |   MERGE JOIN CARTESIAN|             |     2 |    56 |     6   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL   | SLO_USLUGI  |     1 |    17 |     3   (0)| 00:00:01 |
|   4 |    BUFFER SORT         |             |     2 |    22 |     3   (0)| 00:00:01 |
|   5 |     TABLE ACCESS FULL  | SLO_STATUSY |     2 |    22 |     3   (0)| 00:00:01 |
|   6 |   REMOTE               |             |  119K|  3397K|  1190   (6)| 00:00:15 |
-----------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("USLUGI_KONTRAKTU"."USLUGA_ID"="SLO_USLUGI"."USLUGA_ID" AND
              "USLUGI_KONTRAKTU"."STATUS"="SLO_STATUSY"."STATUS")
   3 - filter("SLO_USLUGI"."USLUGA_ID"=5)

Remote SQL Information (identified by operation id):
----------------------------------------------------
   6 - SELECT "A1"."STATUS_OD","A1"."DATA_AKTYWACJI","A1"."KONTRAKT_ID","A1"."USLUGA
       _ID","A1"."USLUGA_ID","A1"."STATUS","A2"."KONTRAKT_ID","A2"."MSISDN","A2"."KO
       NTRAKT_ID" FROM "USLUGI_KONTRAKTU" "A1","KONTRAKTY" "A2" WHERE "A2"."KONTRAKT
       _ID"="A1"."KONTRAKT_ID" AND "A1"."USLUGA_ID"=5 (accessing 'TELEKOM.DB1SH_PUBL
       IC_LINK' )
```

The real time of query execution based on SQL Trace and TKPROF was 1.90 secs.

The above execution plan is identical to the query execution plan, in which a collocated inline view was used, which proves that query optimizer in Oracle rewrote the distributed query to the form that used a collocated inline view to minimize the cost of query execution Query rewrite was done in a manner transparent to the user. [6]

Test without using a collocated inline view

During this test, such a construction of a distributed query was intentionally used, which allowed for instructing the Oracle query optimized not to rewrite a distributed query to the form that used a collocated inline view each time - NO_MERGE hint was used. Following the above modifications, the query has the following form:

```
SELECT /*+ NO_MERGE(K) USE_NL(U S)  */
    K.KONTRAKT_ID, …
FROM
  (SELECT * FROM KONTRAKTY@TELEKOM.DB1SH_PUBLIC_LINK) K,
  (SELECT * FROM USLUGI_KONTRAKTU@TELEKOM.DB1SH_PUBLIC_LINK) UK,
    …
WHERE
    K.KONTRAKT_ID = UK.KONTRAKT_ID AND
    UK.USLUGA_ID = 5 AND …
```

Query execution plan:

```
--------------------------------------------------------------------------------------------
| Id  | Operation              | Name             | Rows  | Bytes |TempSpc| Cost  (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                  |  119K|  9723K|       |  4792   (4)| 00:00:58 |
|*  1 |  HASH JOIN             |                  |  119K|  9723K|  8088K|  4792   (4)| 00:00:58 |
|*  2 |   HASH JOIN            |                  |  119K|  6677K|       |  1199   (6)| 00:00:15 |
|   3 |    NESTED LOOPS        |                  |    2 |    56 |       |     6   (0)| 00:00:01 |
|*  4 |     TABLE ACCESS FULL| SLO_USLUGI       |    1 |    17 |       |     3   (0)| 00:00:01 |
|   5 |     TABLE ACCESS FULL| SLO_STATUSY      |    2 |    22 |       |     3   (0)| 00:00:01 |
|   6 |    REMOTE              | USLUGI_KONTRAKTU |  119K|  3397K|       |  1190   (6)| 00:00:15 |
|   7 |   VIEW                 |                  | 1000K|    24M|       |  1363   (4)| 00:00:17 |
|   8 |    REMOTE              | KONTRAKTY        | 1000K|    24M|       |  1363   (4)| 00:00:17 |
--------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("KONTRAKTY"."KONTRAKT_ID"="USLUGI_KONTRAKTU"."KONTRAKT_ID")
   2 - access("USLUGI_KONTRAKTU"."USLUGA_ID"="SLO_USLUGI"."USLUGA_ID" AND
              "USLUGI_KONTRAKTU"."STATUS"="SLO_STATUSY"."STATUS")
   4 - filter("SLO_USLUGI"."USLUGA_ID"=5)

Remote SQL Information (identified by operation id):
----------------------------------------------------
   6 - SELECT "KONTRAKT_ID","USLUGA_ID","DATA_AKTYWACJI","STATUS","STATUS_OD"
       FROM "USLUGI_KONTRAKTU" WHERE "USLUGA_ID"=5
   8 - SELECT "KONTRAKT_ID","MSISDN" FROM "KONTRAKTY"
```

The real time of query execution based on SQL Trace and TKPROF was 4.13 secs.

In this case, a remote database ORADB1 was accessed twice – each remote table was accessed separately. While in the case of a table named USLUGI_KONTRAKTU the Oracle optimizer took care to limit fetching data from the remote base to the service specified in the WHERE clause of the distributed query, the whole table named KONTRAKTY had to be read and sent to the local base, where the data were joined with other tables. The final results were

then sent to the user. It is worth noting that the increased number of references to the remote database and redundant data sent via the network resulted in over 2-fold increase in query execution time, and higher rate of hardware utilization in both nodes, to which the query referred.

Summary

The performed tests for collocated inline views show that this particular construction allows for limiting the number of times a remote database is accessed, provided that the distributed query includes more than one reference to tables located in the same remote node. Moreover, fetching data from a remote node in one 'collocated' query allows for performing the whole query on the remote database side where, depending on the distributed query content, appropriate filters are put on data. This allows for transferring the minimum required amount of data between the nodes of a distributed database via the network. Another issue worth noting is the fact that the Oracle query optimizer, based on the lowest cost of query execution, limits the amount of data fetched from a remote table using the conditions specified in the WHERE clause, whenever possible.

3.5. Test of the DRIVING_SITE optimizer hint

Tests were based on the query below, which returns the number of active services of all payers broken into payer and service:

```
SELECT /*+ DRIVING_SITE(UK) USE_HASH(P) */
    P.PLATNIK_ID,
    UK.USLUGA_ID,
    COUNT(*) AS LICZBA
FROM
    PLATNICY@TELEKOM.DB1SH_PUBLIC_LINK P,
    KONTRAKTY@TELEKOM.DB3SH_PUBLIC_LINK K,
    USLUGI_KONTRAKTU@TELEKOM.DB1SH_PUBLIC_LINK UK
WHERE
    P.PLATNIK_ID = K.PLATNIK_ID AND
    K.KONTRAKT_ID = UK.KONTRAKT_ID AND
    K.STATUS = 'A' AND
    UK.STATUS = 'A'
GROUP BY
    P.PLATNIK_ID,
    UK.USLUGA_ID;
```

The query refers to three remote tables, two of which are located in the node named ORA_DB1 and one in the node named ORA_DB3. The query itself was run from ORA_DB2. The Oracle query optimizer built the following execution plan for the above query, based on available statistics (without using DRIVING_SITE hint):

```
-----------------------------------------------------------------------------------------------
| Id  | Operation         | Name              | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                   | 1199K|    78M|       | 29374   (2)| 00:05:53 |
|   1 |  HASH GROUP BY    |                   | 1199K|    78M|   183M| 29374   (2)| 00:05:53 |
|*  2 |   HASH JOIN       |                   | 1199K|    78M|    45M|  9657   (3)| 00:01:56 |
|   3 |    REMOTE         | USLUGI_KONTRAKTU  | 1199K|    32M|       |  1190   (6)| 00:00:15 |
|*  4 |    HASH JOIN      |                   | 1000K|    39M|  4888K|  3616   (3)| 00:00:44 |
|   5 |     REMOTE        | PLATNICY          |  200K|  2539K|       |    94   (2)| 00:00:02 |
|   6 |     REMOTE        | KONTRAKTY         | 1000K|    26M|       |  1351   (5)| 00:00:17 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("USLUGI_KONTRAKTU"."KONTRAKT_ID"="KONTRAKTY"."KONTRAKT_ID")
   4 - access("KONTRAKTY"."PLATNIK_ID"="PLATNICY"."PLATNIK_ID")

Remote SQL Information (identified by operation id):
----------------------------------------------------
   3 - SELECT "KONTRAKT_ID","USLUGA_ID","STATUS" FROM "USLUGI_KONTRAKTU" "UK"
       WHERE "STATUS"='A' (accessing 'TELEKOM.DB1SH_PUBLIC_LINK' )
   5 - SELECT /*+ USE_HASH ("P") */ "PLATNIK_ID" FROM "PLATNICY" "P" (accessing
       'TELEKOM.DB1SH_PUBLIC_LINK' )
   6 - SELECT "KONTRAKT_ID","PLATNIK_ID","STATUS" FROM "KONTRAKTY" "K" WHERE "STATUS"='A'
       (accessing 'TELEKOM.DB3SH_PUBLIC_LINK' )
```

Analysing the above query plan, it may be noticed that the first step is sending the whole table PLATNICY from a remote node ORA_DB1 and all active contracts from table named KONTRAKTY (ORA_DB3) to the local database (ORA_DB2), followed by joining the two tables using hash join. The next step involves joining the result of the hash join with the remote table named USLUGI_KONTRAKTU (active services only, ORA_DB1), followed by grouping.

To sum up, the above query execution plan assumes that the distributed query is executed in the local base, where 3 remote tables located at two different nodes of a distributed database are sent using database links.

After using the DRIVING_SITE hint that instructs the optimizer to execute the query in the node storing a table named USLUGI_KONTRAKTU, the query execution plan is as follows:

```
------------------------------------------------------------------------------------------------------
| Id  | Operation              | Name              | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT REMOTE |                   | 1199K|   44M|       | 20180   (3)| 00:04:03 |
|   1 |  HASH GROUP BY          |                   | 1199K|   44M|   119M| 20180   (3)| 00:04:03 |
|*  2 |   HASH JOIN             |                   | 1199K|   44M|    21M|  7957   (4)| 00:01:36 |
|*  3 |    TABLE ACCESS FULL    | USLUGI_KONTRAKTU  | 1199K| 8200K|       |  1190   (6)| 00:00:15 |
|*  4 |    HASH JOIN            |                   | 1000K|   30M| 3128K|  3535   (3)| 00:00:43 |
|   5 |     INDEX FAST FULL SCAN| PLATNICY_IDX1     |  200K|  781K|       |    98   (6)| 00:00:02 |
|   6 |     REMOTE              | KONTRAKTY         | 1000K|   26M|       |  1351   (5)| 00:00:17 |
------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("A3"."KONTRAKT_ID"="A2"."KONTRAKT_ID")
   3 - filter("A3"."STATUS"='A')
   4 - access("A2"."PLATNIK_ID"="A1"."PLATNIK_ID")
Remote SQL Information (identified by operation id):
----------------------------------------------------
   6 - SELECT "KONTRAKT_ID","PLATNIK_ID","STATUS" FROM "KONTRAKTY"@TELEKOM.DB3SH_PUBLIC_LINK "A2"
       WHERE "STATUS"='A' (accessing 'TELEKOM.DB3SH_PUBLIC_LINK' )
```

The first significant change in the query execution plan is execution of the query in the remote node ORA_DB1, where 2 of 3 remote tables mentioned in the query are located, and not in the local database.

First, a local table named PLATNICY and all active contracts from a remote table named KONTRAKTY are read. Data from these two tables are joined using hash join mechanism. Next step involves joining a local table named USLUGI_KONTRAKTU to the two previously joined tables, followed by grouping. [6]

DRIVING_SITE hint allowed for sending only one table between the nodes of a distributed database, and not all 3 tables as in the initial version of the execution plan. Query execution times, which took approx. 200 and 90 seconds for a query with and without the optimizer hint, respectively, indicate that the query execution plan was optimized.

3.6. Tests of materialized views

3.6.1. Tests of base tables partitioning

Tests were performed based on the following tables:

- USLUGI_KONTRAKU – a table located in ORA_DB1 node, non-partitioned, it has an index on columns KONTRAKT_ID and USLUGA_ID;
- USLUGI_KONTRAKTU_PART – the table is a copy of a table named USLUGI_KONTRAKTU also located in the ORA_DB1 node, the table was list-partitioned based on a column named USLUGA_ID – a separate partition was created for each service.

Based on the above tables, two materialized views were created, located in the ORA_DB3 node, which contain selected contract services related to data transfer:

- USLUGA_ID = 4, NAZWA = CLOUD
- USLUGA_ID = 5, NAZWA = PAKIET DANYCH 1GB
- USLUGA_ID = 7, NAZWA = PAKIET DANYCH 500MB
- USLUGA_ID = 8, NAZWA = PAKIET DANYCH 300MB

A materialized view based on a non-partitioned table named V_USLUGI_KONTRAKTU_DATA_IDX was created with the following instruction:

```
CREATE  MATERIALIZED VIEW TELEKOM.V_USLUGI_KONTRAKTU_DATA_IDX
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
AS SELECT * FROM USLUGI_KONTRAKTU@TELEKOM.DB1_PUBLIC_LINK UKP
   WHERE UKP.USLUGA_ID IN (4, 5, 7, 8);
```

The execution plan of the query that performs a complete refresh of the above materialized view is as follows:

```
-------------------------------------------------------------------------------------------
| Id  | Operation               | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT REMOTE|                   |   479K|  9840K|  1212   (7)| 00:00:15 |
|*  1 |  TABLE ACCESS FULL      | USLUGI_KONTRAKTU |   479K|  9840K|  1212   (7)| 00:00:15 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("A1"."USLUGA_ID"=4 OR "A1"."USLUGA_ID"=5 OR "A1"."USLUGA_ID"=7 OR
              "A1"."USLUGA_ID"=8)
```

From the execution plan, we may obtain the information that full table scan was performed to filter out records with the appropriate service identifier on the side of the base table database. The index on a column named USLUGA_ID was not used as the materialized view selects approx. 40% of data in the base table. Average time of a complete refresh was approx. 44 seconds.

A materialized view based on a partitioned table named V_USLUGI_KONTRAKTU_DATA_P was created with the following instruction:

```
CREATE  MATERIALIZED VIEW TELEKOM.V_USLUGI_KONTRAKTU_DATA_P
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
AS SELECT * FROM USLUGI_KONTRAKTU_PART@TELEKOM.DB1_PUBLIC_LINK
   UKP WHERE UKP.USLUGA_ID IN (4, 5, 7, 8);
```

The execution plan of the query that performs a complete refresh of the above materialized view is as follows:

```
-------------------------------------------------------------------------------------
| Id  | Operation               | Name          | Rows  | Bytes | Cost  (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT REMOTE|               |   479K|  9840K|   498   (8)| 00:00:06 |
|   1 |  PARTITION LIST INLIST |               |   479K|  9840K|   498   (8)| 00:00:06 |
|   2 |   TABLE ACCESS FULL    | USLUGI_KONT |   479K|  9840K|   498   (8)| 00:00:06 |
|     |                        | RAKTU_PART  |       |       |            |          |
-------------------------------------------------------------------------------------
```

In this case, data from the base table are fetched only from partitions selected on the basis of the WHERE clause of the query that defines the materialized view. The existing partitions fully satisfy the condition specified in the WHERE clause. Therefore, additional filtering of data is not required. Average time of a complete refresh was approx. 30 seconds.

Summary

If the conditions specified in the WHERE clause of the query that defines the materialized view and the base table partitioning key correspond, data are fetched by means of a complete scan of selected partitions in the base table during a complete refresh of a materialized view. Such solution reduces to minimum the number of I/O operations in the node of the base table. Any other solution that uses an index on the base table or full scan of the base table results in a larger number of I/O operations, and consequently longer query execution time.

3.6.2. Test of using indexes on the base table

Tests of using indexes on the base table of the materialized view were performed based on the table named USLUGI_KONTRAKTU (ORA_DB1), which served as a basis for creating a materialized view V_USLUGI_KONTRAKTU_201502 located in the ORA_DB3 node. The view created contains a fragment of data from the base table, i.e. all contract services activated in February 2015:

```
CREATE  MATERIALIZED VIEW TELEKOM.V_USLUGI_KONTRAKTU_201502
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
AS SELECT * FROM USLUGI_KONTRAKTU@TELEKOM.DB1SH_PUBLIC_LINK UK
   WHERE  TRUNC(UK.DATA_AKTYWACJI, 'MM') =  TO_DATE('201502', 'YYYYMM');
```

For the purpose of the tests, an index was put on the DATA_AKTYWACJI column of the materialized view base table that fully complies with the condition in the WHERE clause of the query that defines the materialized view:

```
CREATE INDEX USLUGI_KONTRAKTU_IDX3 ON TELEKOM.USLUGI_KONTRAKTU(TRUNC(DATA_AKTYWACJI, 'MM'));
```

For a complete refresh of the materialized view, the Oracle query optimizer selected by default the following query plan based on the existing index:

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT REMOTE      |                       | 11996 |  246K|   638   (1)| 00:00:08 |
|   1 |  TABLE ACCESS BY INDEX ROWID| USLUGI_KONTRAKTU      | 11996 |  246K|   638   (1)| 00:00:08 |
|*  2 |   INDEX RANGE SCAN           | USLUGI_KONTRAKTU_IDX3 |  4798 |      |   128   (0)| 00:00:02 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(TRUNC(INTERNAL_FUNCTION("DATA_AKTYWACJI"),'fmmm')=
       TO_DATE('2014-02-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

Average time of a complete refresh of the materialized view using an index on the base table was 3.6 seconds.

In order to compare refresh times of the materialized view with or without an index, the NO_INDEX hint, which forces not using the existing index, was added to the query that defines the materialized view V_USLUGI_KONTRAKTU_201502:

```
---------------------------------------------------------------------------------------
| Id  | Operation               | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT REMOTE|                  | 11996 |   246K|  1340  (16)| 00:00:17 |
|*  1 |  TABLE ACCESS FULL      | USLUGI_KONTRAKTU | 11996 |   246K|  1340  (16)| 00:00:17 |
---------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(TRUNC(INTERNAL_FUNCTION("A1"."DATA_AKTYWACJI"),'fmmm')=
       TO_DATE('2014-02-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

The average time for a complete refresh was 5.1 seconds, which is approx. 40% longer than for the option with an index.

Summary

If indexes on base tables comply with the WHERE clause of the query that defines the materialized views, refreshes may take much shorter than refreshes for base tables with no indexes or with indexes that may not be used. Shorter refresh time for the option that used an index resulted from the fact that the materialized view being tested selected approx. 4% of data from the base table. Therefore, additional time required to use additional structures, namely, the index, was shorter than full table scan with redundant data amounting to 96%.

3.6.3. Tests of simple and complex materialized views.

Base tables for the materialized views being tested, i.e. tables named KONTRAKTY and USLUGI_KONTRAKTU were located in the ORA_DB1 node, and test materialized views were created in the ORA_DB3 node.

Two views V_KONTRAKTY and V_USLUGI_KONTRAKTU were created as simple materialized views, which views are mirror copies of tables KONTRAKTY and USLUGI_KONTRAKTU, respectively. Putting a materialized view log on base tables allowed for incremental (fast) refreshes:

```
CREATE MATERIALIZED VIEW TELEKOM.V_KONTRAKTY
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
AS SELECT * FROM KONTRAKTY@TELEKOM.DB1_PUBLIC_LINK;

CREATE MATERIALIZED VIEW TELEKOM.V_USLUGI_KONTRAKTU
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
AS SELECT * FROM USLUGI_KONTRAKTU@TELEKOM.DB1_PUBLIC_LINK;
```

V_KONTRAKTY_USLUGI_KONTRAKTU, whose base tables are tables named KONTRAKTY and USLUGI_KONTRAKTU was created as a complex materialized view. For the purpose of the tests, the assumption was made that due to the complexity, this view may only be refreshed in the complete mode:

```
CREATE MATERIALIZED VIEW TELEKOM.V_KONTRAKTY_USLUGI_KONTRAKTU
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
AS SELECT
K.*,
UK.USLUGA_ID,
UK.DATA_AKTYWACJI AS DATA_AKTYWACJI_USLUGI,
UK.STATUS AS STATUS_USLUGI,
UK.STATUS_OD AS STATUS_OD_USLUGI
FROM
KONTRAKTY@TELEKOM.DB1_PUBLIC_LINK K,
USLUGI_KONTRAKTU@TELEKOM.DB1_PUBLIC_LINK UK
WHERE
UK.KONTRAKT_ID = K.KONTRAKT_ID;
```

Note that in order to obtain data in the same form as in a complex materialized view with the help of the simple materialized views, it is required to join the two simple views with the equi-join with the column named KONTRAKT_ID as the join condition, on the side of the remote database (ORA_DB3). The following test sequences were performed to compare the execution times for the options that use two simple and one complex materialized view:

Sequence A (simple materialized views):

1. Modification of 10% of data in base tables of the following views: V_KONTRAKTY and V_USLUGI_KONTRAKTU.

2. Refreshes of views V_KONTRAKTY and V_USLUGI_KONTRAKTU in the incremental mode – refreshes of the two views were performed in parallel. Therefore, the refresh time for this step equals the refresh time of the view whose refresh process took longer.

3. Execution of the query that fetches the final result.

Sequence B (simple materialized views)

The only difference between this sequence and sequence A is in the number of data modified in step 1, i.e. in sequence B 50% of data in the base tables of V_KONTRAKTY and V_USLUGI_KONTRAKTU views were modified.

Sequence C (complex materialized view)

1. Modification of data was not required as the view is refreshed in the complete refresh model

2. Refresh of the V_KONTRAKTY_USLUGI_KONTRAKTU view in the complete refresh model.

3. Execution of the query that fetches the final result.

Each of the above test sequences was repeated thrice, and the execution times of each step are an average of the three measurements.

Times of test sequences – tests of simple and complex materialized views:

Time	Sequence A	Sequence B	Sequence C
Step 1	n/a	n/a	n/a
Step 2	00:25	02:35	03:20
Step 3	06:20	06:20	05:45
Sum	**06:45**	**08:55**	**09:05**

Analysis of results and summary

In the case when a small amount of data is modified in base tables (for table A the amount was 10%) between successive refreshes, simple materialized views refreshed incrementally are more efficient – the refresh time is 8 times shorter. However, we need to consider the time required for joining these views in the remote database to obtain data in the same form as in the complex view. The difference was approx. 35 seconds.

In sequence B, as much as 50% of data in base tables were modified (instead of 10%). Nevertheless, the incremental refresh time is still shorter. When we add the time required for joining simple materialized views on the side of the remote database to the refresh time, the total time of sequence B is only 10 seconds shorter than the time of sequence C.

To sum up, if frequent refreshes of data in materialized views are required, and less than 50% of data is modified between subsequent refreshes, using a few simple materialized views refreshed incrementally is a better solution. This solution allows for limiting redundant amount of data sent via the network. The only disadvantage of this solution is lower performance of queries in the remote node of the database resulting from the necessity to join data from simple materialized views prior to their retrieval.

3.6.4. Tests of size of refresh group

Tests were performed using ORA_DB1 and ORA_DB3. The ORA_DB1 node was a master node and the ORA_DB3 node was a snapshot node. In order to enable snapshot replication between the nodes, a replication group named TELEKOM_REP_GRP was created in the master node:

```
DBMS_REPCAT.CREATE_MASTER_REPGROUP ( GNAME => 'TELEKOM_REP_GRP');
```

In the snapshot node, a replication group corresponding to the replication group in the master node was also created:

```
DBMS_REPCAT.CREATE_MVIEW_REPGROUP (
   GNAME => 'TELEKOM_REP_GRP',  MASTER => 'ORA_DB1', PROPAGATION_MODE => 'ASYNCHRONOUS');
```

When the replication groups were created, it was possible to add refresh groups. Below is presented the code for adding the refresh group named TELEKOM_REFG1 in the snapshot node:

```
DBMS_REFRESH.MAKE (
   NAME => 'TELEKOM_REFG1',
   LIST => '',
   NEXT_DATE => SYSDATE,
   INTERVAL => 'SYSDATE + 120',
   IMPLICIT_DESTROY => FALSE,
```

```
      ROLLBACK_SEG => '',
      PUSH_DEFERRED_RPC => TRUE,
      REFRESH_AFTER_ERRORS => FALSE);
```

9 materialized views – one materialized view for each base table in the master node (ORA_DB1) – were created in the snapshot node (ORA_DB3) for the purpose of the tests. An assumption was made that the materialized views are complete copies of their respective base tables. Below is the code for creating a materialized view for the KONTRAKTY table – other views were created analogically:

```
CREATE MATERIALIZED VIEW TELEKOM.V_KONTRAKTY
REFRESH FAST WITH PRIMARY KEY
AS SELECT * FROM TELEKOM.KONTRAKTY@ORA_DB1;
```

Tests of „large” refresh group

The previously created refresh group named TELEKOM_REFG1, to which all test views were added, was used for “large” refresh group tests.

```
BEGIN
   DBMS_REFRESH.ADD (
      NAME => 'TELEKOM_REFG1',
      LIST => 'TELEKOM.V_KONTRAKTY, TELEKOM.V_USLUGI_KONTRAKTU,
               TELEKOM.V_PLATNICY, TELEKOM.V_ADRESY,
               TELEKOM.V_USLUGI_PLATNIKA, TELEKOM.V_SLO_STATUSY,
               TELEKOM.V_SLO_USLUGI,
               TELEKOM.V_SLO_PLANY_TARYFOWE,
               TELEKOM.V_CENY_USLUGI_W_PLANIE',
      LAX => TRUE);
END;
```

The next step was modification of data in base tables. 50% of data were modified in the tables that did not contain dictionary data, i.e. KONTRAKTY, USLUGI_KONTRAKTU, PLATNICY, USLUGI_PLATNIKA, ADRESY and CENY_USLUGI_W_PLANIE, while data in the dictionary tables: SLO_STATUSY, SLO_PLANY_TARYFOWE and SLO_USLUGI did not require modification. The refresh process was invoked with the following instruction:

```
EXECUTE DBMS_REFRESH.REFRESH ('TELEKOM_REFG1');
```

The test was repeated twice. Detailed results of tests are presented below:

Refresh group	Materialized views in the refresh group	Refresh time in sequential mode [mins:secs]	Refresh time in parallel mode [mins:secs]
TELEKOM_REFG1	V_KONTRAKTY, V_USLUGI_KONTRAKTU, V_PLATNICY, V_USLUGI_PLATNIKA, V_ADRESY, V_CENY_USLUGI_W_PLANIE, V_SLO_USLUGI, V_SLO_PLANY_TARYFOWE, V_SLO_STATUSY	03:52	
TELEKOM_REFG2	V_KONTRAKTY	01:25	03:16
TELEKOM_REFG3	V_USLUGI_KONTRAKTU	01:46	03:50
TELEKOM_REFG4	V_PLATNICY	00:16	00:51
TELEKOM_REFG5	V_USLUGI_PLATNIKA	00:13	00:58
TELEKOM_REFG6	V_ADRESY	00:17	01:02
TELEKOM_REFG7	V_CENY_USLUGI_W_PLANIE	00:03	00:05
TELEKOM_REFG8	V_SLO_USLUGI, V_SLO_PLANY_TARYFOWE, V_SLO_STATUSY	<00:01	<00:01

Tests of “small” refresh groups

A separate refresh group was created for each table not storing dictionary data, while all dictionary tables were added to one refresh group.

The scope of modified data in the base tables and the number a given test was repeated was identical as for the test of “large” refresh groups.

Detailed refresh times:

Type of test	Refresh time of a set of materialized views [mins:secs]
Test of “large” refresh group	03:51
Test of “small” refresh groups - sequential	04:01
Test of “small” refresh groups – in parallel	03:50

The results of tests clearly show that the refresh time of a set of materialized view in a “large” refresh group is approximately equal the refresh time of the same set of views in “small” refresh groups in the parallel refresh mode. However, it is worth noting that all materialized views refreshed in the same refresh group are from the same moment in time and contain consistent data – in the case when a network connection fails during a refresh of a “large”

refresh group all changes are rolled back. In the same situation, views refreshed in "small" refresh groups would contain data from different moments in time.

Another question worth noting is the time required for refresh of materialized views that are tables of dictionary tables, whose data were not modified in the base tables – the time was under 1 second. Therefore, it may be concluded that refresh of a materialized view that does not require changes of data in the base table is performed in a flash, i.e. the time of refresh is so short that it may be neglected.

To sum up, a mechanism for refreshes of "large" groups in Oracle is as efficient as parallel refreshes of the same set of materialized views in dedicated refresh groups. Moreover, Oracle ensures data consistency as data in materialized views refreshed within one refresh group are from one moment in time. The fact that the refresh time of materialized views, for which no changes in base tables are required is negligible confirms full optimization of the refresh mechanism of materialized views in Oracle.

4. Summary of tests results

The tests performed applied to three areas of a distributed database:

1. mechanisms of communication between nodes of a distributed database and handling related requests;
2. snapshot replication mechanisms;
3. mechanisms of accessing data located in individual nodes of a distributed database.

Within the first area, configuration of service processes in a node of a distributed database and different types of database links were tested. The use of shared service processes resulted in 50-fold decrease in PGA memory usage and 5% decrease in CPU usage. However, replacing a dedicated server process with only one shared process, handling in turn requests of all users, resulted in approx. 2-fold increase in times in comparison with dedicated processes, which was a direct consequence of queuing incoming requests. Therefore, depending on the expected responsiveness of the database, increasing the number of shared server processes was recommended, which also resulted in significant reduction of database hardware resources use while maintaining required request handling times.

Based on the performed tests of database links, it may be observed that a shared database link allows for limiting the number of network connections between databases when large amounts of data are transferred between nodes of a distributed database. However, it was done

at the cost of network capacity between these bases. Limited network capacity between remote nodes resulted in longer query execution times.

Comparative tests of simple and complex materialized views and refresh times of "large" and "small" refresh groups **to test mechanisms of snapshot replication**. Moreover, structures of base tables, which support refreshes of materialized views based on them, were tested.

The outcomes of the tests performed provided information that for obtaining the most current data in the materialized views, with small modifications between refreshes it is better to use a few simple materialized views refreshed incrementally. Such a solution allowed for limiting the redundant data sent via the network. The only disadvantage of this solution is lower efficiency of requests in a remote node of the database.

Based on the results of tests for refresh times of "large" and "small" groups of views, we may conclude that the mechanism for "large" group refreshes in Oracle is as efficient as parallel refreshes of the same set of materialized views in dedicated refresh groups.

Based on the tests of partitioning of materialized views base tables it was determined that when the conditions in the WHERE clause comply with the partitioning key, during a complete refresh of a materialized view, a full scan of selected partitions of the base table was required to fetch the data. Such a solution reduced to minimum the number of I/O operations in the node of the base table. Also, in the case of a complete refresh of a view, whose base tables had indexes compliant with the conditions in the WHERE clause, the time gain was 40%. In the case of partitioning, the amount of data (expressed as percentage) selected from the base table was unimportant while for using indexes, fetching less than 15% of data from the base table is cost-effective.

Mechanisms of access to data distributed in individual nodes of a distributed database

The tests confirmed that the use of collocated inline views limits the number of times a remote database is accessed, provided that the distributed query contains more than one reference to tables located in the same remote node. Moreover, the whole query is executed on the side of the remote database as the data are fetched from the remote node with one "joined" query. This allows for the transfer of minimum amount of data between the nodes. Another issue worth paying attention to is the fact that the Oracle query optimizer chooses the lowest cost of query execution, and limits the amount of data fetched from a remote table whenever possible by means of the conditions specified in the WHERE clause of the distributed query.

Another method to optimize distributed queries was the use of optimizer hints. The DRIVING_SITE hint, which forces the execution of the whole query in the specified remote

node, was used in the tests. This allowed for sending a minimum amount of data between the nodes of a remote database, and shortening query execution time by more than half, which proves that the plan of query execution was optimized.

Finally, it should be noted that the presented results apply to the particular characteristic and amount of data in individual objects of the distributed database and for the assumptions adopted for each test. Therefore, in the case of change of any factors, it is required to repeat the tests.

References

1. Antognini Ch.: Troubleshooting Oracle Performance, Apress, 740 pages, 2008.
2. Barczak A., Zacharczuk D., Pluta D.: Tools and methods of databases optimization in Oracle Database 10g. Part 1 – tuning instance, Publishing House of University of Natural Science, Studia Informatica, volume 1-2(16), pp. 5 -18, 2012.
3. Barczak A., Zacharczuk D., Pluta D.: Tools and methods for optimization of databases in Oracle 10g. Part 2 – Tuning of hardware, applications and SQL queries, Publishing House of University of Natural Science, Studia Informatica, volume 1-2(18), pp. 5 - 21, 2014.
4. Barczak A., Zacharczuk D., Pluta D.: Tools and methods for optimization of databases in Oracle 10g. Part 3 – theory in practice, Publishing House of University of Natural Science, Studia Informatica, volume 1-2(18), pp. 23 - 36, 2014.
5. Barczak A., Zacharczuk D., Korzeniecka A.: The influence of indexing methods on effective functioning of the database, Publishing House of University of Natural Science, Studia Informatica, volume 1-2(17), pp. 5 - 18, 2013.
6. Kyte T.: Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, Apress, 832 pages, 2010.
7. Oracle: Technical Documentation.
8. Powell G.: Oracle Performance Tuning for 10gR2, Elsevier Digital Press, 960 pages, 2007.
9. Tow D.: SQL. Optymalizacja, Helion, 384 pages, 2004.
10. Whalen E., Schroeter M.: Oracle Optymalizacja wydajności, Helion, 408 pages, 2003.