Parallel Query Processing

Kiruthika Sivaraman
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
kiruthika.sivaraman@sjsu.edu

ABSTRACT
The rapid growth of technology and increasing volume of data being captured by enterprises has led to the development of many large database applications in the market. In order to improve the performance of such applications, parallel query processing features have been integrated with pretty much most of the databases available for use today. This paper discusses some of the parallel query processing techniques, their basic implementation methods, and their impact on performance in some of the most commonly used databases in the IT industry.

1. INTRODUCTION
The rate at which digital data has been rapidly growing is staggering. The ability to organize and consume end-user data efficiently, in turn, decides the growth rate of companies literally. In order to do that efficiently, the performance of queries running in databases is of at most concern to companies. It is a huge challenge for database enterprises to offer databases that scale up performance. Different databases deal with this problem in different ways. Parallel query processing is one of the solutions available in various forms in most of the databases. Oracle provides this feature in the form of HINTS. MongoDB provides a horizontal scaling technique to improve performance. In this paper, we will look into some of these parallel query processing techniques in detail, underlying concepts, and some test results, which shows how much impact parallel query processing can have on performance in the context of MongoDB and Oracle databases.

2. MONGODB

2.1 Introduction
A single huge, powerful database server may not be able to provide optimal performance for queries trying to retrieve significant volume of data. Database servers with such high horse power are usually very expensive. This problem is overcome in MongoDB by a technique called Sharding. Sharding or horizontal scaling is a technique in which data is distributed across multiple relatively cheap and less powerful servers or shards. Each shard acts as a single database. All the shards collectively represent one logical database. This technique makes parallel query processing feasible in MongoDB.

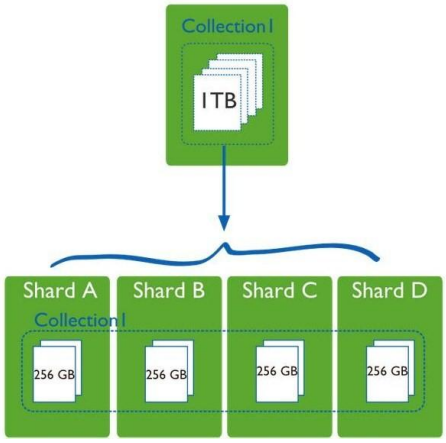

Figure 1: Data distributed across 4 shards. Image from [4]

2.2 Sharding Components
A sharded-cluster of MongoDB has three main components – query routers, shards and config servers. Shards are the ones which actually store data. Query routers route queries to the appropriate shards. Shards process queries and return result sets to query routers. Config server stores metadata about the data stored in the shards. The query router will use this information to figure out which shard contains what data. A single sharded-cluster may have more than one query router to handle user requests efficiently.

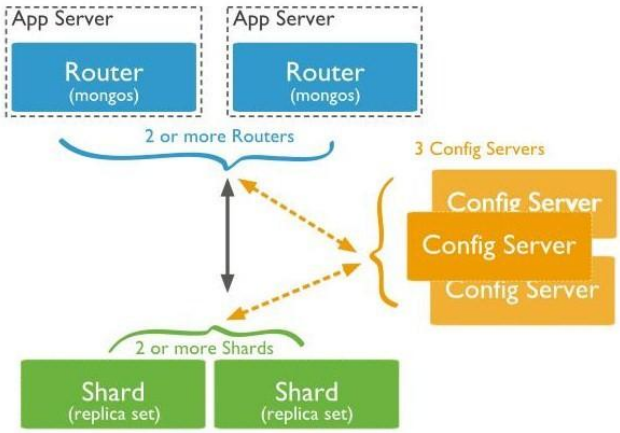

Figure 2: Sample sharded cluster. Image from [4]

2.3 Sharding in MongoDB

In MongoDB, a collection, equivalent to the concept table in RDBMS database, is distributed across shards. In order to do that, a shard key is necessary. A shard key should be an indexed field that is available in all the records in the collection. According to the Shard key, the records in the collection are divided into chunks. These chunks are evenly distributed across all the shards.

2.3.1 Range based Sharding

In Range-based partitioning, the shard key is used to determine the range of the data in the collection. For example, if the shard key is an integer, then consider a range from negative infinity to infinity. Then this range is divided into smaller partitions called chunks. Each chunk has its maximum and minimum value. Then it checks the range in which each shard key falls. Based on that, the records are then distributed across these chunks.

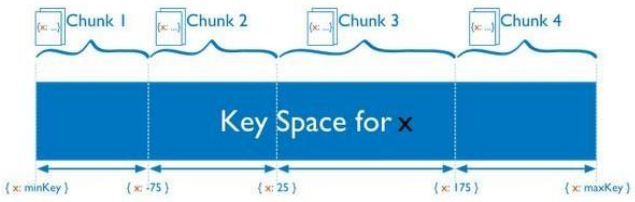

Figure 3: Range-based Sharding. Image from [4]

2.3.2 Hash based Sharding

In hash-based partitioning, hash is computed for each shard key in the collection. Then these hashes are used to create the chunks. This method is used to distribute data randomly.

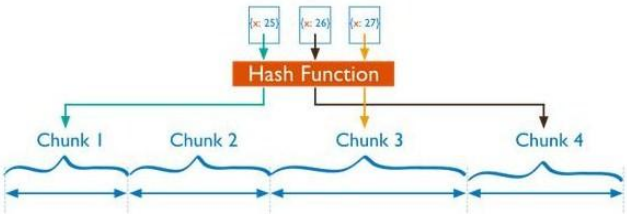

Figure 4: Hash-based Sharding. Image from [4]

2.3.3 Hash vs. Range based Sharding

Range based partitioning is more efficient for queries with range-based filters. For example, if a select query is issued to find all the records in a collection within a given range, then it is easier to find which shard the required data is in, using range-based partition strategy. Consequently the query can be directed to only those shards. Also if more queries are looking for data of the same range, a few shards could become more active than others. Hence we are not using all the resources effectively. On the other hand, in hash-based partitioning, as data is always randomly distributed across all shards, all shards are equally active but at the expense of range-based queries.

2.4 Maintaining Shards

Shards get unbalanced as we add more data to the database. Hence it is necessary to maintain shards over a period of time. This is done by splitting and balancing.

Splitting is the technique used to prevent the shards growing beyond a specified chunk size. All insert and update statements will in turn trigger a "split function" which checks the size of the affected shard and splits it if the size reaches a threshold.

Balancing is the technique used to split the data evenly across all the chunks or shards. When a chunk's size becomes bigger than other chunks, the balancer moves data from the bigger chunk into other smaller chunks till they become balanced again.

2.5 Sharding Implementation

To implement sharding, three things have to be configured – query router, config server and shard servers.

2.5.1 Starting Config Server Instances

Config server stores the metadata information about the data stored in shards. Generally it is advised to have more than one config server. The best practice is to have the same number of config servers as the number of replica sets configured. A replica set is the replication of data stored in a database. It acts as a back-up database in event of any data loss.

The first step is to create a data directory for the config server. Once the data directory is created, the config server can be started using the below command from a terminal.

Syntax:

mongod --configsvr --dbpath <path to data directory created> --port <port>

Example:

mongod –configsvr –dbpath /opt/mongodb --port 27300

The default port for a config server is 27019. If no port is specified in the above command then the server is started in the default port.

2.5.2 Starting Shard Server Instances

Shard servers are the ones that store the actual data. The data is divided into chunks and is distributed evenly across all the configured shards.

Like config servers, shard servers also needs data directory. Hence one data directory per each shard server should be created. Use below command to start a shard server.

Syntax:

mongod --shardsvr --dbpath <path to data directory created> --port <port>

Example:

mongod –shardsvr --dbpath /opt/mongodb --port 27200

No two servers can run on the same port on the same server. The best practice is to have at least four shard servers and three replica sets for each shard server. But depending upon the nature of application, more shard servers may be configured.

2.5.3 Starting Query Router Instances

The mongos instances act as query routers. They are light weight and do not need data directories. The mongos instances can even run on the same server which runs the config server.

The config server names have to be specified while starting a mongos instance. The config names should be identical and also in the same order for all the instances of mongos for a sharded cluster. The below command is used to start a mongos instance.

Syntax:

mongos --configdb <config server host names> --port <port>

Example:

mongos --configdb example1.edu:27019, example2.edu: 27019, example3.edu: 27019 --port 27100

The default port for a mongos instance is 27017. Also this port is the default port of mongo. Hence the best practice is to only configure the query routers on this port. The config servers and shard servers should never be configured on port 27017.

2.5.4 Shard Configuration

Once config servers, shard servers and query routers are up, they have to be added to shard cluster. The first step is to login to a mongos instance. The "mongo" command from the terminal will automatically login into a mongos instance as the query router is configured on the default port. After logging in, the below command is used to add the shard servers to the sharded cluster.

Syntax:

mongos>sh.addShard("<host>");

Example:

mongos>sh.addShard("example4.edu: 27200");

This command is executed for all the shard servers. To check if the shard servers are added *sh.status()* may be used. Once the shard servers are added, the sharding is enabled for a database using the below command.

Syntax:

mongos>sh.enableSharding("<database name>");

Example:

mongos>sh.enableSharding("ParallelProcessingDb");

After enabling sharding for a database, collections in that database can be sharded. To do that for a collection, a shard key must be chosen first. The shard key may be a single indexed field or compound indexed field. If the collection is not empty, then the shard key must be indexed using *ensureIndex()* command. But if the collection is empty, then the shard key will be indexed automatically when *shardCollection* command is executed.

Range-based sharding is done using the below command.

Syntax:

mongos> sh.shardCollection("<database.collection>", "<shardkey>", "<unique>");

Example 1:

mongos> sh.shardCollection("ParallelProcessingDb.PPData", {"name": 1, "address": 1});

Example 2:

mongos> sh.shardCollection("ParallelProcessingDb.PPData", {"empId": 1}, true);

In the first example, name field is used as a shard key to partition data. In case there are two records with the same name, the address field is used as shard key. In the second example, empId is used as a shard key to partition data as it is always unique.

Hash-based sharding is done using the below command.

Syntax:

mongos> sh.shardCollection("<database.collection>", "<shardkey>");

Example:

mongos> sh.shardCollection("ParallelProcessingDb.PPData", {"empId": "hashed"});

In the above example, hash of the field "empId" is used as a shard key to partition data into chunks.

2.5.5 Insert data into sharded collection

Data may be inserted, updated, removed or queried in the sharded cluster from any of the mongos instances. As huge set of data have to be inserted to test the performance of parallel query processing, a third party mongo-java driver jar has been used along with a JAVA program. The below code is used for inserting data. The variable declaration and class declarations are not provided here.

```
MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
DB db = mongoClient.getDB( "PP" );
DBCollection coll = db.getCollection("PPHashData");
mongoClient.setWriteConcern(WriteConcern.NORMAL);
BasicDBObject doc;
for(int i = 1 ; i <= 10000000 ; i++)
{
        income = (int)(Math.random() * 100000);
        doc = new BasicDBObject("empId", id).
        append("name",name). append("income", income);
        coll.insert(doc);
        id++;
}
```

Alternatively, the data may also be inserted directly from a mongos instance using the below command where the collection name is "*PPHashData"* and the db name is *"PP"*.

mongos> use PP;

mongos> db.PPHashData.insert({empId: 1, name: "Example", income: 1500});

When the data is inserted, it is automatically divided into chunks and distributed across the shards, as the collection into which the data is inserted has been already sharded. To check how the data is sharded, *sh.status()* command may be used. A sample output of sh.status() command is given below.

```
mongos> sh.status();
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "version" : 3,
        "minCompatibleVersion" : 3,
        "currentVersion" : 4,
        "clusterId" : ObjectId("5344bf9b45cbbb762af0bb60")
}
  shards:
        {  "_id" : "shard0000",  "host" : "localhost:27100" }
        {  "_id" : "shard0001",  "host" : "localhost:27101" }
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
        {  "_id" : "test",  "partitioned" : false,  "primary" : "shard0000" }
        {  "_id" : "PP",  "partitioned" : true,  "primary" : "shard0000" }
                PP.PPData
                        shard key: { "empId" : 1 }
                        chunks:
                                shard0001       12
                                shard0000       13
                        too many chunks to print, use verbose if you want to force print
                PP.PPHashData
                        shard key: { "empId" : "hashed" }
                        chunks:
                                shard0000       10
                                shard0001       12
                        too many chunks to print, use verbose if you want to force print
```

Figure 5: Screenshot of sh.status() output

The above screen shot shows that there are two shards in the configuration, shard0000 and shard0001. Sharding is enabled in the database "PP" as "partitioned" attribute is true for that. Also, "PPData" collection is partitioned using range-based sharding with a total of 25 chunks, where 12 chunks are distributed to shard0000 and 13 chunks are distributed to shard0001. Collection "PPHashData" is partitioned using hash-based sharding with a total of 22 chunks, where 10 chunks are distributed to shard0000 and 12 chunks are distributed to shard0001.

2.6 Experiment Results

For this experiment, all the shards, config servers and query routers are configured in the same machine. The test was done on a machine with 6 GB memory and Intel Quad Core i5 CPU at 1.70 GHz. The configuration status is given in Figure 5. Exactly 10 million records are inserted into a MongoDB collection. Then it is tested with queries to retrieve 10000, 100000, 1000000 and 10,000,000 records. Finally the query performances are compared against sequential processing, hash-based sharding with two shards and range-based sharding with two shards. The below table and graph provides the performance comparison of the queries.

Table 1: Query Performance Results

Technique	Time taken to retrieve records (in secs)			
	10,000	**100,000**	**1,000,000**	**10,000,000**
Sequential	5.1	6.4	5.1	5.1
Range Based (2 shards)	0.7	0.3	26	22
Hash Based (2 shards)	17	29	9.6	10.5

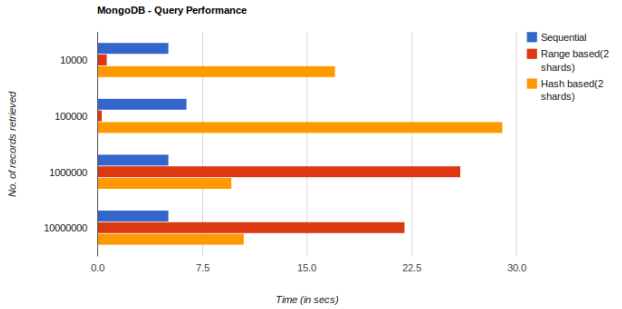

Figure 6: MongoDB Query Performance Graph

From the graph in Figure 6, it is seen that for sequential processing, the query takes around 5.5 seconds to retrieve the results. Whereas when range based partitioning is used, the query takes only around 0.5 seconds to retrieve 10,000 and 100,000 records as they have to go to only one shard to retrieve the result. Hence only half of the records in the collection have to be scanned by the query. Whereas to retrieve 1,000,000 and 10,000,000 records the queries take around 20 seconds because it has to go to both the shards to retrieve the data. With Hash-based sharding, queries take around 15-20 seconds to retrieve the results as it goes to both the shards in all the cases. Queries against range-based and hash-based shards exhibit very poor performance here because all the shards run on the same server. When they run on different servers, the performance will be better.

3. ORACLE

3.1 Introduction

Oracle supports parallel query processing by providing an option called "PARALLEL HINT." By using hint in a query, it can be specified whether that query should be executed in parallel or sequential. Also the degree of parallelism can be specified, i.e. how many number of processes or threads to be spanned to execute the query. The work to be performed is divided equally among the parallel processes.

3.2 Shared-Everything vs. Shared-Nothing Architecture

In Shared-Nothing architecture, a system has its own processing unit and memory. Hence for the query to execute in parallel the data must be partitioned into chunks using some kind of hash function. The partition strategy has to be decided upon the initial creation of the system. Then each system will be responsible for processing or querying its own data. This enforces only fixed parallelism. Most of the non-oracle based systems are based on Shared-Nothing architecture.

Oracle databases relies on Shared-Everything architecture. As data is shared across all systems, it is not necessary to think about parallelism aspects during the data modeling phase. In other words, oracle provides flexible parallelism. Also degree of parallelism can be specified without worrying about the data layout.

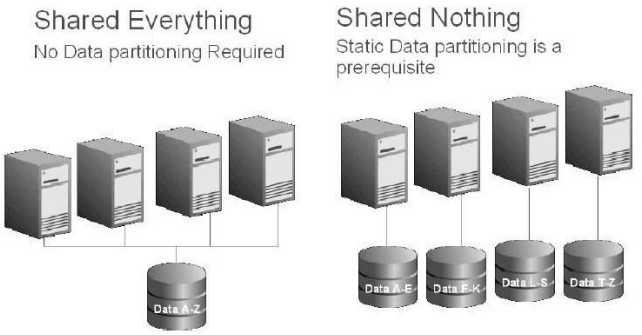

Figure 7: Shared-Everything vs. Shared-Nothing Architecture. Image from [2]

3.3 Parallel SQL Processing Components

The two main components in parallel SQL processing are query coordinator (QC) and parallel execution (PX) server processes. QC is responsible for delegating the work to PX servers and also performs some tasks which have to be executed sequentially. For instance, queries using aggregate functions like SUM should add the results returned by PX servers. QC takes responsibility of adding the results returned by PX servers and returns the final sum to the user. The actual session opened by the user will be used as QC. PX servers are selected from the pool of available servers.

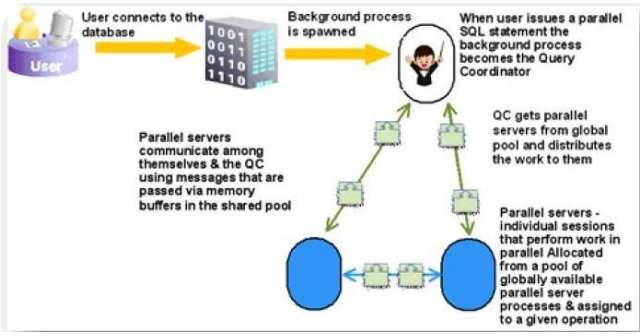

Figure 8: Oracle Parallel SQL Processing Architecture. Image from [2]

3.3.1 Block based vs. Partition based granules

A granule is the smallest piece of work in Oracle. The size of the granule is determined based on query requirements. Block-based granule is a block of data that a PX process uses to execute the query in parallel. When block-based granules are used, data re-distribution is required once all PX servers finish executing their part due to the fact that data stored in each granule of one PX process is unknown to others. Partition-based granules are the ones, in which each partition is grouped together as a granule. When partition-based granules are used, no data re-distribution is necessary as only one PX server will work on data of each partition. The operation that uses partition-based granules is partition-based join. Depending on the query, Oracle decides whether block-based or partition-based granule is to be used.

3.4 Producer-Consumer Model

Consider for example, the query have to get all the customers who have bought something by performing a join of Customers table with Sales table. Also the requirement is to execute this query in parallel with two PX servers. In this case, the first PX server will work on the first block of data from Customers table and Sales table. Then it will return its result set to QC. The second PX server does the same on the second block of each table and then provides the result set to QC. Now QC has to scan through the result sets returned by both PX servers. Then it joins the result sets. In this case, the QC will perform the same task as PX servers.

To avoid this overhead, Oracle uses Producer-Consumer Model. In this model, two sets of PX servers will work in parallel where one set will act as producer of rows and the other set will be the consumer of these rows. Therefore, if the degree of parallelism is two then it actually needs four PX servers to run the query. One set of PX servers will act as Producer, will read and send the data from Customers table. The other set of PX servers will act as consumers, will receive the data and will join it with the data from Sales table.

Mostly, the PX servers will operate in pairs. But there are some cases such as a simple query, which counts the number of records in a table, does not require pairs of PX servers. The database decides whether the query has to be executed with PX servers in pairs or not.

3.5 Data re-distribution

Most of the parallel operations like sorting, aggregation and joining will require data re-distribution. If block-based granules are used, data stored in each granule of one PX process is unknown to others. Hence, in order to perform the parallel operation, the data has to be re-distributed among the PX servers. The only exception where data re-distribution is not necessary is when partition-based granules are used. In partition-based granules, the tables to be joined are already equi-partitioned based on the join key. Hence the join column data in the first partition of one table will be the same as the data in the first partition of the other. This will ensure that there are no matching rows for the join outside the first partition.

Most commonly used data re-distribution techniques are given below.

HASH – This is the most common technique as it achieves equal distribution of work to all the PX servers.

BROADCAST – This is used when one result set of the join query is small. The smaller result set is broadcasted to all the PX servers to perform the join.

RANGE – This is used for operations like parallel sort. Each PX server will work on a range of data and sort it. Hence QC need not have to perform a final sort. Instead it just displays the results returned by PX servers in correct order.

KEY – When a partitioned table is joined with a non-partitioned table, Oracle first groups data from non-partitioned table based on the join key using KEY data redistribution.

ROUND ROBIN – This can be the final re-distribution operation before sending a result set to the requesting process.

Data re-distribution techniques are explained in the below section using a join example for each.

3.5.1 Serial Join vs. Parallel Join

Consider a situation where two large tables, Customers and Sales, have to be read and joined based on its common column "cust_id". In serial join, a single process will be responsible to scan both the tables entirely and then to perform the join. The red arrow in the below diagram represents that single process.

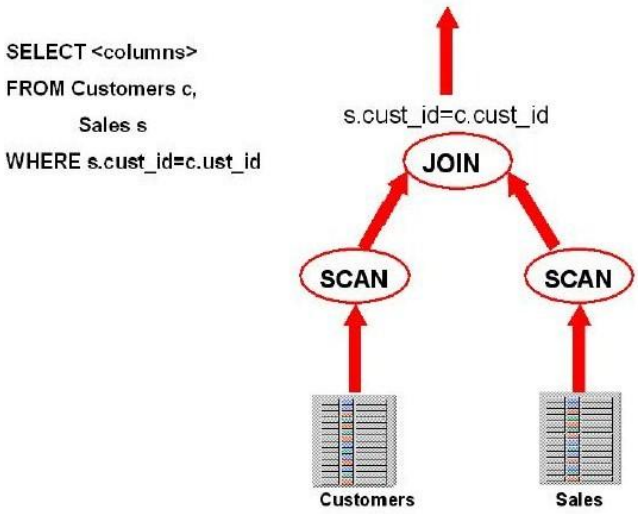

Figure 9: Serial Join. Image from [2]

Consider a parallel join with two PXs using block-based granules. The two PXs, red arrow and a green arrow, scan both the tables in parallel. Once both the scans are completed, the data read by both red and green arrows will be re-distributed in order to find the common records between the tables.

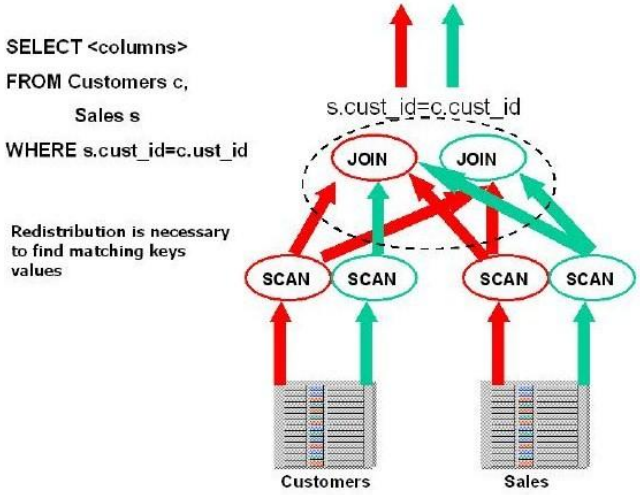

Figure 10: Parallel join using block based granules. Image from [2]

Consider a parallel join with two PXs using partition-based granules and that both tables are equi-partitioned based on the join column "cust_id". The two PXs, red arrow and green arrow work on the same partition of the Customers table and Sales table in parallel. Hence, there is no need for data-redistribution.

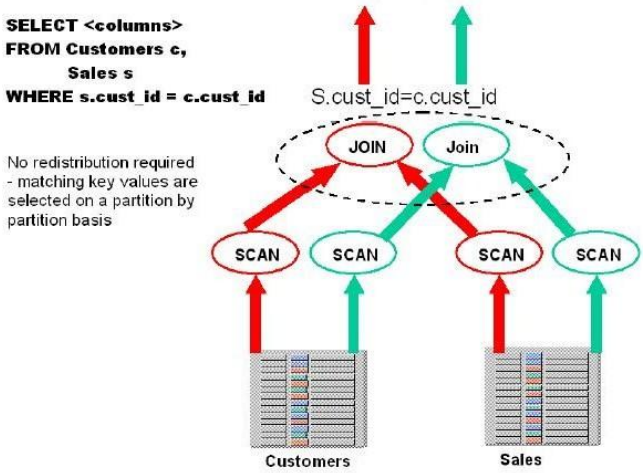

Figure 11: Parallel join using partition based granules. Image from [2]

3.6 Enabling Parallel SQL Processing

Parallel processing is enabled by default in Oracle when automatic degree of parallelism is active. For each query, Oracle decides by itself whether that query needs to be executed in parallel or sequential. In order to disable parallel processing for a table, the below query can be used.

ALTER TABLE <table_name> NOPARALLEL;

In order to enable parallelism again for that table, the below query can be used.

ALTER TABLE <table_name> PARALLEL;

If all the queries in a particular session have to be executed in parallel, then the below query can be used.

ALTER SESSION FORCE PARALLEL QUERY;

Another way to force parallelism on a table is by providing parallel hint in SQL queries as below.

SELECT /+ PARALLEL (2) */ c.cust_name FROM Customers c, SALES s WHERE c.cust_id = s.cust_id;*

The "PARALLEL (2)" is the hint used in the above query. This means that two parallel processes will be spawned. The table will be scanned in parallel to compute the result of this query. In case only "PARALLEL" is specified, the default degree of parallelism is used. The default degree of parallelism is calculated using the below formula.

2 * CPU_COUNT * ACTIVE_INSTANCES_COUNT

For example, in a two-node cluster, where each cluster has quad core CPU, the default degree of parallelism will be 16 (2 * 4 * 2).

The default degree of parallelism is designed to make use of all the resources available. But in a real-time scenario this may not work because if all the resources are used by a single query then

the other processes may be starved. Hence it is not a good idea to use default degree of parallelism in all cases. The degree of parallelism can be specified at a table or index level using the below query.

ALTER TABLE <table_name> PARALLEL 4;

This query specifies to use 4 as the degree of parallelism for that table.

3.7 Automatic Degree of Parallelism (Auto DOP)

If Auto DOP is active, then Oracle itself decides whether a query should run in parallel or not. If the total elapsed time Oracle takes to execute a query is less than the PARALLEL_MIN_TIME_THRESHOLD parameter value, then the query is executed serially. The default value for this parameter is 10 seconds.

Otherwise, the database will use parallelism. The ideal DOP is calculated using the cost of all the scan operations in a query. In order to make sure that the ideal DOP does not starve the system, the optimizer will cap the DOP. The cap is set by a parameter called PARALLEL_DEGREE_LIMIT. The formula used to calculate it is given below.

PARALLEL_DEGREE_LIMIT = PARALLEL_THREADS_PER_CPU * CPU_COUNT * ACTIVE_INSTANCE_COUNT

Once this value is calculated, the optimizer will compare the ideal DOP with PARALLEL_DEGREE_LIMIT and then it will take the lowest of the both parameter values.

ACTUAL_DOP = MIN (IDEAL DOP, PARALLEL_DEGREE_LIMIT)

This DOP value will actually be used for that query.

The below diagram explains how Auto DOP works.

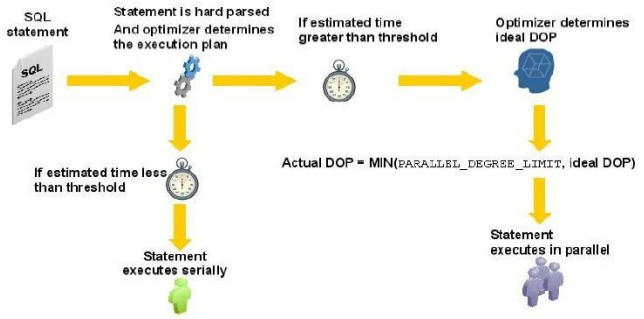

Figure 12: Working of Auto DOP. Image from [2]

The Auto DOP is controlled by the initialization parameter PARALLEL_DEGREE_POLICY. The values of this parameter can be one of the below.

MANUAL: It is the default value which means Auto DOP is not active.

LIMITED: Auto DOP is assigned only to that tables which do not have DOP specified and are invoked with only "PARALLEL" hint.

AUTO: Auto DOP is active for all the tables irrespective of whether the "PARALLEL" hint is used or not for that query.

3.8 Queuing

When the requested number of PXs is not available for a query, then that query is put in a queue till all requested resources become available. By doing so, Oracle ensures that queries will not be run in less DOP than specified or serially. The queue is maintained on First In-First Out (FIFO) basis. The query first executed will take the first position in the queue.

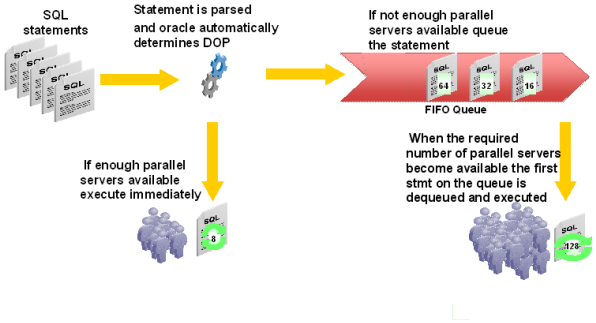

Figure 12: Working of Queuing. Image from [2]

3.9 Experiment Results

The test was done on a machine with 6 GB memory and Intel Quad Core i5 CPU at 1.70 GHz. A total of 17 million records were inserted into the test table for testing query performance.

3.9.1 Experiment 1

The first experiment conducted is to count the total number of records present in the test table. An aggregate function count(*) is used to accomplish this task. The query performance is tested by executing the query sequentially, in parallel with two and four threads. The below graph provides the performance comparison of the queries.

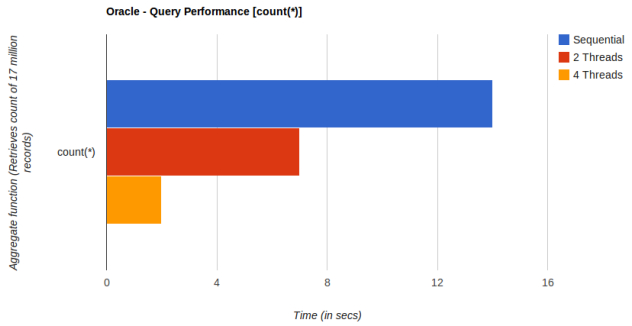

Figure 12: Oracle query performance graph for aggregate function

3.9.2 Experiment 2

The second experiment conducted is to compare the time taken to retrieve 10 million, 1 million, 100 thousand and 10 thousand records sequentially and in parallel using 2 threads and 4 threads. The below graph provides the performance comparison of the queries.

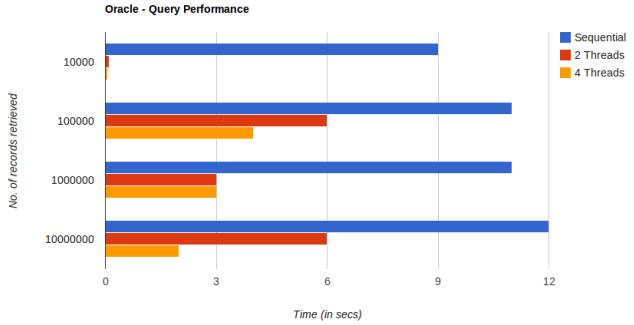

Figure 12: Oracle query performance graph for record retrieval

4. CONCLUSION

MongoDB provides a parallel query processing solution called Sharding to get maximum performance gain. From the test results, it is observed that the way the shards are designed, and the kind of Sharding being done according to the nature of the application play an important role in performance. Oracle offers parallel query processing through parallel hints. Also parallel hints are used closely along with Auto degree of parallelism feature. From the tests conducted in Oracle database, it is understood that as long as there are enough CPU cores on the machine, using more parallelism will yield better performance.

5. REFERENCES

[1] Deploy a Sharded Cluster, 2013. Retrieved April 10, 2014, from MongoDB Manual 2.6, MongoDB, Inc. : http://docs.mongodb.org/manual/tutorial/deploy-shard-cluster/

[2] Dijcks, J.P., Baer, H., and Colgan, M. Oracle Database Parallel Execution Fundamentals, 2010. Retrieved April 12, 2014, from An Oracle White Paper, Oracle: http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-parallel-execution-fundamentals-133639.pdf

[3] How Parallel Execution Works, 2011. Retrieved April 12, 2014, from Oracle Database VLDB and Partitioning Guide: http://docs.oracle.com/cd/E11882_01/server.112/e25523/parallel002.htm

[4] Sharding and MongoDB, 2014. Retrieved April 10, 2014, from MongoDB Documentation Project, MongoDB, Inc. : http://docs.mongodb.org/master/MongoDB-sharding-guide.pdf

[5] Sharding Introduction, 2013. Retrieved April 10, 2014 from MongoDB Manual 2.6, MongoDB, Inc. : http://docs.mongodb.org/manual/core/sharding-introduction/