Parallel Processing in MySQLDB Database Systems

Shahid Akhtar Khan and DR. Suryakant Yadav

Centre for Research and Studies,
Noida International University, Noida.

Abstract

The ever-growing data and its need to process the data with utmost efficiency and cost effectiveness for the companies has motivated various processing technologies which can handle both traditional relational database management system as well as unstructured and semi-structured database emerging technologies. So, there is a need of some strategy which can process the data from these large database systems with ease and cost effectiveness. In this paper we have discussed and compare between the approaches known as "Lazy Fetch" and "Parallel Fetch" and demonstrate our assumption by analysing the response time of the queries executed on MySQLDB databases under AWS Aurora cloud environment and we also discuss the open issues and challenges raised on parallel data analysis with MySQLDB.

Keywords*-AWS Cloud; Amazon Aurora; Lazy Fetch; Parallel Fetch.*

1. Introduction

Information technology provides a platform to access the data sets in all multidisciplinary domains endeavors including scientific, professional, social and so on. As we are working in the era of big data which is increasing at a very high speed on day basis; it's management and storage in an explicit way has becomes highly challenging and tedious. The evolution of highly scalable infrastructures has a major role in the successive progression of storage management systems. However, various issues have been emerged viz., availability and consistency of datasets, along with the scalability of environments and its access. The continuous data generation results in an exponential increase in data and therefore, requires suitable parallelism techniques to process the data efficiently.

Starting from our traditional parallel database management systems, we have popularly known 3 database architectures classified as shared- everything, shared-disk and shared-nothing architecture. Based on these 3 available parallel database architectures and its components like processors, memory modules and storage disk. Now, the responsibility of Hardware vendors is now to develop a model considering these components and how they are connected with each other.

Objective

To study and investigate the power of parallel processing in MySQLDB database systems, our main focus is to divide a complex transaction into multiple small transactions and test the same through parallel fetching technique. We exploit the power of parallel fetch technology in comparison with lazy fetch technology in database systems to investigate this objective.

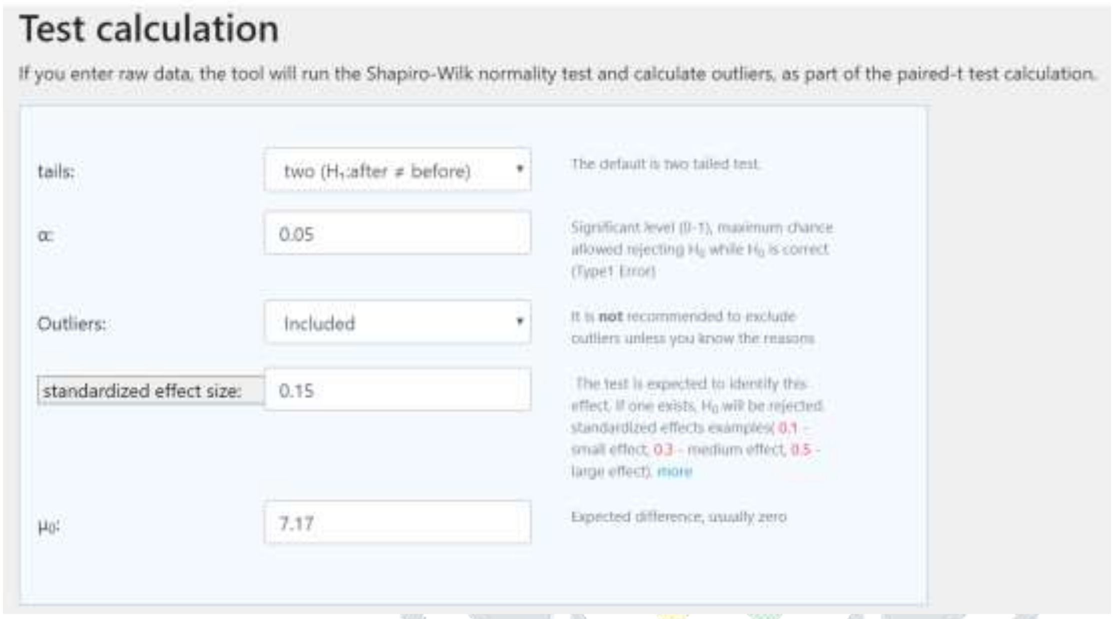
2. Research Hypothesis

We took Shapiro-Wilk normality test calculator to test our research hypothesis based on our 1st objective regarding parallel processing.

Hypothesis Formation:

- Null Hypothesis (H_0): There is no significant difference between Lazy and Parallel Fetch technology when it comes to compare on response time and throughput of the queries when checked on Amazon Arora MySQLDB.

1. Two tailed hypothesis test where after values =! Before values
2. Significant Level α: 0.05
3. Standardized effect size: 0.15
4. µ0: 7.17



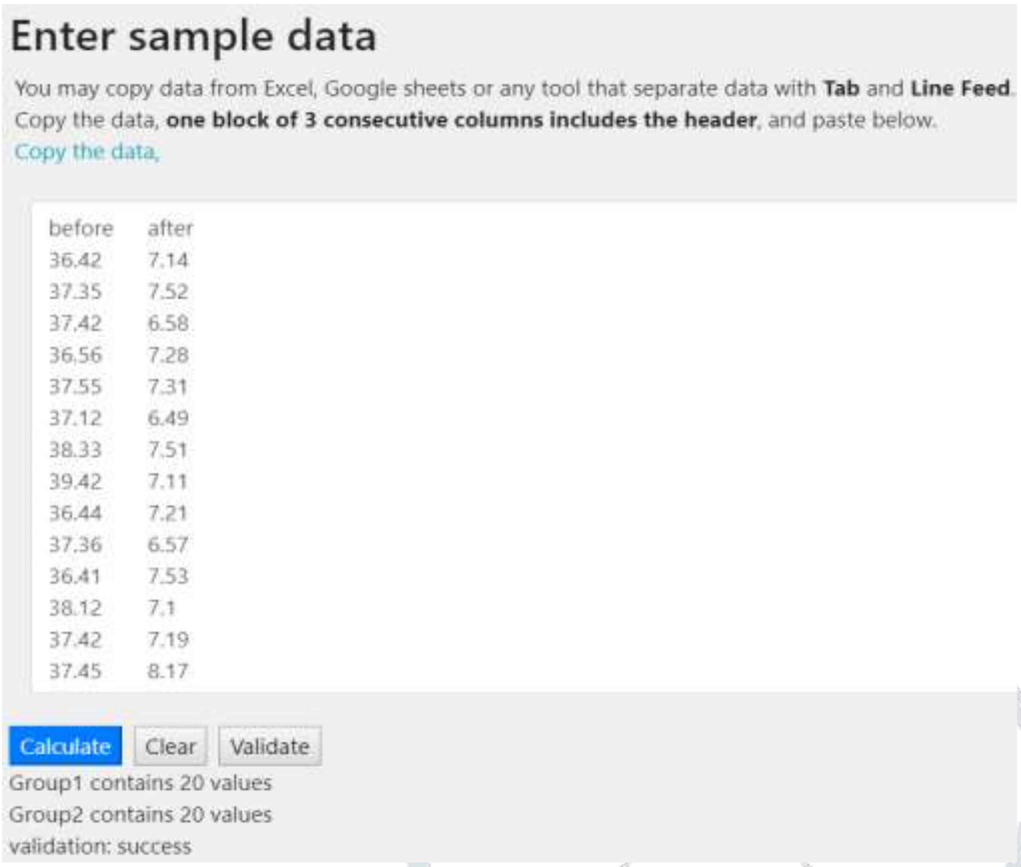
Test calculation

If you enter raw data, the tool will run the Shapiro-Wilk normality test and calculate outliers, as part of the paired-t test calculation.

tails:	two (H_1:after ≠ before)	The default is two tailed test.
α:	0.05	Significant level (0-1), maximum chance allowed rejecting H_0 while H_0 is correct (Type1 Error)
Outliers:	Included	It is **not** recommended to exclude outliers unless you know the reasons
standardized effect size:	0.15	The test is expected to identify this effect. If one exists, H_0 will be rejected. standardized effects examples(0.1 - small effect, 0.3 - medium effect, 0.5 - large effect). more
μ_0:	7.17	Expected difference, usually zero

Figure 2.1: Test calculation – Shapiro-Wilk normality test

Before values are from the year-wise single queries and after values are results of year-wise parallel queries on same datasets.



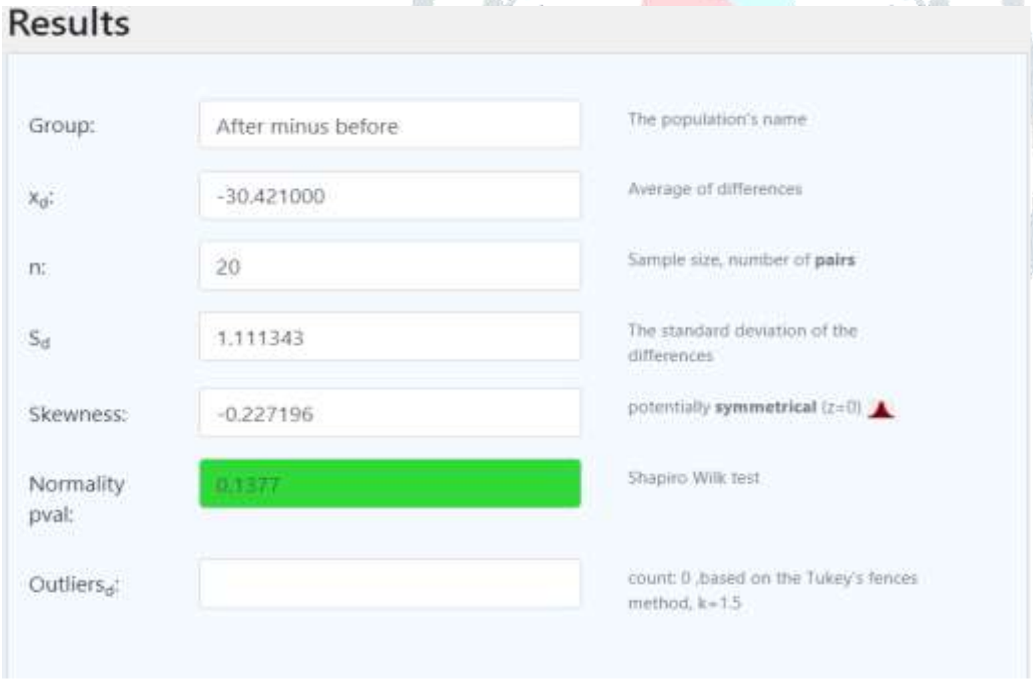
Enter sample data

You may copy data from Excel, Google sheets or any tool that separate data with **Tab** and **Line Feed**. Copy the data, **one block of 3 consecutive columns includes the header**, and paste below.
Copy the data,

before	after
36.42	7.14
37.35	7.52
37.42	6.58
36.56	7.28
37.55	7.31
37.12	6.49
38.33	7.51
39.42	7.11
36.44	7.21
37.36	6.57
36.41	7.53
38.12	7.1
37.42	7.19
37.45	8.17

Group1 contains 20 values
Group2 contains 20 values
validation: success

Figure 2.2: Data input in Shapiro-Wilk normality test calculator



Results

Group:	After minus before	The population's name
x_d:	-30.421000	Average of differences
n:	20	Sample size, number of **pairs**
S_d	1.111343	The standard deviation of the differences
Skewness:	-0.227196	potentially **symmetrical** (z=0)
Normality pval:	0.1377	Shapiro Wilk test
Outliers$_d$:		count: 0 ,based on the Tukey's fences method, k=1.5

Figure 2.3: Results – Normality test

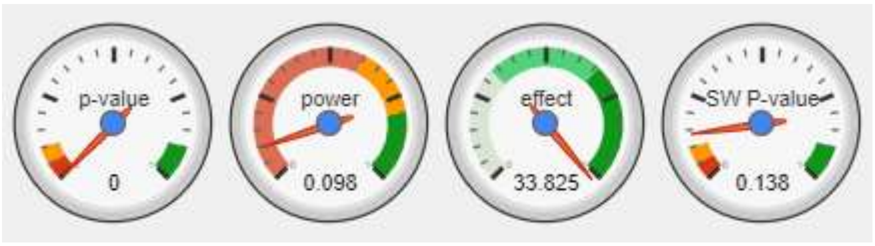


Figure 4: Results – Normality test

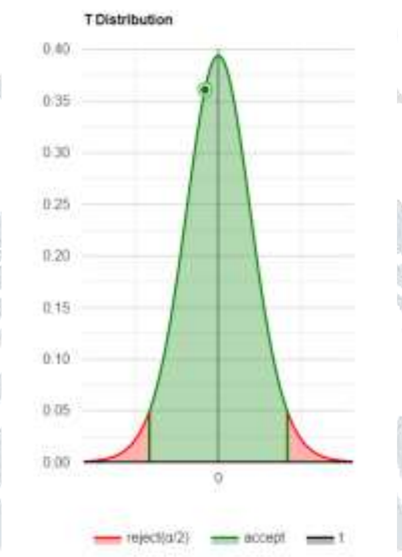
Paired sample T-test, using T distribution (DF=19) (two-tailed) (validation)

1. H_0 hypothesis: Since p-value<α, H_0 is rejected. The average of **After minus before's** population is considered to be **not equal to** the μ0. In other words, the difference between the average of the **After minus before** and μ0 is big enough to be statistically significant.

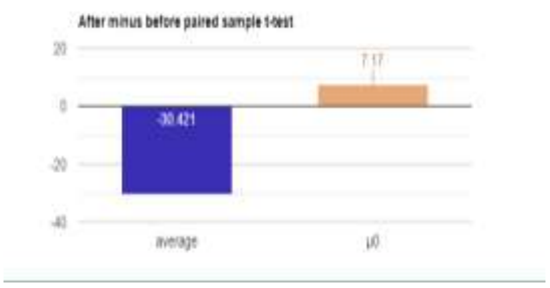
2. P-value: p-value equals **0.00000**, ($p(x \le t) = 0.00000$). This means that the chance of type1 error (rejecting a correct H_0) is small: 0.000 (0.0%). The smaller the p-value the more it supports H_1.

3. The statistics: The test statistic t equals **-151.269287**, is not in the 95% critical value accepted range: [-2.0930 : 2.0930]. x=-30.42, is not in the 95% accepted range: [6.6500 : 7.6900].

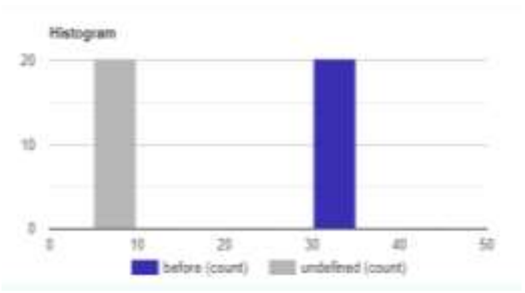
4. Effect size: The observed standardized effect size is large (33.82). That indicates that the magnitude of the difference between the average and μ0 is large.



Graph 1: T distribution



Graph 2. Delta distribution



Graph 3: Data Histogram

Test validation **The requested test was calculated, it is likely you chose the right test.**

- **Outliers:** Outliers' detection method: Tukey Fence, k=1.5 The data doesn't have outliers.

- **Normality assumption:**The assumption was checked based on the Shapiro-Wilk Test. (α=0.05) It is assumed that **After minus before** is normally distributed (p-value is 0.138), or more accurately, we can't reject the normality assumption.

Test power Although the priori power is low (0.09770), **the H_0 is rejected.**

So , the null hypothesis is rejected by using Shapiro-Wilk normality test on our paired- T test results (before values-Single query execution time and after values – parallel query execution time on same data sets.)

Reject the null hypothesis (meaning there is a definite, consequential relationship between the two test results of lazy v/s parallel fetch on same data sets).

3. Solution Validation, Analysis of Data

APPROACH

Essentially, we are working on to demonstrate between two methodologies in our distributed environment: a Parallel Fetch Technology and Lazy Fetch Technology. So as to accomplish a typical comprehension, we currently portray the two systems in the accompanying segment. Subsequently we center around their particular usage lastly, we assess the exhibition of the two systems and contrast them with one another.

A. Lazy Fetch Strategy

This technique is like notable lethargic bringing procedures, otherwise called on-request getting, sluggish stacking or conceded stacking, from object-social mappers, for example, Hibernate or others. Here, as opposed to the equal bring not all segments from the questioned tuples are stacked, yet the whole information from one parcel.

B. Parallel Fetch Strategy

In this technique, we make one string for every database segment on every customer so as to bring all information at the same time. Regarding the starting model this implies that we bring all questioned tuples from the principal segment SUBSCRIBER Partition1 and from the second parcel SUBSCRIBER Partition2 with two strings in equal. When the customer has gotten all questioned tuples, they are consolidated. We guarantee the customer side receipt of all questioned tuples with the join() technique [9] that holds up until all strings are done effectively. This technique synchronizes all strings and ensures that all parcels of a tuple are conveyed to the customer. However, this synchronization strategy bears the danger of starvation when for example a distant worker isn't reachable because of organization issues or different causes. Nonetheless, we accept that this methodology in mix with a specific hanging tight an ideal opportunity for the strings is more appropriate than getting inadequate information.

4. IMPLEMENTATION

In order to measure the performance of the *parallel* and the *lazy fetching* strategy, we implemented both strategies in a local environment, where we installed all components (2 caches, 2 database partitions and 1 client) on one physical machine. Above that, we measured the performance in a remote environment in which we connected 3 machines via 1 GB Ethernet, where the client is on one physical machine and the 2 caches and the 2 database partitions are located in different clouds. Finally, we state the hardware dimensions of every machine as follows:

So as to quantify the exhibition of the of the *parallel* and the *lazy fetch* methodology, we executed the two methodologies in a datacenter, where we installed all componentss (12 database partitions, cache and 1 thin client) on one actual machine. Over that, we estimated the presentation in a remote database environment wherein we associated 3 machines by means of 1 GB Ethernet, where the customer is on one actual machine and the 2 reserves and the 2 database allotments are situated in various clouds. At long last, we express the equipment measurements of each machine as follows:

- *24*xIntel(R) Xeon(R) CPU L5639 @ 2.13GHz
- 96 GB RAM
- 1xSSD drive =250 GB HDD
- 64 Bit CentOS OS
- 1 Gbit Internet connection
- Aurora MySQL 5.6 Software

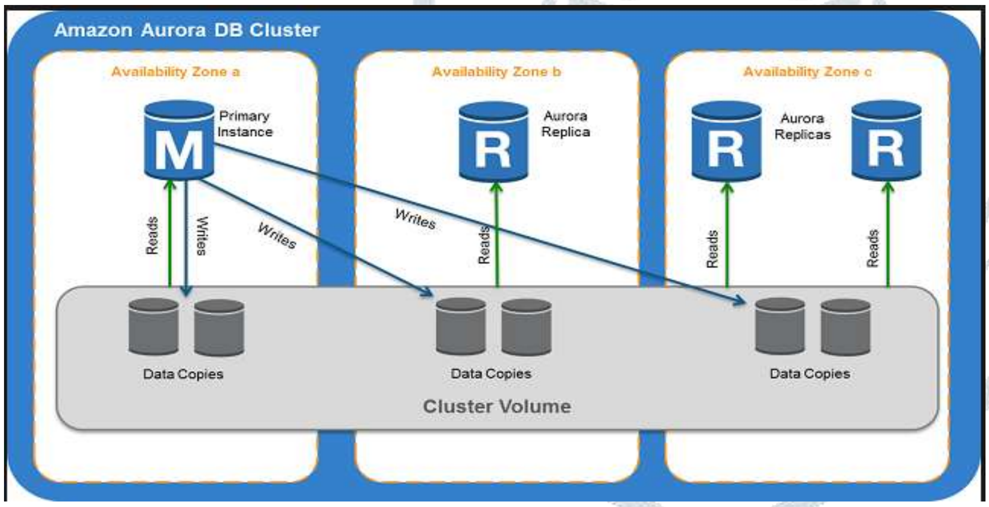
The whole assessment depends on the TPC-W benchmark [10]. We utilized the SUBSCRIBER table, parceled it vertically into 2 segments (Fig. 4.1) and utilized the TPC-W usage for Aurora MySQL 5.6 from [11].

The parallel query highlight utilizes the major compositional standards of Aurora MySQL: decoupling the database motor from the capacity subsystem, and decreasing organization traffic by smoothing out correspondence conventions. Aurora MySQL utilizes these methods to accelerate compose escalated tasks, for example, re-try log handling. Equal inquiry applies similar standards to understand tasks.

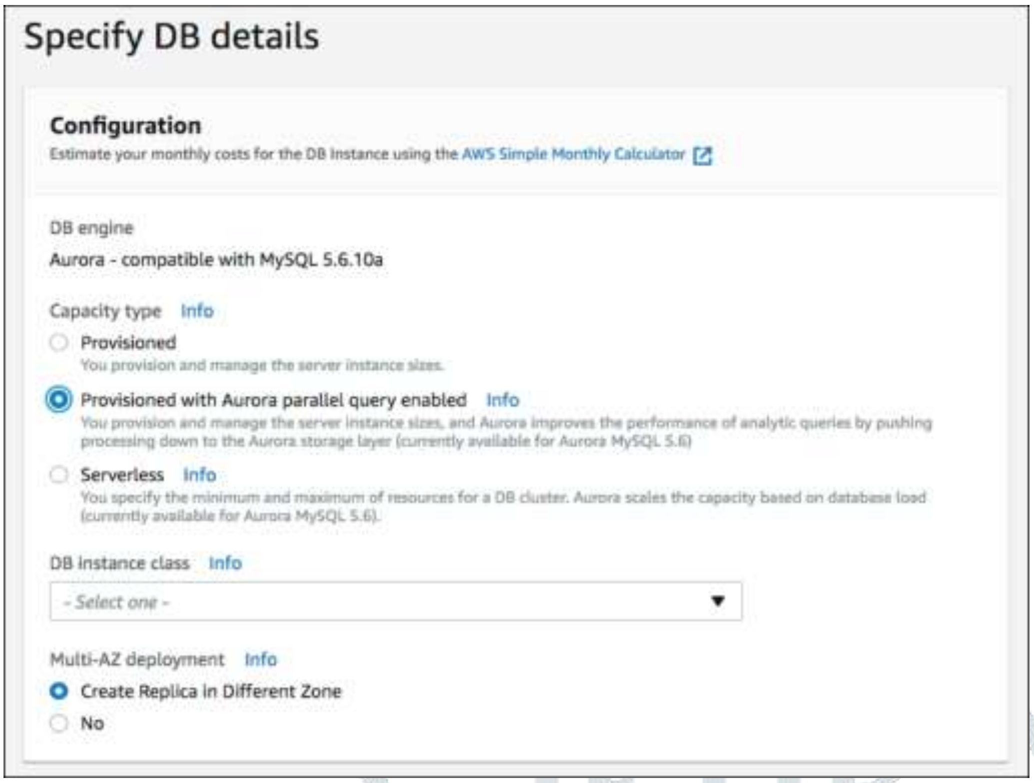
The design of Aurora MySQL equal inquiry contrasts from that of correspondingly named highlights in other database frameworks. Aurora MySQL equal inquiry doesn't include symmetric multiprocessing (SMP) thus doesn't rely upon the CPU limit of the database worker. The equal execution occurs in the capacity layer, autonomous of the Aurora MySQL worker that fills in as the question facilitator.

Of course, without equal inquiry, the handling for an Aurora question includes sending crude information to a solitary hub inside the Aurora bunch (the head hub) and playing out all further preparing in a solitary string on that solitary hub. With equal question, a lot of this I/O-concentrated and CPU-serious work is designated to hubs in the capacity layer. Just the smaller lines of the outcome set are communicated back to the head hub, with lines previously separated, and segment esteems previously extricated and changed. The exhibition advantage originates from the decrease in network traffic, decrease in CPU use on the head hub, and parallelizing the I/O over the capacity hubs. The measure of equal I/O, separating, and projection is free of the quantity of DB cases in the Aurora bunch that runs the inquiry.

I have created an Aurora MySQL cluster with parallel query, it is MySQL 5.6 compatible database cluster works with parallel query by using. Created a DB cluster by using the AWS management console and AWS CLI to carry out my work and tests results. MySQL version



Source: Amazon Aurora DB Cluster



Specify DB details

Configuration

Estimate your monthly costs for the DB Instance using the AWS Simple Monthly Calculator

DB engine

Aurora - compatible with MySQL 5.6.10a

Capacity type Info

- Provisioned
  You provision and manage the server instance sizes.
- [✓ Provisioned with Aurora parallel query enabled] Info
  You provision and manage the server instance sizes, and Aurora improves the performance of analytic queries by pushing processing down to the Aurora storage layer (currently available for Aurora MySQL 5.6)
- Serverless Info
  You specify the minimum and maximum of resources for a DB cluster. Aurora scales the capacity based on database load (currently available for Aurora MySQL 5.6).

DB instance class Info

- Select one -

Multi-AZ deployment Info

[✓ Create Replica in Different Zone] No

Figure 4.1: Snapshot of GUI while creating Aurora cluster.

Following Option have been taken while creating the Aurora MySQL5.6 DB Cluster for to test Parallelism and Storage.

- To get the output in JSON format, AWS cloud property AWS –RDS (Relational Database) has been used . The following code example shows how.
- aws rds describe property to get the details of the AWS-RDS used which gives us engine information and the region used is default us-east-2 by using following command: *aws rds describe -orderable-db-instance-options --engine aurora --engine-mode parallelquery --region us-east-2*
- AWS management console and AWS CLI has been used to create Amazon Aurora database cluster
- We took following options to while creating cluster: For the --engine option, use aurora.
- --Enigine -Mode → ParallelQuery, this option is required while creating DB cluster.
- Enigine Version → Aurora 5.6.1 10a

Following command snippet was used to configure the Aurora MySQL DB cluster:

- Create DB cluster --- aws rds create-db-cluster-identifier $CLUSTER_ID
- --Engine -- aurora
- --Engine-mode – parallelquery
- --engine-version – 5.6.1 10a

- --master-username -- $MASTER_USER_ID
- --master-user-password -- $MASTER_USER_PW
- --db-subnet-group-name -- $SUBNET_GROUP
- --vpc-security-group-ids -- $SECURITY_GROUP
- *aws rds create-db-instance --db-instance-identifier ${INSTANCE_ID}-1 \ --engine aurora \ --db-cluster-identifier $CLUSTER_ID --db-instance-class $INSTANCE_CLASS*

After successful creation of Aurora DB Cluster, I have verified that a new DB cluster can use parallel query.

1. Create or restore a cluster using the preceding techniques.
2. Check that the aurora_pq_supported configuration setting is true.
3. mysql> select @@aurora_pq_supported;
4. +-----------------------+
5. | @@aurora_pq_supported |
6. +-----------------------+
7. | 1 |
8. +-----------------------+

To flip the aurora_pq boundary at the meeting level, for instance through the mysql order line or inside a JDBC or ODBC application, the order on the standard MySQL customer is set meeting aurora_pq = {'ON'/'OFF'}. You can likewise add the meeting level boundary to the JDBC setup or inside your application code to empower or debilitate equal question progressively.

To test the parallel component of the query the schema objects/tables created in such a manner that PARTITION BY clause should not be considering. In Aurora MySQL The parallel query requires tables to use the ROW_FORMAT=Compact setting. Also, the parallel query currently (in this configuration) requires to be nonpartitioned. We have copy all the partition tables data into nonpartitioned tables with same column definition and indexes. Then rename old and new tables so that the nonpartitioned table is used by existing queries and ETL workflows

We have changed the blueprint to empower equal inquiry to work with more tables, led tests to affirm if resemble question brings about a net expansion in execution of inquiries on those tables and ensuring that the pattern necessity for equal inquiry are generally viable with our objectives.

In common activity, you don't have to play out any extraordinary moves to make favorable position of equal inquiry. After an inquiry meets the fundamental necessities for equal question, the question analyzer consequently concludes whether to utilize equal question for every particular question.

In the event that we run tests in a turn of events or test climate, we may locate that equal inquiry isn't utilized in light of the fact that our tables are excessively little of lines or generally information volume. The information for the table may likewise be totally in the cradle pool, particularly for tables we made as of late to perform tests.

For checking and overseeing reason or tuning the DB execution, we have to choose whether parallelism is being utilized in the suitable settings. We may need to change the database pattern, settings, SQL questions and reports and even bunch geography and application association settings to exploit parallelism include.

All through this segment, I have utilized tables from TPC-H dataset especially for Subscribers table. This tables is having around 40 million columns and following table definition:

```
+---------------+---------------+------+-----+---------+-------+
| Record        | Record_Type   | Null |Key  |Default | Extra |
+---------------+---------------+------+-----+---------+-------+
| Subscribers_ID| int(11)       | NO   | PRI | NULL    |       |
| s_name        | varchar(55)   | NO   |     | NULL    |       |
| s_msisdn      | char(25)      | NO   |     | NULL    |       |
| s_imsi        | char(10)      | NO   |     | NULL    |       |
| s_type        | varchar(25)   | NO   |     | NULL    |       |
| s_imei        | int(11)       | NO   |     | NULL    |       |
| s_handset     | char(10)      | NO   |     | NULL    |       |
| s_vendor      | decimal(15,2) | NO   |     | NULL    |       |
| s_price   | varchar(23)   | NO   |     | NULL    |       |
+---------------+---------------+------+-----+---------+-------+
```

So , to check whether our queries is using parallelism techniques , we have checked the same with the help of explain output .

First we tried by disabling parallel query, as per the explain plan of the query, it is using hash joins rather than parallel

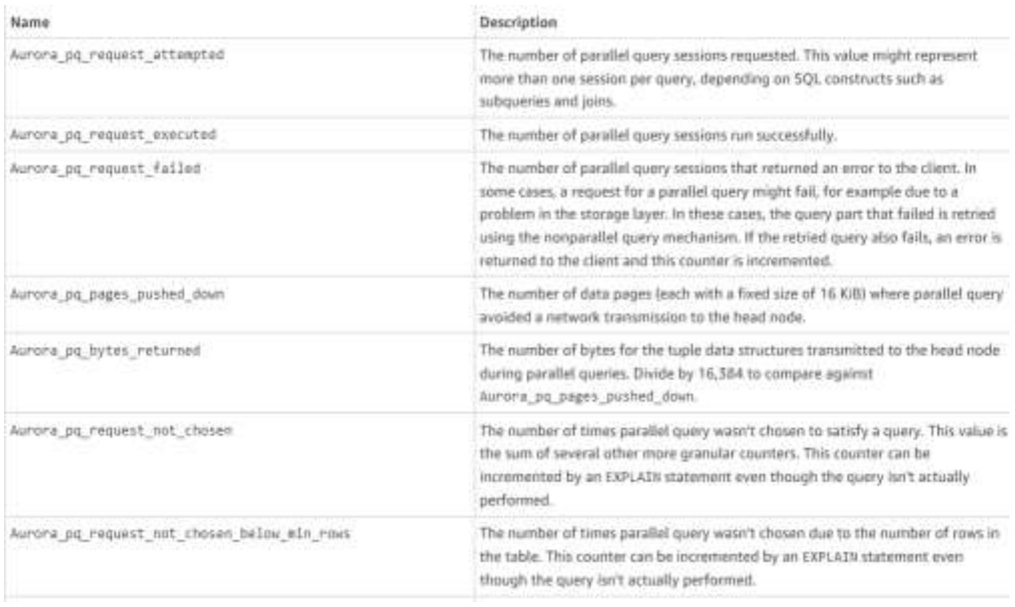
```
+----+-------------+----------+...+-----------+------------------------------------------+
| id | select_type | table    |...| rows      | Extra|
+----+-------------+----------+...+-----------+------------------------------------------+
|  1 | SIMPLE      | Subscribers |...|   6218213 | Using where; Using index; Using temporary; Using filesort       |
|  1 | SIMPLE      | Handsets   |...| 213841403 | Using where; Using join buffer (Hash Join Outer table orders)   |
|  1 | SIMPLE      | Tac |...| 581199404 | Using where; Using join buffer (Hash Join Outer table Tac) |
+----+-------------+----------+...+-----------+------------------------------------------+
```

Secondly, After enabling the parallelization, the two where clauses are using parallel query optimization, as shown in below explain plan of the query.

```
+----+...+------------------------------------------------------------------------------+
|id      |...|   Extra                                                                  |
+----+...+------------------------------------------------------------------------------+
|  1 |...| Using where; Using index; Using temporary; Using filesort                     |
|  1 |...| Using where; Using join buffer (Hash Join Outer table orders); Using parallel query (5 columns, 1 filters, 1 exprs; 0 extra)   |
|  1 |...| Using where; Using join buffer (Hash Join Outer table lineitem); Using parallel query (5 columns, 1 filters, 1 exprs; 0 extra) |
+----+...+------------------------------------------------------------------------------+
```

Now, we have to check for monitoring techniques to help verify how often the parallel query is used in the given workloads and side by side checking the performance of the same.

As we are testing on Amazon Aurora, we can use Amazon CloudWatch metrics and global status variables which is helpful to monitor parallel query execution and give us insights into optimizer statistics and can show us as to why query is using and not using parallel query while executing. While tracking the counters at DB instance level we have seen that when we connect with different endpoints we saw different metrics because each DB instance runs its own set of parallel queries.



Name	Description
Aurora_pq_request_attempted	The number of parallel query sessions requested. This value might represent more than one session per query, depending on SQL constructs such as subqueries and joins.
Aurora_pq_request_executed	The number of parallel query sessions run successfully.
Aurora_pq_request_failed	The number of parallel query sessions that returned an error to the client. In some cases, a request for a parallel query might fail, for example due to a problem in the storage layer. In these cases, the query part that failed is retried using the nonparallel query mechanism. If the retried query also fails, an error is returned to the client and this counter is incremented.
Aurora_pq_pages_pushed_down	The number of data pages (each with a fixed size of 16 KiB) where parallel query avoided a network transmission to the head node.
Aurora_pq_bytes_returned	The number of bytes for the tuple data structures transmitted to the head node during parallel queries. Divide by 16,384 to compare against Aurora_pq_pages_pushed_down.
Aurora_pq_request_not_chosen	The number of times parallel query wasn't chosen to satisfy a query. This value is the sum of several other more granular counters. This counter can be incremented by an EXPLAIN statement even though the query isn't actually performed.
Aurora_pq_request_not_chosen_below_min_rows	The number of times parallel query wasn't chosen due to the number of rows in the table. This counter can be incremented by an EXPLAIN statement even though the query isn't actually performed.

Figure 4.2: Amazon Aurora Parallel query DB metrics

Source: https://aws.amazon.com/rds/aurora/faqs/

For the next phase of testing, we have focused on how a parallel query works on SQL constructs that is how a particular SQL statements using and not using parallel query and how Aurora MySQL react with parallel query. We likewise have analyze execution issues for a cluster that utilizes parallel query and to see how parallel query applies for our database workload.

There is number of conditions in the SQL query on which the output and performance of the query depends. We are taking some of the clauses/conditions/predicates used in SQL query to check our workload and performance of the queries we have executed while testing the performance of the parallel query.

4.1.1 Parallel Query – Test and Performance aspects of Clauses and Conditions

I have used following clauses/conditions/predicates in our queries to produce the result and form our analysis based on the data captured while performing test results.

4.1.2 Parallel Query Execution and Performance

As per the previous readings and experiments, generally MYSQL did not perform well with multiple CPUSs. There was the time when MySQL shows poorer performance with higher number of CPU cores then with less number of CPU cores. MySQL 5.6 did overcome this limitation and can scale-up with multiple CPUs but still one highly intensive query executes on only one CPU with no parallelism.

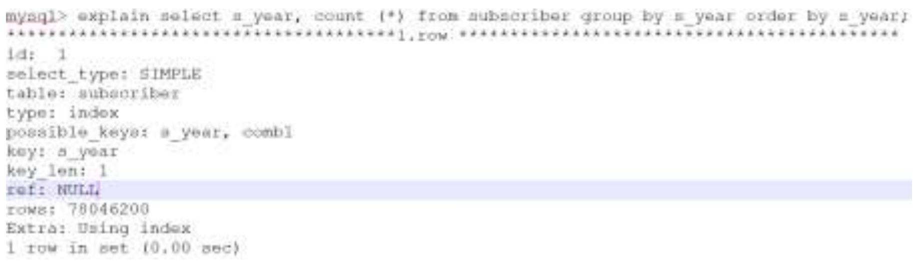
Aurora MySQL parallel query can overcome this limitation and we have tested our MySQL DB cluster to chow the same. We used telecom subscriber's database to test out the parallel execution with very large tables in this case "SUBSCRIBER" table having around 40 million rows. We here now demonstrating how parallel query works and side by side to improve the performance of the report queries.

```
+---------------+---------------+------+-----+---------+-------+
| Column        |Data_Type      | Null | Key | Default | Extra |
+---------------+---------------+------+-----+---------+-------+
| Subscribers_ID| int(11)       | NO   | PRI | NULL    |       |
| s_name        | varchar(45)   | NO   |     | NULL    |       |
| s_msisdn      | char(15)      | NO   |     | NULL    |       |
| s_imsi        | char(15)      | NO   |     | NULL    |       |
| s_type        | varchar(35)   | NO   |     | NULL    |       |
| s_imei        | int(11)       | NO   |     | NULL    |       |
| StartDate     | Date          | NO   |     | NULL    |       |
| EndDate       | Date          | NO   |     | NULL    |       |
| s_year        | year(4)       | NO   |     | NOT NULL|       |
| s_handset     | char(10)      | NO   |     | NULL    |       |
| s_vendor      | decimal(15,2) | NO   |     | NULL    |       |
| s_price       | varchar(23)   | NO   |     | NULL    |       |
+---------------+---------------+------+-----+---------+-------+
```

We have created subscriber table with above table structure and load data of Airtel subscribers base which is around 40 million rows and size is 83GB.

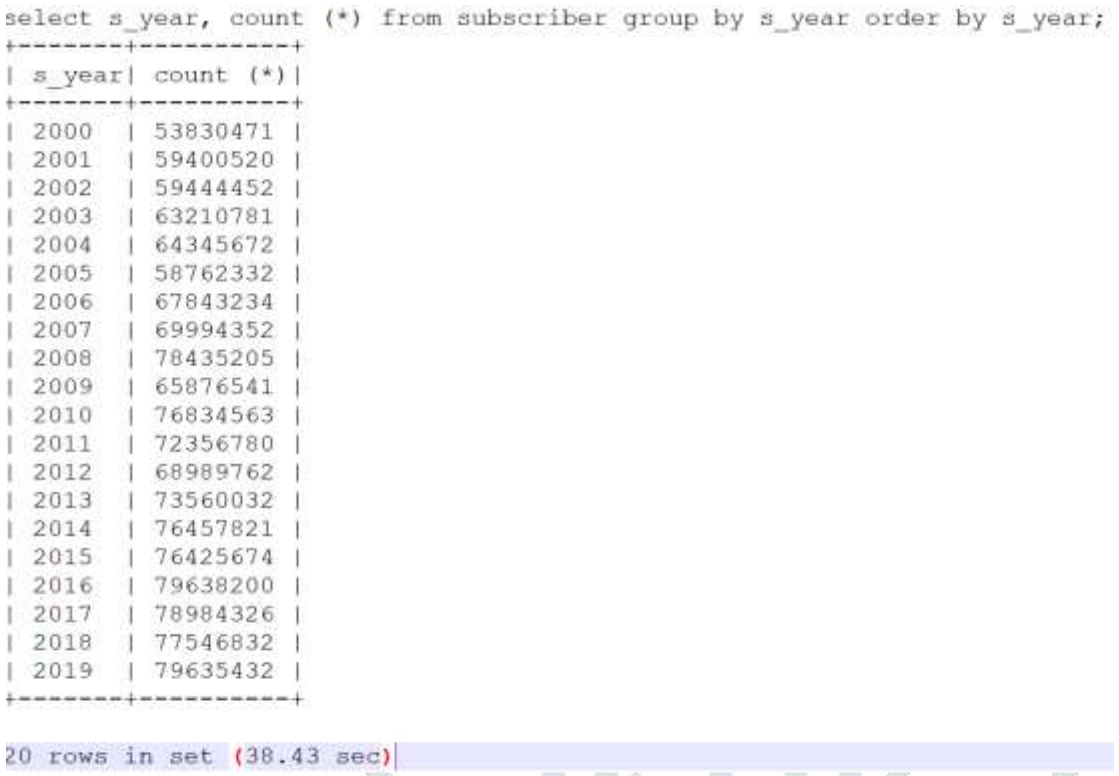
Now we executed some queries to demonstrate the execution to find all the subscribers logged into network year-wise.

1. *select s_year, count(*) from subscriber group by s_year*



```
mysql> explain select s_year, count (*) from subscriber group by s_year order by s_year;
***************************************1.row *******************************************
id:  1
select_type: SIMPLE
table: subscriber
type: index
possible_keys: s_year, comb1
key: s_year
key_len: 1
ref: NULL
rows: 78046200
Extra: Using index
1 row in set (0.00 sec)
```

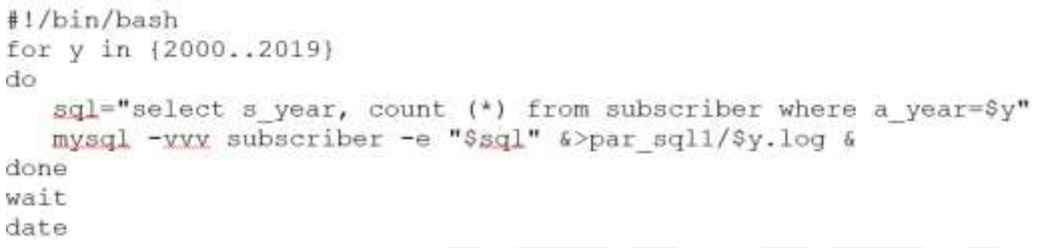
The query is simple and extract around 80 million of rows. Below is the result of query cached.



```
select s_year, count (*) from subscriber group by s_year order by s_year;
+--------+----------+
| s_year| count (*)|
+--------+----------+
| 2000   | 53830471 |
| 2001   | 59400520 |
| 2002   | 59444452 |
| 2003   | 63210781 |
| 2004   | 64345672 |
| 2005   | 58762332 |
| 2006   | 67843234 |
| 2007   | 69994352 |
| 2008   | 78435205 |
| 2009   | 65876541 |
| 2010   | 76834563 |
| 2011   | 72356780 |
| 2012   | 68989762 |
| 2013   | 73560032 |
| 2014   | 76457821 |
| 2015   | 76425674 |
| 2016   | 79638200 |
| 2017   | 78984326 |
| 2018   | 77546832 |
| 2019   | 79635432 |
+--------+----------+

20 rows in set (38.43 sec)
```

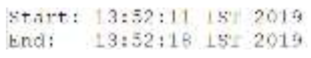
The query took 38.43 seconds and utilized only 1 CPU core. Hence not using the parallel query functionality but this query is good candidate for executing in parallel. To make this query to run in parallel, we will create a shell script which will run 14 parallel queries, and each will count the year-wise data that is each query will run for one year parallelly.



```
#!/bin/bash
for y in {2000..2019}
do
   sql="select s_year, count (*) from subscriber where a_year=$y"
   mysql -vvv subscriber -e "$sql" &>par_sql1/$y.log &
done
wait
date
```

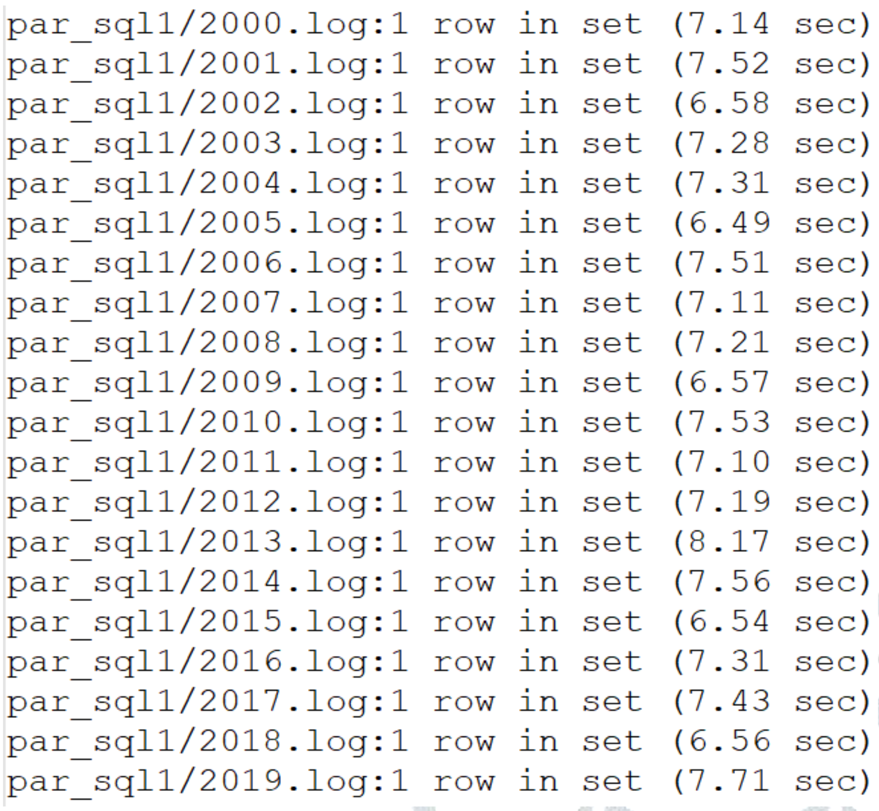
Following is the output from the shell query

Shell



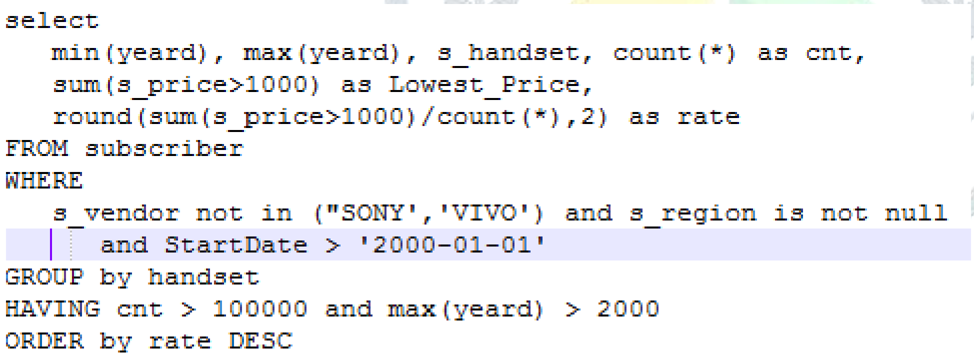
```
Start: 13:52:11 IST 2019
End:   13:52:18 IST 2019
```

Individual output from all parallel query statements



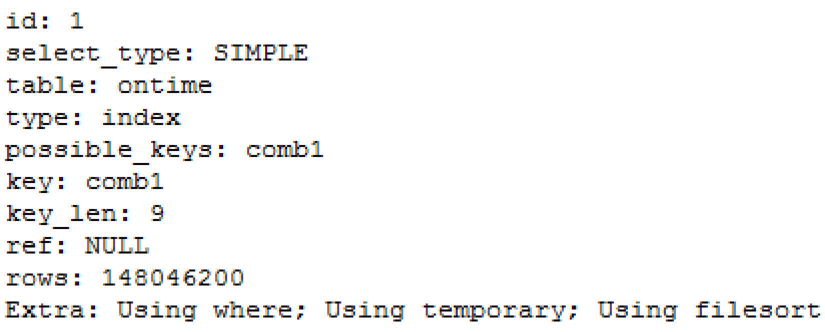
```
par_sql1/2000.log:1 row in set (7.14 sec)
par_sql1/2001.log:1 row in set (7.52 sec)
par_sql1/2002.log:1 row in set (6.58 sec)
par_sql1/2003.log:1 row in set (7.28 sec)
par_sql1/2004.log:1 row in set (7.31 sec)
par_sql1/2005.log:1 row in set (6.49 sec)
par_sql1/2006.log:1 row in set (7.51 sec)
par_sql1/2007.log:1 row in set (7.11 sec)
par_sql1/2008.log:1 row in set (7.21 sec)
par_sql1/2009.log:1 row in set (6.57 sec)
par_sql1/2010.log:1 row in set (7.53 sec)
par_sql1/2011.log:1 row in set (7.10 sec)
par_sql1/2012.log:1 row in set (7.19 sec)
par_sql1/2013.log:1 row in set (8.17 sec)
par_sql1/2014.log:1 row in set (7.56 sec)
par_sql1/2015.log:1 row in set (6.54 sec)
par_sql1/2016.log:1 row in set (7.31 sec)
par_sql1/2017.log:1 row in set (7.43 sec)
par_sql1/2018.log:1 row in set (6.56 sec)
par_sql1/2019.log:1 row in set (7.71 sec)
```

Now we are using bit complex query where we are calculating number of handset counts year-wise since 2000 and excluding SO(SONY) and VI(VIVO) as vendor. As the query has “group by” and “order by” and multiple ranges in the where clause it will have to create a temporary table:



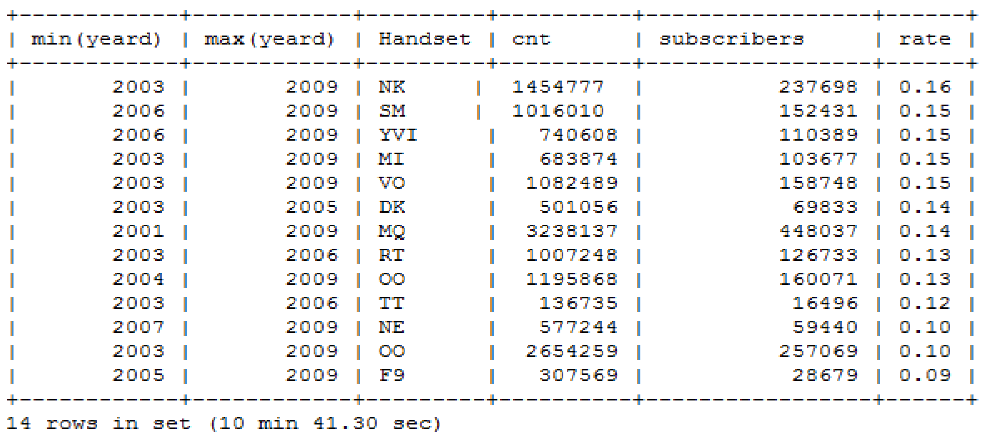
```
select
   min(yeard), max(yeard), s_handset, count(*) as cnt,
   sum(s_price>1000) as Lowest_Price,
   round(sum(s_price>1000)/count(*),2) as rate
FROM subscriber
WHERE
   s_vendor not in ("SONY','VIVO') and s_region is not null
      and StartDate > '2000-01-01'
GROUP by handset
HAVING cnt > 100000 and max(yeard) > 2000
ORDER by rate DESC
```

MySQL



```
id: 1
select_type: SIMPLE
table: ontime
type: index
possible_keys: comb1
key: comb1
key_len: 9
ref: NULL
rows: 148046200
Extra: Using where; Using temporary; Using filesort
```

To increase performance a combined index has been used as comb1 on columns s_handset,yeard and s_price

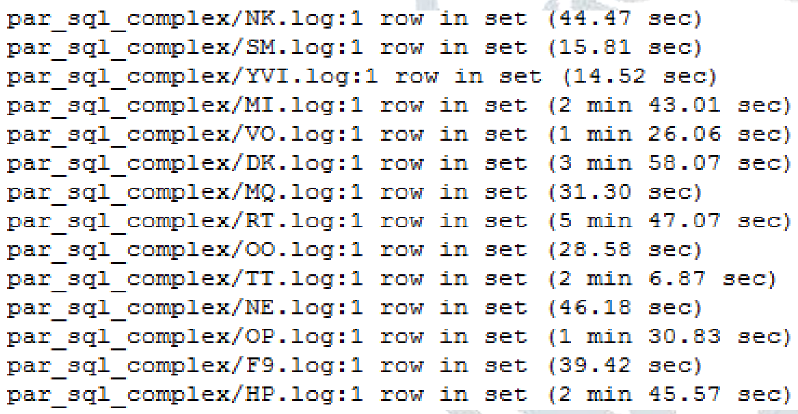


```
+------------+------------+---------+----------+-----------------+------+
| min(yeard) | max(yeard) | Handset | cnt      | subscribers     | rate |
+------------+------------+---------+----------+-----------------+------+
|       2003 |       2009 | NK      | 1454777  |          237698 | 0.16 |
|       2006 |       2009 | SM      | 1016010  |          152431 | 0.15 |
|       2006 |       2009 | YVI     |   740608 |          110389 | 0.15 |
|       2003 |       2009 | MI      |   683874 |          103677 | 0.15 |
|       2003 |       2009 | VO      |  1082489 |          158748 | 0.15 |
|       2003 |       2005 | DK      |   501056 |           69833 | 0.14 |
|       2001 |       2009 | MQ      |  3238137 |          448037 | 0.14 |
|       2003 |       2006 | RT      |  1007248 |          126733 | 0.13 |
|       2004 |       2009 | OO      |  1195868 |          160071 | 0.13 |
|       2003 |       2006 | TT      |   136735 |           16496 | 0.12 |
|       2007 |       2009 | NE      |   577244 |           59440 | 0.10 |
|       2003 |       2009 | OO      |  2654259 |          257069 | 0.10 |
|       2005 |       2009 | F9      |   307569 |           28679 | 0.09 |
+------------+------------+---------+----------+-----------------+------+
14 rows in set (10 min 41.30 sec)
```

Now we have executed the query and splitting it into 14 queries in parallel. The result set is 3 times faster and we also avoid creating temporary table while executing.

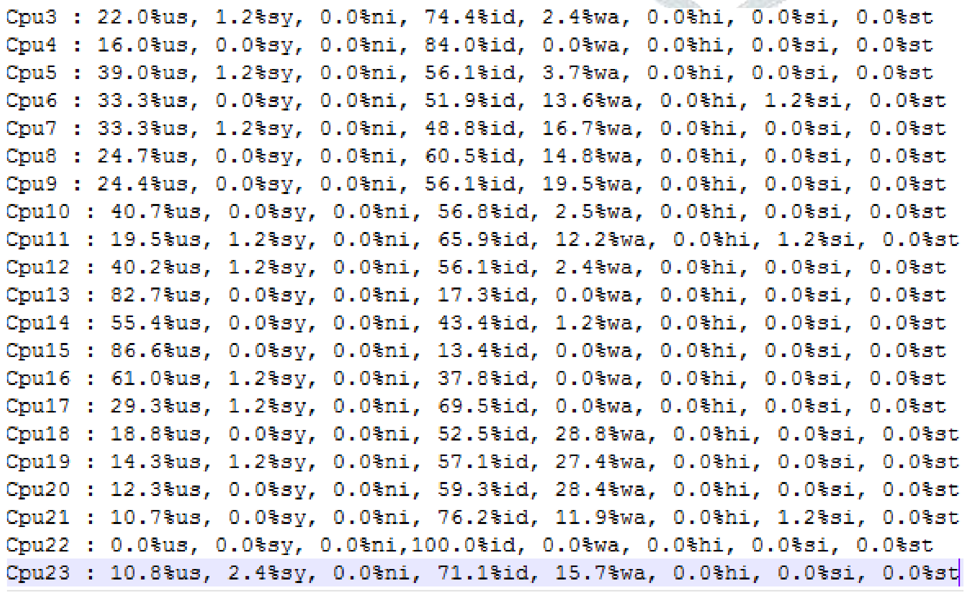
Results: total time is 3 min 17 seconds (3times faster)

Statistics Per query:



```
par_sql_complex/NK.log:1 row in set (44.47 sec)
par_sql_complex/SM.log:1 row in set (15.81 sec)
par_sql_complex/YVI.log:1 row in set (14.52 sec)
par_sql_complex/MI.log:1 row in set (2 min 43.01 sec)
par_sql_complex/VO.log:1 row in set (1 min 26.06 sec)
par_sql_complex/DK.log:1 row in set (3 min 58.07 sec)
par_sql_complex/MQ.log:1 row in set (31.30 sec)
par_sql_complex/RT.log:1 row in set (5 min 47.07 sec)
par_sql_complex/OO.log:1 row in set (28.58 sec)
par_sql_complex/TT.log:1 row in set (2 min 6.87 sec)
par_sql_complex/NE.log:1 row in set (46.18 sec)
par_sql_complex/OP.log:1 row in set (1 min 30.83 sec)
par_sql_complex/F9.log:1 row in set (39.42 sec)
par_sql_complex/HP.log:1 row in set (2 min 45.57 sec)
```

By running the query is parallel we got 3 times better response time on the server .

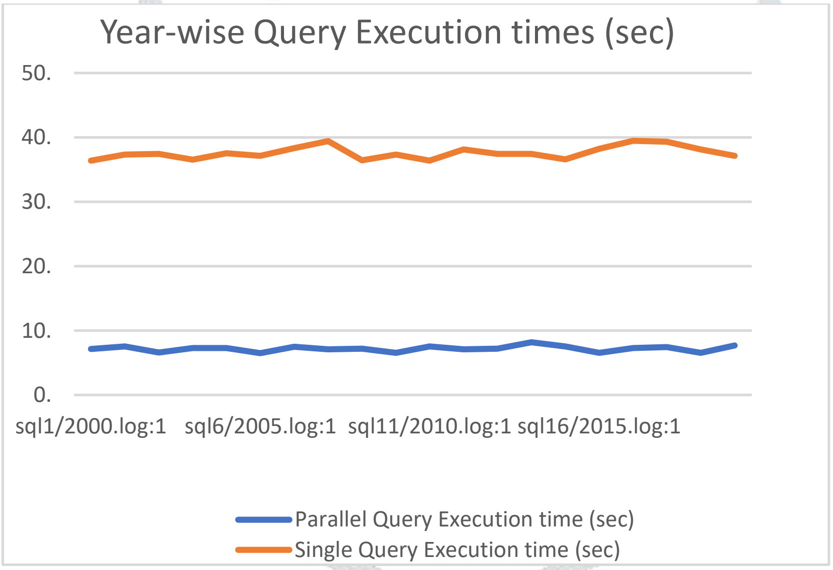


```
Cpu3 : 22.0%us, 1.2%sy, 0.0%ni, 74.4%id, 2.4%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu4 : 16.0%us, 0.0%sy, 0.0%ni, 84.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu5 : 39.0%us, 1.2%sy, 0.0%ni, 56.1%id, 3.7%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu6 : 33.3%us, 0.0%sy, 0.0%ni, 51.9%id, 13.6%wa, 0.0%hi, 1.2%si, 0.0%st
Cpu7 : 33.3%us, 1.2%sy, 0.0%ni, 48.8%id, 16.7%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu8 : 24.7%us, 0.0%sy, 0.0%ni, 60.5%id, 14.8%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu9 : 24.4%us, 0.0%sy, 0.0%ni, 56.1%id, 19.5%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu10 : 40.7%us, 0.0%sy, 0.0%ni, 56.8%id, 2.5%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu11 : 19.5%us, 1.2%sy, 0.0%ni, 65.9%id, 12.2%wa, 0.0%hi, 1.2%si, 0.0%st
Cpu12 : 40.2%us, 1.2%sy, 0.0%ni, 56.1%id, 2.4%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu13 : 82.7%us, 0.0%sy, 0.0%ni, 17.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu14 : 55.4%us, 0.0%sy, 0.0%ni, 43.4%id, 1.2%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu15 : 86.6%us, 0.0%sy, 0.0%ni, 13.4%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu16 : 61.0%us, 1.2%sy, 0.0%ni, 37.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu17 : 29.3%us, 1.2%sy, 0.0%ni, 69.5%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu18 : 18.8%us, 0.0%sy, 0.0%ni, 52.5%id, 28.8%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu19 : 14.3%us, 1.2%sy, 0.0%ni, 57.1%id, 27.4%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu20 : 12.3%us, 0.0%sy, 0.0%ni, 59.3%id, 28.4%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu21 : 10.7%us, 0.0%sy, 0.0%ni, 76.2%id, 11.9%wa, 0.0%hi, 1.2%si, 0.0%st
Cpu22 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu23 : 10.8%us, 2.4%sy, 0.0%ni, 71.1%id, 15.7%wa, 0.0%hi, 0.0%si, 0.0%st
```

Table: Overall CPU usage of Server

Year-Wise	Single Query Execution time (sec)	Parallel Query Execution time (sec)
sql1/2000.log:1	36.42	7.14
sql2/2001.log:1	37.35	7.52
sql3/2002.log:1	37.42	6.58
sql4/2003.log:1	36.56	7.28
sql5/2004.log:1	37.55	7.31
sql6/2005.log:1	37.12	6.49
sql7/2006.log:1	38.33	7.51
sql8/2007.log:1	39.42	7.11
sql9/2008.log:1	36.44	7.21
sql10/2009.log:1	37.36	6.57
sql11/2010.log:1	36.41	7.53
sql12/2011.log:1	38.12	7.1
sql13/2012.log:1	37.42	7.19
sql14/2013.log:1	37.45	8.17
sql15/2014.log:1	36.59	7.56
sql16/2015.log:1	38.22	6.54
sql17/2016.log:1	39.46	7.31
sql18/2017.log:1	39.32	7.43
sql19/2018.log:1	38.12	6.56
sql20/2019.log:1	37.16	7.71

Table 4.1 – Query performance – Lazy v/s Parallel fetch



Graph 4.1- Query performance – Lazy v/s Parallel fetch

Hence, we can say that, splitting a large report into multiple parallel query will enhance the performance of the simple and complex reports executed on large tables. Scalability can also increase by splitting the queries on multiple MySQL slave servers.

5. Conclusion And Recommendations

On the basis of our experiments in chapter 4 of this thesis we have seen significant enhancement in performance results when compared with both lazy and parallel fetch approach and same has been compared between chache based databases and non-distributed databases.

As we have checked our performance results on latest Amazon RDS by using Amazon Aurora MySQL 5.6 database. Earlier MySQL database was not the good candidate when it comes to perform on multiple CPU's or we can say that prior to introduction of Amazon Aurora MySQL 5.6 and higher versions, the MySQL works badly when it comes to multiple CPU's.

Apart from the database selection further we require an approach which can works on high number of tuples (288K in our experiments) , the *parallel fetch* approach we used in our experiments when compared with non-parallel approach i.e lazy fetch. We saw significant improvement in performance of the queries and actually it is 3 times faster when using parallelism.

If we look on the Fig 5.1, we can see the performance comparison of both the approaches and it further shows that for lower workloads or when the volume of data is nearly upto 250 tuples, the performance of both the approach is nearly same but when the data volume increases the performance of both parallel and lazy or non-parallel approach differs and *parallel fetch* far outclass the *lazy fetch* or non-parallel fetch approach when it comes to performance on large datasets.

So we can say that the parallel fetch approach is the answer when we deals with the fast OLAP queries based on both horizontally and vertically distributed data in distributed and non-distributed databases.

This *parallel fetch* approach is yet to implement or experimented on new clouds databases like MariaDB or Postgres databases and it will be good to see how this cache mechanisms works on other cloud databases.

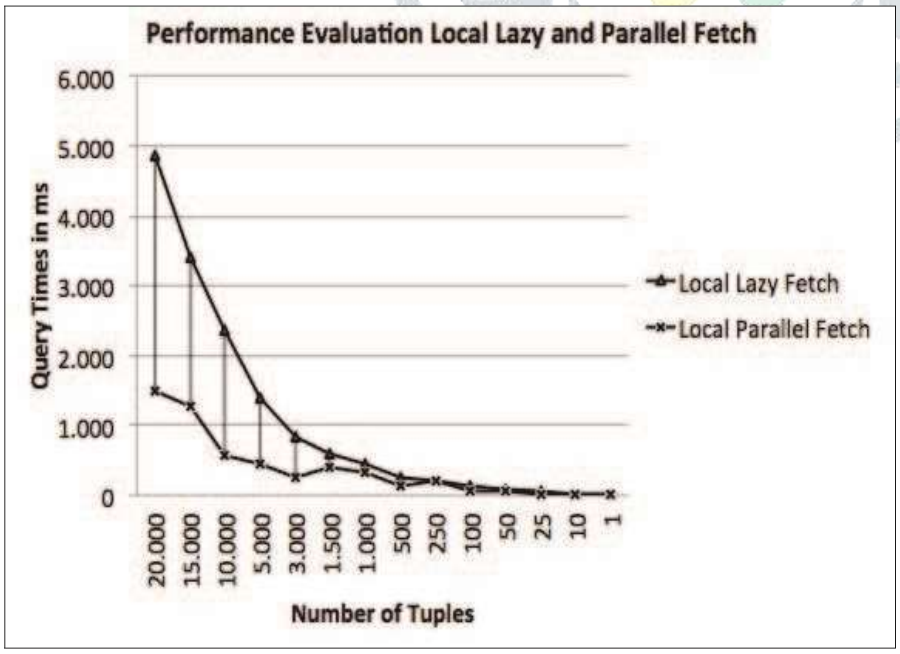


Fig. 5.1. Lazy Fetch v/s Parallel Fetch graphical representation.

References

1. Samaniego, M., Jamsrandorj, U. and Deters, R., 2016, December. Blockchain as a Service for IoT. In *2016 IEEE international conference on internet of things (iThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData)* (pp. 433-436). IEEE.

2. Islam, M., Dinh, A., Wahid, K. and Bhowmik, P., 2017, April. Detection of potato diseases using image segmentation and multiclass support vector machine. In *2017 IEEE 30th Canadian conference on electrical and computer engineering (CCECE)* (pp. 1-4). IEEE.

3. Su, H., Cai, Y. and Du, Q., 2016. Firefly-algorithm-inspired framework with band selection and extreme learning machine for hyperspectral image classification. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, *10*(1), pp.309-320.

4. Goldberg, K., 2011. What is automation?. *IEEE transactions on automation science and engineering*, *9*(1), pp.1-2.

5. Shakkottai, S., Rappaport, T.S. and Karlsson, P.C., 2003. Cross-layer design for wireless networks. *IEEE Communications magazine*, *41*(10), pp.74-80.

6. Pawlowski, T., 2016, May. Memory, Storage and Processing in Future Parallel and Distributed Processing Systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (pp. 473-473). IEEE.

7. Raza, M.U. and XuJian, Z., 2020, May. A Comprehensive Overview of BIG DATA Technologies: A Survey. In *Proceedings of the 2020 5th International Conference on Big Data and Computing* (pp. 23-31).

8. Published in: IEEE Journal of Biomedical and Health Informatics (Volume: 21, Issue: 4, July 2017) Page(s): 1049 – 1057 Date of Publication: 13 June 2016; ISSN Information: PubMed ID: 27323383; DOI: 10.1109/JBHI.2016.2580145; Publisher: IEEE.

9. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D. and Moon, B., 2012. Parallel data processing with MapReduce: a survey. *AcM sIGMoD Record*, *40*(4), pp.11-20.

10. Rackl, G., 2001. *Monitoring and managing heterogeneous middleware* (Doctoral dissertation, Technische Universität München).

11. Watson, P. and Townsend, P., 1990, September. The EDS parallel relational database system. In *Workshop on Parallel Database Systems* (pp. 149-166). Springer, Berlin, Heidelberg.

12. Buff, H.W., 1988. Why Codd's Rule No. 6 Must be Reformulated. *SIGMOD Record*, *17*(4), pp.79-80.

13. Afrati, F.N. and Ullman, J.D., 2010, March. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology* (pp. 99-110).

INTERNET- SOURCES

1. http://www.dbs.ethz.chinternet/
2. docs.amazonaws.cn
3. ecommons.usask.ca
4. http://www.macs.hw.ac.ukinternet/
5. www.cs.cmu.edu
6. www.coursehero.com
7. www.livrozilla.com
8. www.etd.auburn.edu
9. www.aws.amazon.com