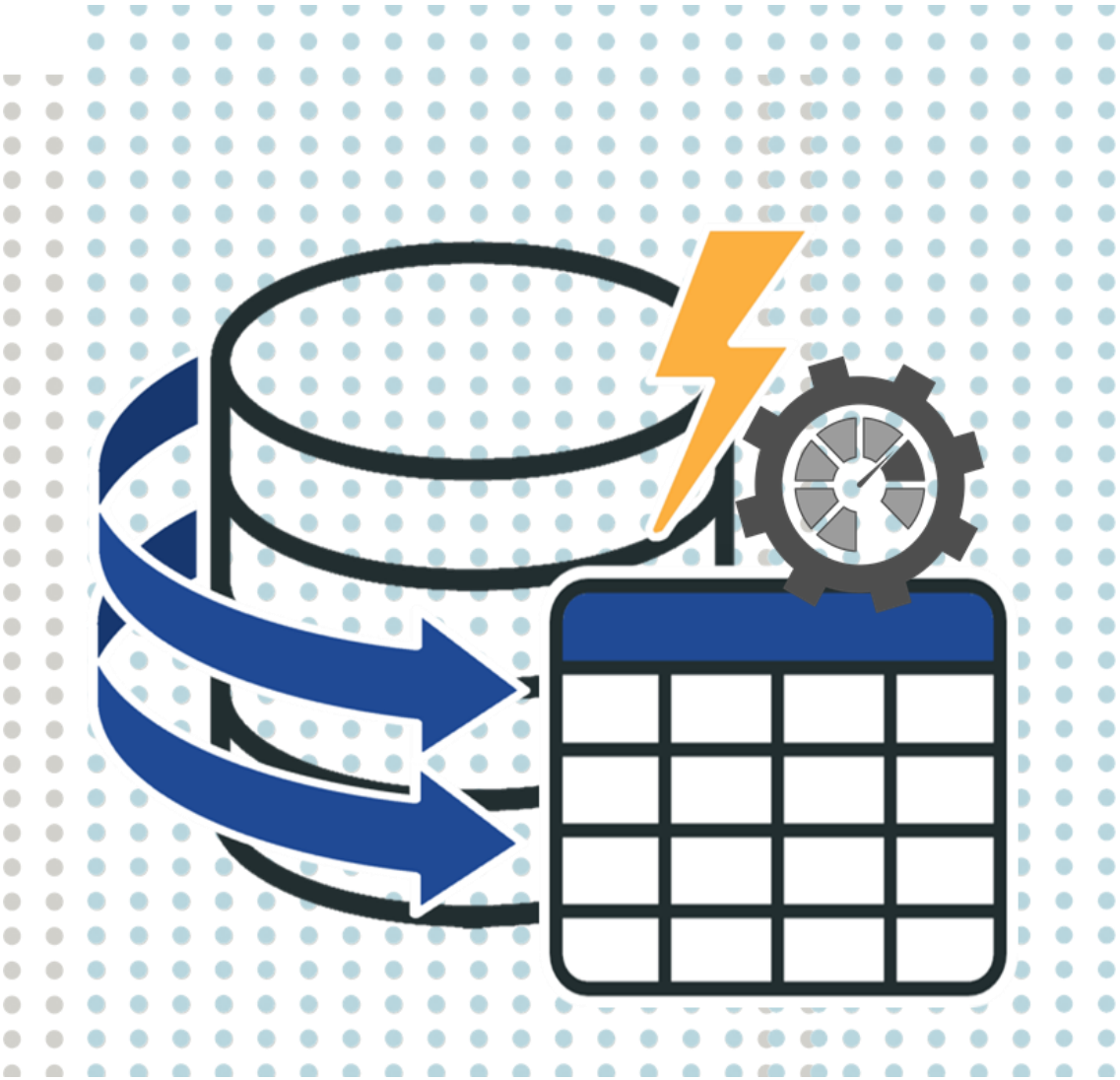
SQL Query Optimization

Guidelines

(DRAFT)



NIC-SQG-DB-GUD-001

Oct-2018

एनआईसी
National Informatics Centre

AMENDMENT LOG

Version	Date	Brief Description	Section Change
1.0	Oct 2018		

No part of this document may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of NIC.

SQL QUERY OPTIMIZATION GUIDELINES

PURPOSE:

NIC is involved in design, development and implementation of various e-governance related applications and services. These applications and services deal with enormous amount of data wherein, database plays a vital role. Storing and retrieving data efficiently determines the quality and standard of the application or service provided. An inefficient query kills the user's time and the applications performance. Query optimization comes into play when the application's performance and response time needs to be improved to enhance the user experience with the application.

This document is developed with an objective to assist NIC officials to improve the way in which queries are built and executed. It provides query optimization guidelines and best practices to validate the current querying practice and to improve the same for future endeavors.

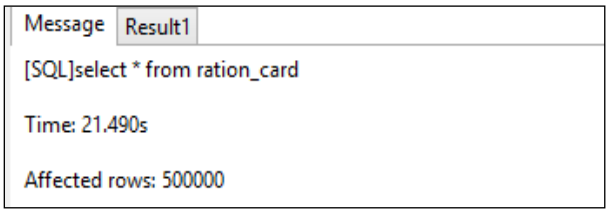
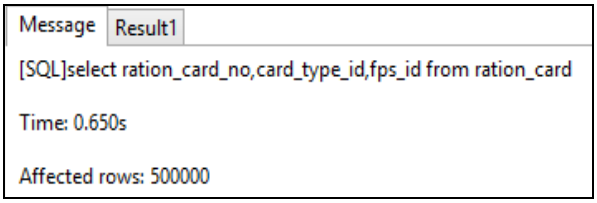
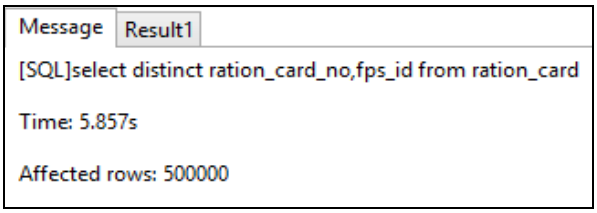
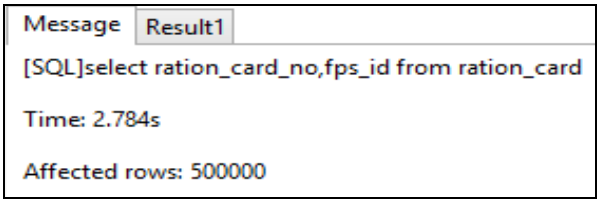
Please send your valuable feedback/suggestions to Software Quality Group at support-sqg@nic.in

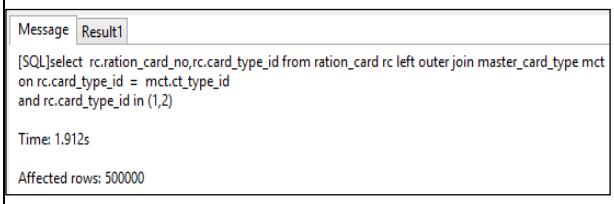
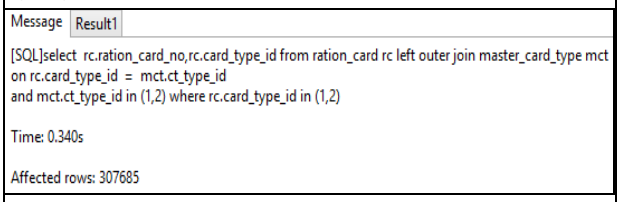
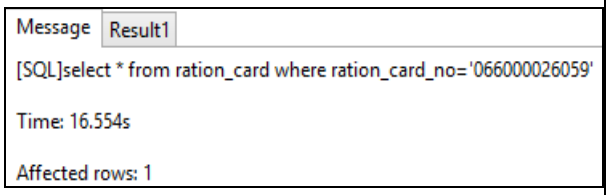
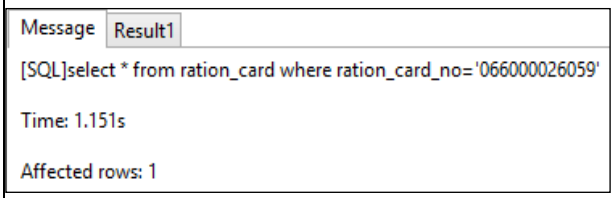
REFERENCE TABLES

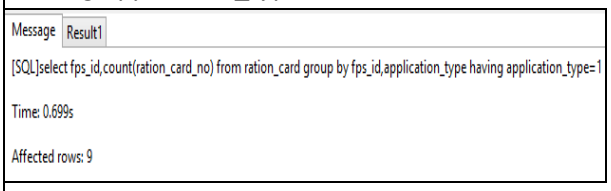
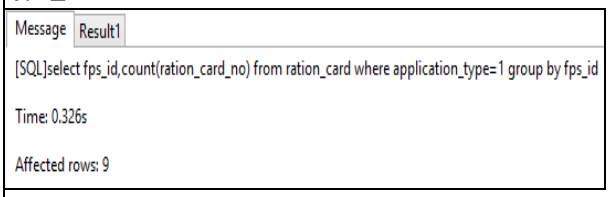
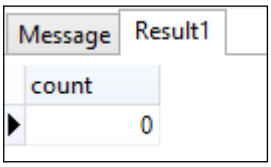
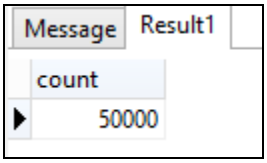
The following table schema shall be referred for the execution of SQL Query optimization techniques discussed in this document.

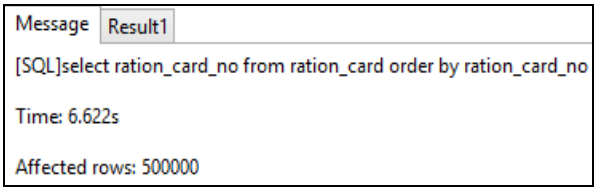
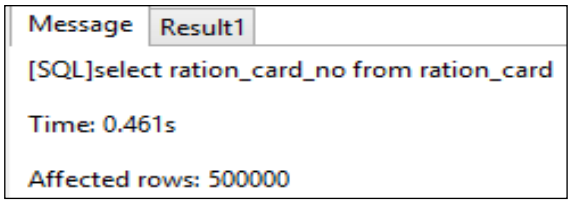
TABLE: MASTER_CARD_TYPE	
Number of Records: 6	
Column Name	**Data Type**
card_type_id	smallint
card_desc_en	character varying(150)
card_desc_ll	character varying(150)
active	integer

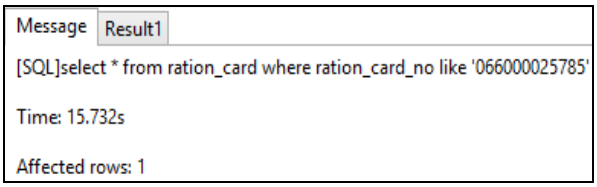
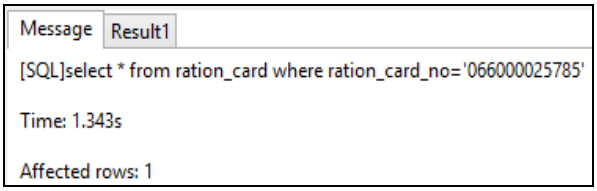
TABLE: RATION CARD	
Number of Records: 5,00,000	
Column Name	**Data Type**
ration_card_id	integer
ration_card_no	character varying(12)
card_type_id	integer
application_type	bigint
fps_id	character varying(12)
village_code	character varying(16)
panchayat_code	character varying(13)
tehsil_code	character varying(5)
subdivision_code	character varying
district_code	character varying(3)
state_code	character varying(2)
plc_code	character varying(16)
total_income	integer
income_type_id	smallint
rc_status	character varying(30)
active	smallint

S. No.	Guidelines	Description	Test Cases
1. Select Statement			
1.1	Avoid unnecessary columns in select clause. Select required columns instead of select *	• Selecting unnecessary columns can force the Database to do additional I/O. • If the query request is made from an external application (online application), choosing new/all columns compels the database to send supplementary data over the network in which case the application will take more time waiting on network I/O to send over data that it is not expected in.	**Query:** *Select * from ration_card*  **Time: 21.490s** **Optimized Query:** *Select ration_card_no,card_type_id,fps_id from ration_card*  **Time: 0.650s**
2. Distinct			
2.1	DISTINCT could be avoided if the objective can be achieved otherwise. DISTINCT requires extra sort operation and therefore slowdowns the queries.	• Distinct builds overall result set (including duplicates) based on FROM and WHERE clauses. • DISTINCT collects all the rows, including any expressions that need to be evaluated, and then tosses out duplicates. GROUP BY can filter out the duplicate rows before performing any of that work. • DISTINCT simply de-duplicates the resultant record set after all other query operations have been performed. • GROUP BY returns a single row for each unique combination of the GROUP BY fields	**Query:** *Select distinct ration_card_no,fps_id from ration_card*  **Time: 5.857s** **Optimized Query:** *Select ration_card_no,fps_id from ration_card*  **Time: 2.784s**

3. JOINS			
3.1	Duplicate conditions for constant values whenever possible	When two tables, **ration_card** with alias **rc** and m**aster_card_ type** with alias **mct**, are left joined and there is a constant predicate on one of the joined columns, eg., **rc.card_type_id=mct.ct_type_id and rc.card_type_id in (1,2)**, the constant predicate should be duplicated for the joined column of the second table. That is, **rc.card_type_id=mct.ct_type_id and rc. card_type_id in (1, 2) and mct.ct_type_id in (1,2)**.	**Query:** *select rc.ration_card_no,rc.card_type_id from ration_card rc left outer join master_card_type mct on rc.card_type_id = mct.ct_type_id and rc.card_type_id in (1,2)*  Message Result1 [SQL]select rc.ration_card_no,rc.card_type_id from ration_card rc left outer join master_card_type mct on rc.card_type_id = mct.ct_type_id and rc.card_type_id in (1,2) Time: 1.912s Affected rows: 500000 **Time: 1.912s** **Optimized Query:** *select rc.ration_card_no,rc.card_type_id from ration_card rc left outer join master_card_type mct on rc.card_type_id = mct.ct_type_id and mct.ct_type_id in (1,2) where rc.card_type_id in (1,2)*  Message Result1 [SQL]select rc.ration_card_no,rc.card_type_id from ration_card rc left outer join master_card_type mct on rc.card_type_id = mct.ct_type_id and mct.ct_type_id in (1,2) where rc.card_type_id in (1,2) Time: 0.340s Affected rows: 307685 ***Time: 0.340s***
4. WHERE			
4.1	Leading index columns in WHERE clause	Using indexed column in where clause improves performance compared to non-indexed column in where clause.	**Without Index:** *select * from ration_card where ration_card_no ='066000026059'*  Message Result1 [SQL]select * from ration_card where ration_card_no='066000026059' Time: 16.554s Affected rows: 1 **Time: 16.554s** **With Index:** *select * from ration_card where ration_card_no ='066000026059'*  Message Result1 [SQL]select * from ration_card where ration_card_no='066000026059' Time: 1.151s Affected rows: 1 **Time: 1.151s**

4.2	WHERE clause can be used in place of HAVING clause to define filters, since SQL evaluates the WHERE clause before HAVING clause.	As per the order of SQL Execution, **having** statements are calculated after **where** statements. If the intent is to filter a query based on conditions, a where statement is more efficient.	**Query:** *select fps_id,count(ration_card_no) from ration_card group by fps_id,application_type having application_type=1*  **Time: 0.699s** **Optimized Query:** *select fps_id,count(ration_card_no) from ration_card where application_type=1 group by fps_id*  **Time: 0.326s**
5. IN and EXISTS			
5.1	The EXISTS clause is much faster than IN clause, when the subquery result is very large. Conversely, the IN clause is faster than EXISTS when the subquery result is very small.	Applicable for Oracle DB only.	**Reference:** http://www.dba-oracle.com/t_exists_clause_vs_in_clause.htm
5.2	Use NOT EXISTS operator instead of NOT IN (NOT IN doesn't consider null values) to obtain accurate result.	NOT IN doesn't consider the NULL records whereas NOT EXISTS considers it.	Following queries give the different outputs. **Not In:** *select count(1) from ration_card where card_type_id not in (select ct_type_id from master_card_type)*  **Not Exists:** *select count(1) from ration_card rc where not exists (select 1 from master_card_type mct where rc.card_type_id=mct.ct_type_id)* 

6. ORDER BY and GROUP BY			
6.1	Avoid using ORDER BY on a large data set especially if the response time is important.	• SQL queries with an order by clause do not need to sort the result explicitly if the relevant index already delivers the rows in the required order. • Order by an indexed field should *not* be slow as it can pull the data in index order. • An indexed order by execution not only saves the sorting effort, however; it is also able to return the first results without processing all input data. The order by is thus executed in a pipelined manner. • An INDEX RANGE SCAN becomes inefficient for large data sets- especially when followed by a table access. This can nullify the savings from avoiding the sort operation. (Sorting is a very resource intensive operation. It needs a fair amount of CPU time, but the main problem is that the database must temporarily buffer the results.) • A FULL TABLE SCAN with an explicit sort operation might be even faster in this case. It is the optimizer's job to evaluate the different execution plans and select the best one.	**Query:** *select ration_card_no from ration_card*  **Time: 6.622s** **Optimized Query:** *select ration_card_no from ration_card order by ration_card_no*  **Time: 0.461s**
7. DELETE vs TRUNCATE			
7.1	To delete all the rows of table permanently, use truncate instead of delete.	Delete logs the operations (DML) and imposes significant time when table is large.	Delete command maintains log and it can be undone. Truncate removes the data by de-allocating the data pages used to store the table data and it cannot be undone.
8. UNION			
8.1	SET operator UNION could be avoided if the objective can be achieved through an UNION ALL.	UNION incurs an extra sort operation which can be avoided.	

9. DATATYPES			
9.1	***Char vs. Varchar2***: Prefer char to varchar2 if column width is less than 5.	This prevents surplus overhead of adjustments when data is changed.	
9.2	***Varchar vs. Varchar2***: Varchar is prone to changes in future, so it is advisable to prefer varchar2 over varchar.		
9.3	Do not mix data types	Do not use quotes if a WHERE clause column predicate is numeric. Similarly use quotes for char index columns. This reduces the rate of typecasting.	
10. MISCELLANEOUS			
10.1	Avoid LIKE predicate: Always replace "like" with an equality, when appropriate.	• The LIKE operator "implements a pattern match comparison" that attempts to match "a string value against a pattern string containing wild-card characters." • LIKE is generally used only with strings and 'equals' is faster. The 'equals' operator treats wild-card characters as literal characters. • If you search for an exact match, you can use both, = and LIKE. Using "=" is a tiny bit faster (searching for an exact match). • Search "LIKE Mill%" can still be quite fast if index is present in that particular table. Searching "LIKE %expression%" is horribly slow.	**Query:** *select * from ration_card where ration_card_no like '066000025785'*  **Time: 15.732s** **Optimized Query:** *select * from ration_card where ration_card_no='066000025785'*  **Time: 1.343s**
10.2	Use Identical Query statements	Identical queries are parsed once, whereas non-identical statements are parsed each time on their arrival.	**Different data values:** *select a from t where c=1;* *select a from t where c=2;* **Uneven spacing:** *select a from t where c=1;* *select a from t where c = 1;* **Discrepancy in case of letters: (in case of case sensitive DBs)** *select a from t where c=1;* *select a FROM t WHERE c=1;*

10.3	Always use table aliases when referencing columns	• Alias removes ambiguity when multiple tables are used in the query. • The SQL parsing engine looks at the aliases, and uses it to help remove ambiguities in symbol lookups.	*Select rc.card_type_id from ration_card* ***rc*** *join master_card_type* ***mct*** *on* ***rc****.card_type_id=****mct****.ct_type_id*
10.4	Rewrite complex sub-queries with temporary tables	Long and complex queries are hard to understand and optimize. Staging tables can break a complicated SQL statement into several smaller statements, and then store the result of each step in the database.	*select A.ration_card_no from* ***(select rc.ration_card_no,rc.card_type_id from ration_card rc join master_card_type mct on rc.card_type_id=mct.ct_type_id)A*** *where A.card_type_id in (1,2) and active=1*
10.5	Tracing Query Execution	Tracing the query execution provides us time elapsed for execution, cost of CPU, plan hash value and number of bytes accessed and other details which can be used for monitoring the query performance	**Oracle:** *Select x.sid,x.serial#,x.username,x.sql_id, x.sql_child_number,optimizer_mode,hash_value, address,sql_text from v$sqlarea sqlarea, v$session x where x.sql_hash_value = sqlarea.hash_value and x.sql_address= sqlarea.address and x.username is not null;*
10.6	Use decode and case	• Performing complex aggregations with the "decode" or "case" functions can minimize the number of times a table has to be selected. • In order to perform more calculations upon same rows in table, prefer CASE to multiple queries.	***select case*** *when count(total_income)>0 then sum(total_income) else 0 end as total_income from ration_card*
10.7	Use Wildcards at the End of a Phrase only	• When searching plaintext data, such as cities or names, wildcards create the widest search possible. However, the widest search is also the most inefficient search. • When a leading wildcard is used, especially in combination with an ending wildcard, the database is tasked with searching all records for a match anywhere within the selected field.	Consider this query to pull cities beginning with 'char': ***Select*** *name from beneficiaries* ***where*** *name like* ***'%char%'*** This query will pull the expected results of charles, charley and charanya. However, it will also pull unexpected results, such as clement charles, richard, and richardson. **A more efficient query would be:** ***Select*** *name from beneficiaries* ***where*** *name like* ***'char%'*** This query will pull only the expected results of charles, charley and charanya.

10.8	If a query is to be run over a large dataset, validate the query using limit statement to fetch limited records. (Some DBMS, uses 'top' in place of limit)	The limit statement returns only the number of records specified. Using a limit statement prevents taxing the production database with a large query, only to find out the query needs editing or refinement.	
10.9	Consider using materialized views. These are pre-computed tables comprising aggregated or joined data from fact and possible dimension tables.	• In a materialized view the result set is stored on disk like a base table but is computed like a view. • Materialized views are designed to improve performance when: ○ The database is large ○ Frequent queries result in repetitive aggregation and join operations on large amounts of data ○ Changes to underlying data are relatively infrequent	
10.10	Use full-table scans when needed	• Not all OLTP queries are made optimal using indexes. If the query returns a large percentage of table's rows, a full-table scan may be faster than index scan. • It depends on factors, including configuration (values for db_file_multiblock_read_count, db_block_size),query parallelism and the number of table/index blocks in the buffer cache.	
10.11	Use Stored Procedure for frequently used data and more complex queries.		
10.12	Triggers shall not be overused	Trigger chaining will drag down the performance.	
10.13	Access rows using Row id/similar identifier depending on database	Accessing row using rowid is fastest compared to many other methods.	
10.14	Avoid using functions such as RTRIM, TO_CHAR, UPPER, TRUNC with indexed columns.	It prevents the optimizer from identifying the index.	
10.15	Avoid Using nested Views.	View inside another view is an inefficient way of creating views. It is complex & consumes more time.	