**Assignment 1: Database Design and Optimisation**

<Milestone 2 Report>

Student Name: Keewoo Moon

Student Number: s3831370

## **Task 1.**

There are three main tables that are estimated to increase in size after 10 years of use are “**Ballot**”, “**VoterRegistry**”, and “**BallotPreferences**”.

Because assuming that the population grows steadily over time over a decade, the records of voters registered in the **VoterRegister** table are expected to increase simultaneously. At the same time, there will be more records on the Ballot table due to increased voters.

As a result, the **BallotPreference** table, which stores multiple preferences written on one ballot, is also expected to grow extremely. Because assuming that if you can pick eight preferences on one ballot, eight preference records are stored in the ballot preference table according to one ballot.

**(Storage strategy for VoterRegistry tables)**

For the **VoterRegistry** table with an average record size of 200 bytes (0.2 KB) while considering the current enrollment of approximately 17,259,000 Australians and an annual growth rate of approximately 1.2%, which is proportional to the population growth in Australia.

* I assume an initial table size of 3,458,000 KB (or 3.458 GB) based on the current enrollment of 17,259,000 voters at an estimated average record size of 0.2 KB (200 bytes).
* Estimated Sizes:
* Initial Table Size: 3,458,000 KB (or 3.458 GB) based on the current enrollment.
* Estimated Table Size After 10 Years:
* Initial size: 3,458,000 KB (or 3.458 GB)
* Growth per year: 17,259,000 voters \* 1.2% growth rate \* 0.2 KB/voter = 41,409 KB (or 40.41 MB)
* Estimated size after 10 years: 3,458,000 KB (initial size) + 10 years \* 41,409 KB/year = 3,900,490 KB (or 3.81 GB)

With an average record size of 200 bytes (0.2 KB), the estimated size of the VoterRegistry table after 10 years would be approximately 3,900,490 KB or 3.81 GB, assuming the specified growth rate and updated average record size.

|  |
| --- |
| CREATE TABLE VoterRegistry (VoterID VARCHAR2(20 BYTE) NOT NULL,Title VARCHAR2(20 BYTE),FirstName VARCHAR2(20 BYTE),MiddleName VARCHAR2(20 BYTE),LastName VARCHAR2(20 BYTE),Gender VARCHAR2(20 BYTE),DOB DATE,Contactemail VARCHAR2(40 BYTE),AddressID VARCHAR2(20 BYTE) NOT NULL)STORAGE (INITIAL 3458000K -- Initial size of 3,458,000 KB (or 3.458 GB)NEXT 1729000K -- Next extent size of 1,729,000 KB (or 1.729 GB)MAXSIZE UNLIMITED -- Allow the table to grow as needed); |

**(Storage strategy for Ballot tables)**

## I assume an initial table size of 5,000 KB (or 5 MB) based on the current enrollment of 17,259,000 voters and an estimated small average record size of 100 bytes (0.1 KB).

## Estimated Sizes:

## Initial Table Size: 5,000 KB (or 5 MB) based on the current enrollment.

## Estimated Table Size After 10 Years (3 times federal election):

## Initial size: 5,000 KB (or 5 MB)

## Growth per year: 17,259,000 voters \* 1.2% growth rate \* 0.1 KB/voter = 20,711 KB (or 20.21 MB)

## Estimated size after 10 years: 5,000 KB (initial size) + 10 years \* 20,711 KB/year = 205,110 KB (or 200.1 MB)

## With an estimated small average record size and an annual growth rate of 1.2%, the estimated size of the Ballot table after 10 years (3 times the federal election) would be approximately 205,110 KB or 200.1 MB, assuming the specified growth rate and average record size.

|  |
| --- |
| CREATE TABLE Ballot (BallotID VARCHAR2(20 BYTE) NOT NULL,ElectorateName VARCHAR2(100 BYTE) NOT NULL,ElectionID VARCHAR2(20 BYTE) NOT NULL)STORAGE (INITIAL 5000K -- Initial size of 5,000 KB (or 5 MB)NEXT 2500K -- Next extent size of 2,500 KB (or 2.5 MB)MAXSIZE UNLIMITED -- Allow the table to grow as needed); |

**(Storage strategy for BallotPreferences tables)**

## Assumptions:

## Current enrollment: 17,259,000 voters

## Annual growth rate: 1.2%

## Average record size: 200 bytes (0.2 KB)

## There are 8 preferences in a ballot.

## I assume an initial table size of 6,000,000 KB (or 6 GB) based on the current enrollment of 17,259,000 voters, an estimated average record size of 200 bytes (0.2 KB), and the fact that there are 8 preferences in a ballot.

## Estimated Sizes:

## Initial Table Size: 6,000,000 KB (or 6 GB) based on the current enrollment.

## Estimated Table Size After 10 Years (3 times federal election):

## Initial size: 6,000,000 KB (or 6 GB)

## Growth per year: 17,259,000 voters \* 1.2% growth rate \* 8 preferences \* 0.2 KB/voter = 49,483.52 KB (or 48.28 MB)

## Estimated size after 10 years: 6,000,000 KB (initial size) + 10 years \* 49,483.52 KB/year = 6,494,835.2 KB (or 6.33 GB)

## With an updated average record size of 200 bytes (0.2 KB), the estimated size of the BallotPreferences table after 10 years (3 times the federal election) would be approximately 6,494,835.2 KB or 6.33 GB, assuming the specified growth rate, average record size, and the fact that there are 8 preferences in a ballot.

|  |
| --- |
| CREATE TABLE BallotPreferences (  Preference VARCHAR2(50 BYTE),  BallotID VARCHAR2(20 BYTE) NOT NULL,  CandidateID VARCHAR2(20 BYTE) NOT NULL,  ElectorateName VARCHAR2(100) NOT NULL,  ElectionID VARCHAR2(20) NOT NULL  )  STORAGE (  INITIAL 6000000K -- Initial size of 6,000,000 KB (or 6 GB)  NEXT 3000000K -- Next extent size of 3,000,000 KB (or 3 GB)  MAXSIZE UNLIMITED -- Allow the table to grow as needed  ); |

## **Task 2.**

**Task2-1.**

|  |
| --- |
| -- Task 2-1  -- Before adding the indexes  SELECT Electorate, COUNT(\*) AS TotalVoters  FROM VoterRegistry  GROUP BY Electorate  ORDER BY TotalVoters DESC;  -- Index on VoterRegistry(Electorate)  CREATE INDEX idx\_electorate\_name ON VoterRegistry (Electorate);  -- After adding the indexes  SELECT Electorate, COUNT(\*) AS TotalVoters  FROM VoterRegistry  GROUP BY Electorate  ORDER BY TotalVoters DESC; |

* With the electorate name and the total number of voters in each electorate, sorted in descending order of the total number of voters.
* The indexes on **VoterRegistry** (Electorate) can help optimize this query. In this case, we should consider adding an index on the **ElectorateName** column in the **VoterRegistry** table. This index will help speed up the grouping and counting process.
* Before adding the index, the database would likely perform a full table scan of the **VoterRegistry** table when executing the query. This means it would scan the entire table to group and count the records, which can be slow for large datasets.
* After adding the index on the **ElectorateName** column, the database should be able to use the index for grouping and counting. It can perform an index scan instead of a full table scan, which is much more efficient for this type of query.
* Regarding how the index is utilized and join algorithms, the index on **ElectorateName** is utilized when performing the GROUP BY operation because it helps quickly identify the rows with the same **ElectorateName**.

**Task2-2.**

|  |
| --- |
| -- Task 2-2  -- Before adding the indexes  SELECT  E.ElectorateName,  C.CandidateName,  C.PartyCode  FROM  Electorate E  JOIN Candidate C ON E.ElectorateName = C.Electorate  JOIN ElectionEvent EE ON E.ElectorateName = EE.ElectorateName  WHERE  EE.ElectionID = '20220521'  ORDER BY  E.ElectorateName,  DBMS\_RANDOM.RANDOM;  -- Index on Candidate.Electorate  CREATE INDEX idx\_candidate\_electorate ON Candidate (Electorate);  -- Index on ElectionEvent.ElectionID  CREATE INDEX idx\_electionevent\_electionid ON ElectionEvent (ElectionID);  -- After adding the indexes  EXPLAIN PLAN FOR  SELECT  e.ElectorateName,  c.CandidateName,  c.PartyCode  FROM  Electorate E  JOIN Candidate C ON E.ElectorateName = C.Electorate  JOIN ElectionEvent EE ON e.ElectorateName = EE.ElectorateName  WHERE  EE.ElectionID = '20220521'  ORDER BY  E.ElectorateName,  DBMS\_RANDOM.RANDOM;  -- Display the execution plan  SELECT \*  FROM TABLE(DBMS\_XPLAN.DISPLAY); |

* This query joins the Electorate, Candidate, and ElectionEvent tables, filters by the specified election ID, and randomizes the candidate order within each electorate.
* To optimize this query, I created indexes on the columns involved in the joins and the filtering conditions.
  + Index on Candidate.Electorate:
    - Type: B-tree index
    - Columns: Electorate
  + Index on ElectionEvent.ElectionID:
    - Type: B-tree index
    - Columns: ElectionID
* Before and after adding the indexes, check the query execution plan using the EXPLAIN PLAN command.
* In conclusion, the initial query execution plan before adding indexes may involve full table scans and sorting operations. After adding the indexes, the execution plan should utilize these indexes for faster access.

**Task2-3.**

|  |
| --- |
| -- Task 2-3  -- Before adding the indexes  EXPLAIN PLAN FOR  SELECT VR.FirstName, VR.LastName  FROM VoterRegistry VR  WHERE VR.VoterID NOT IN (  SELECT IR.VoterID  FROM IssuanceRecord IR  WHERE IR.ElectionID = '20220521'  ) AND VR.VoterID NOT IN (  SELECT IR.VoterID  FROM IssuanceRecord IR  WHERE IR.ElectionID = '20190518'  );  -- Index on IssuanceRecord  CREATE INDEX idx\_IssuanceRecord\_ElectionID ON IssuanceRecord (ElectionID);  -- Index on VoterRegistry  CREATE INDEX idx\_VoterRegistry\_VoterID ON VoterRegistry (VoterID);  -- After adding the indexes  EXPLAIN PLAN FOR  SELECT VR.FirstName, VR.LastName  FROM VoterRegistry VR  WHERE VR.VoterID NOT IN (  SELECT IR.VoterID  FROM IssuanceRecord IR  WHERE IR.ElectionID = '20220521'  ) AND VR.VoterID NOT IN (  SELECT IR.VoterID  FROM IssuanceRecord IR  WHERE IR.ElectionID = '20190518'  ); |

* Selected the first name and last name of registered voters from the VoterRegistry table.
* Used two subqueries with the NOT IN clause to filter out voters who did not vote in the specified elections (2022 and 2019).

|  |
| --- |
| -- Index on IssuanceRecord  CREATE INDEX idx\_IssuanceRecord\_ElectionID ON IssuanceRecord (ElectionID); |

* Index on **IssuanceRecord** for the **ElectionID** column: To improve the performance of the subqueries that filter voters based on election IDs, you can create an index on the **ElectionID** column in the **IssuanceRecord** table. This index will help speed up the lookups when filtering by election ID.

|  |
| --- |
| -- Index on VoterRegistry  CREATE INDEX idx\_VoterRegistry\_VoterID ON VoterRegistry (VoterID); |

* Index on **VoterRegistry** for the **VoterID** column: Since we are using the **VoterID** column in the NOT IN clause, creating an index on this column in the **VoterRegistry** table can significantly improve the query's performance.
* Before adding the indexes, the query execution plan may involve full table scans on the **IssuanceRecord** and **VoterRegistry** tables, which can be inefficient for large datasets.
* After adding the recommended indexes, the query execution plan should involve index scans, which are much more efficient for filtering and joining. The indexes will be utilized to locate the relevant rows in the tables more quickly.

## **Task 3.**

To address the performance issues with the extremely large tables Ballot, VoterRegistry and BallotPreferences that I identified, I implemented Horizontal Partitioning strategies for each table.

**(Ballot Table Partitioning)**

For the **Ballot** table, I used range partitioning based on the **ElectionID** column, which appears to be a candidate for partitioning since it is used frequently in queries related to elections.

|  |
| --- |
| CREATE TABLE Ballot (  BallotID VARCHAR2(20 BYTE) NOT NULL,  ElectorateName VARCHAR2(100 BYTE) NOT NULL,  ElectionID VARCHAR2(20 BYTE) NOT NULL  )  PARTITION BY RANGE (ElectionID) (  PARTITION p\_election\_2022 VALUES LESS THAN ('2023'),  PARTITION p\_election\_2023 VALUES LESS THAN ('2024'),  ...  PARTITION p\_election\_future VALUES LESS THAN (MAXVALUE)  ); |

This partitioning strategy allows for efficient partition pruning when querying based on **ElectionID**. With range partitioning on the **ElectionID**, when querying for specific elections, the database engine can easily prune partitions that do not contain relevant data.

For example, if you want to retrieve all the ballots for the 2023 election, the database engine only needs to access the ‘**p\_election\_2023’** partition, reducing the amount of data to scan and significantly improving query performance.

In terms of partition joins, when performing joins with other tables based on the **ElectionID**, such as joining the Ballot table with the **BallotPreferences** table to analyse election results, the partitioning on **ElectionID** can facilitate partition-wise joins. This means that only corresponding partitions from both tables need to be joined, which is more efficient than joining the entire tables.

**(BallotPreferences Table Partitioning)**

For the **BallotPreferences** table, I used list partitioning based on the **ElectorateName** column, which seems to be a logical choice for partitioning since it's related to voter preferences within specific electorates.

|  |
| --- |
| CREATE TABLE BallotPreferences (  Preference VARCHAR2(50 BYTE),  BallotID VARCHAR2(20 BYTE) NOT NULL,  CandidateID VARCHAR2(20 BYTE) NOT NULL,  ElectorateName VARCHAR2(100) NOT NULL,  ElectionID VARCHAR2(20) NOT NULL  )  PARTITION BY LIST (ElectorateName)(  PARTITION p\_electorate\_north VALUES ('North Electorate'),  PARTITION p\_electorate\_south VALUES ('South Electorate'),  ...  PARTITION p\_electorate\_other VALUES (DEFAULT)  ); |

With list partitioning on **ElectorateName**, queries that filter by electorate can efficiently prune partitions that do not match the specified electorate.

For instance, when you want to retrieve preferences for the "North Electorate," the database engine will only scan the ‘**p\_electorate\_north’** partition, resulting in significant query performance improvements.

When performing joins (partition joins) with other tables based on **ElectorateName**, such as joining the **BallotPreferences** table with the **VoterRegistry** table to analyse voter demographics within electorates, partition-wise joins can be applied.

**(VoterRegistry Table Partitioning)**

For the **VoterRegistry** table, I used range partitioning based on the DOB (Date of Birth) column. This partitioning strategy allows for efficient retrieval of voter information based on birthdate ranges.

|  |
| --- |
| CREATE TABLE VoterRegistry (  VoterID VARCHAR2(20 BYTE) NOT NULL,  Title VARCHAR2(20 BYTE),  FirstName VARCHAR2(20 BYTE),  MiddleName VARCHAR2(20 BYTE),  LastName VARCHAR2(20 BYTE),  Gender VARCHAR2(20 BYTE),  DOB DATE,  Contactemail VARCHAR2(40 BYTE),  Electorate VARCHAR2(20 BYTE),  AddressID VARCHAR2(20 BYTE) NOT NULL  )  PARTITION BY RANGE (DOB)(  PARTITION p\_dob\_1980s VALUES LESS THAN (TO\_DATE('1990-01-01', 'YYYY-MM-DD')),  PARTITION p\_dob\_1990s VALUES LESS THAN (TO\_DATE('2000-01-01', 'YYYY-MM-DD')),  ...  PARTITION p\_dob\_future VALUES LESS THAN (MAXVALUE)  ); |

This partitioning strategy allows for efficient partition pruning when querying based on birthdate ranges. For example, if you want to retrieve voter information for people born in the 1990s, the database engine only scans the ‘**p\_dob\_1990s’** partition, minimizing the amount of data to process and improving query performance.

When joining the **VoterRegistry** table with other tables based on birthdate ranges, such as aggregating voter data by age groups, partition-wise joins can be employed. Only relevant partitions need to be joined, reducing computational overhead.

In addition, queries that involve birthdate ranges can benefit from parallelism, as each relevant partition can be processed in parallel, leading to faster query execution.

**Task 4 (Write a stored function – previouslyVoted()).**

|  |
| --- |
| -- Task 4: previouslyVoted()  CREATE OR REPLACE FUNCTION previouslyVoted(P\_ElectionID IN VARCHAR2, P\_ElectorateName IN VARCHAR2, P\_VoterID IN VARCHAR2)  RETURN BOOLEAN IS V\_VoteCount INTEGER;  BEGIN  -- Check if the voter has voted before in the given election  SELECT COUNT(\*)  INTO V\_VoteCount  FROM IssuanceRecord IR  WHERE IR.ElectionID = P\_ElectionID  AND IR.ElectorateName = P\_ElectorateName  AND IR.VoterID = P\_VoterID;  -- Return true if the vote count is greater than 0, indicating the voter has voted before  IF V\_VoteCount > 0 THEN  RETURN TRUE;  ELSE  RETURN FALSE;  END IF;  END previouslyVoted; |

## This function takes three input parameters: p\_ElectionID, p\_ElectorateName, and p\_VoterID.

## It then checks the IssuanceRecord table to see if there is a record matching the provided election ID, electorate name, and voter ID.

## If such a record exists, it returns TRUE, indicating that the voter has voted before in that election. Otherwise, it returns FALSE.

## **Task 5 (Write a stored procedure – primaryVoteCount()).**

|  |
| --- |
| CREATE OR REPLACE PROCEDURE primaryVoteCount(  p\_ElectionID VARCHAR2,  p\_ElectorateName VARCHAR2  )  AS  BEGIN  -- Delete any existing primary vote records for the given election and electorate  DELETE FROM FinalResult  WHERE ElectionID = p\_ElectionID  AND ElectorateName = p\_ElectorateName;  -- Calculate and insert primary votes for each candidate in the chosen electorate and election  INSERT INTO FinalResult (ElectionID, ElectorateName, CandidateID, FinalPrefCount)  SELECT  p\_ElectionID,  p\_ElectorateName,  BP.CandidateID,  COUNT(\*) AS PrimaryVotes  FROM BallotPreferences BP  WHERE BP.ElectionID = p\_ElectionID  AND BP.ElectorateName = p\_ElectorateName  AND BP.Preference = '1' -- Assuming '1' represents the first preference  GROUP BY BP.CandidateID;  COMMIT;  END primaryVoteCount; |

* Deletes any existing primary vote records for the specified election and electorate to ensure that the results are up-to-date.
* Calculates and inserts the primary votes (first preferences) for each candidate in the chosen electorate and election by counting the '1' preferences from the **BallotPreferences** table.
* Commits the transaction to save the changes.

## **Task 6 (Write a stored procedure - distributePreferences()).**

|  |
| --- |
| CREATE OR REPLACE PROCEDURE distributePreferences(  electionCode IN VARCHAR2,  electorateName IN VARCHAR2  ) AS  -- Declare variables for processing  roundNumber NUMBER := 1;  candidateCount NUMBER := 0;  BEGIN  -- Get the total number of candidates in this election and electorate  SELECT COUNT(DISTINCT CandidateID)  INTO candidateCount  FROM BallotPreferences  WHERE ElectionID = electionCode  AND ElectorateName = electorateName;  -- Loop until all preferences have been distributed  WHILE roundNumber <= candidateCount LOOP  -- Insert preference distribution data into PrefDistribution table  INSERT INTO PrefDistribution (PrefCount, RoundNum, ElectorateName, ElectionID, CandidateID)  SELECT  COUNT(\*),  roundNumber,  electorateName,  electionCode,  bp.CandidateID  FROM  BallotPreferences bp  WHERE  bp.ElectionID = electionCode  AND bp.ElectorateName = electorateName  AND bp.Preference = roundNumber  GROUP BY  roundNumber,  electorateName,  electionCode,  bp.CandidateID;  -- Update FinalResult table with preference counts  FOR candidateRow IN (SELECT DISTINCT CandidateID FROM BallotPreferences WHERE ElectionID = electionCode AND ElectorateName = electorateName) LOOP  UPDATE FinalResult fr  SET fr.FinalPrefCount = (  SELECT SUM(pd.PrefCount)  FROM PrefDistribution pd  WHERE  pd.RoundNum = roundNumber  AND pd.ElectorateName = electorateName  AND pd.ElectionID = electionCode  AND pd.CandidateID = candidateRow.CandidateID  )  WHERE  fr.ElectorateName = electorateName  AND fr.ElectionID = electionCode  AND fr.CandidateID = candidateRow.CandidateID;  END LOOP;  roundNumber := roundNumber + 1; -- Move to the next round  END LOOP;  END distributePreferences; |