

# Assignment 1: Database Design and Optimisation

Note:

This report contains answers to all the Milestone 2 tasks, including the relevant SQL code, as well as the output and estimated plan screenshots for reference. Also attached in this submission is the "Assignment 1 SQL.sql" file, which contains for reference all the code I used for the assignment, including the code for each task, the creation of tables and the population of tables. Please mark my assignment based on this pdf document though, the sql file is just provided for reference.

## Task 1

Below are three tables in my final schema that I identified to be extremely large and are expected to grow over time:

### 1. VoterRegistry

This table will be extremely large due to there being over 21 million people in Australia over the age of 18 registered to vote. This table will also grow with every election as new voters reach the eligible voting age of 18. Each election brings in new registered voters while retaining existing voter records. Below are the variables held in my "VoterRegistry" table for each Voter, with an estimate of the storage size for each variable. After this I calculate the current expected size of the table as well as the expected size of the table after 10 years.

#### **Record Size Calculation:**

VoterID (int) = 4 bytes  
FirstName (varchar(50)) = 50 bytes  
MiddleNames (varchar(50)) = 50 bytes  
LastName (varchar(50)) = 50 bytes  
Address (varchar(255)) = 255 bytes  
DoB (date) = 3 bytes  
Gender (char(1)) = 1 byte  
ResidentialAddress (varchar(255)) = 255 bytes  
PostalAddress (varchar(255)) = 255 bytes  
ContactPhone (varchar(15)) = 15 bytes  
ContactMobile (varchar(15)) = 15 bytes  
ContactEmail (varchar(100)) = 100 bytes  
DivisionName (varchar(100)) = 100 bytes

Total Record Size for a single voter = 4 + 50 + 50 + 50 + 255 + 3 + 1 + 255 + 255 + 15 + 15 + 100 + 100 = 1,153 bytes ≈ 1.15 KB

### Initial Size:

2024 population of eligible voters = 21,357,108  
Table size in 2024 = 21,357,108 \* 1.15 KB = 23.47 GB  
Current estimated table size in megabytes: **23470 MB**

### Size After 10 Years:

Projected voter population after 10 years (based on population growth)  $\approx$  22,445,500  
Table size in 2034 = 22,445,500 \* 1.15 KB = 24.77 GB  
Estimated table size after 10 years in megabytes: **24770 MB**

## 2. ElectionEvent

This table stores data related to each election event, including voting details for different electoral divisions. Each election event records data for every division, resulting in a large number of rows. Below are the variables held in my "ElectionEvent" table for each event, with an estimate of the storage size for each variable. After this, I calculate the current expected size of the table as well as the expected size of the table after 10 years.

### Record Size Calculation:

ElectionEventID (INT) = 4 bytes  
TotalVoters (INT) = 4 bytes  
VotesCast (INT) = 4 bytes  
VotesReject (INT) = 4 bytes  
VotesValid (INT) = 4 bytes  
ElectionSerialNo (INT) = 4 bytes  
DivisionName (NVARCHAR(100)) = 100 bytes  
TwoCandidatePrefWinnerCandidateID (INT) = 4 bytes  
WinnerTally (INT) = 4 bytes  
TwoCandidatePrefLoserCandidateID (INT) = 4 bytes  
LoserTally (INT) = 4 bytes

Total Record Size for a single election event = 4 + 4 + 4 + 4 + 4 + 4 + 100 + 4 + 4 + 4 + 4 = 50 bytes

### Initial Size:

- For each election, there are 151 divisions.
- Per election event: 151 rows
- Holding data for 2019 and 2022 elections:

- Table size in 2024 = 151 divisions \* 2 elections \* 50 bytes = 15.1 KB

Current estimated table size in megabytes: **0.0151 MB**

### Size After 10 Years:

- By 2034, assuming 4 election cycles and an increase of 1 million voters per cycle:
  - Total records = 151 divisions \* 4 elections = 604 rows
  - Table size in 2034 = 604 rows \* 50 bytes = 30.2 KB

Estimated table size after 10 years in megabytes: **0.0302 MB**

### 3. IssuanceRecord

This table will store records of ballot papers issued to voters during each election. It will grow significantly over time because each voter receives a ballot paper, and records are kept for every election. Below are the variables held in my "IssuanceRecord" table, with an estimate of the storage size for each variable. After this, I calculate the current expected size of the table as well as the size after 10 years.

### Record Size Calculation:

VoterID (NVARCHAR(50)) = 50 bytes

ElectionEventID (INT) = 4 bytes

IssueDate (DATE) = 3 bytes

Timestamp (DATE) = 3 bytes

PollingStation (NVARCHAR(100)) = 100 bytes

Total Record Size for a single issuance record = 50 + 4 + 3 + 3 + 100 = 160 bytes

### Initial Size:

For 2022 general election: 21.6 million voters

Table size in 2022 = 21,600,000 \* 160 bytes = 3.46 GB

Current estimated table size in megabytes: **3460 MB**

### Size After 10 Years:

By 2034, with 4 election cycles and a projected increase in voters:

Total records = (21.6M + 22.1M + 22.6M + 23.1M) = 89.4M

Table size in 2034 = 89.4M \* 160 bytes = 13.46 GB

Estimated table size after 10 years in megabytes: **13460MB**

## Task 2

1.

### SQL code used for task 2 part 1:

```
--1. Display electoral division name and total number of voters in descending order of the total number of voters
SELECT
    vr.DivisionName,
    COUNT(vr.VoterID) AS TotalVoters
FROM
    VoterRegistry vr
GROUP BY
    vr.DivisionName
ORDER BY
    TotalVoters DESC;
```

### Above code explanation:

The SQL query retrieves the total number of voters (**TotalVoters**) for each electoral division from the **VoterRegistry** table. It groups the results by **DivisionName**, counts the number of **VoterIDs** in each group, and orders the results in descending order by **TotalVoters**. This provides a list of electoral divisions ranked by the number of registered voters.

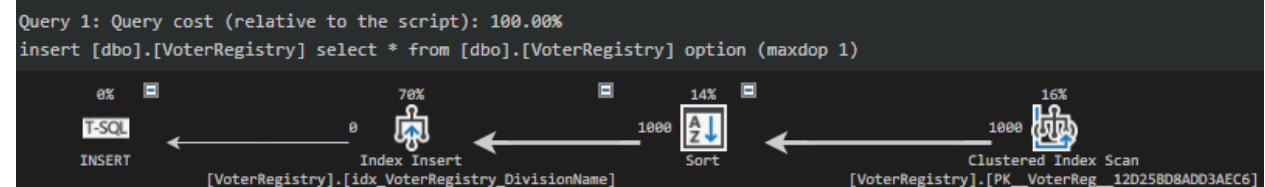
### Indexes used for task 2 part 1:

#### Index on **VoterRegistry(DivisionName)**:

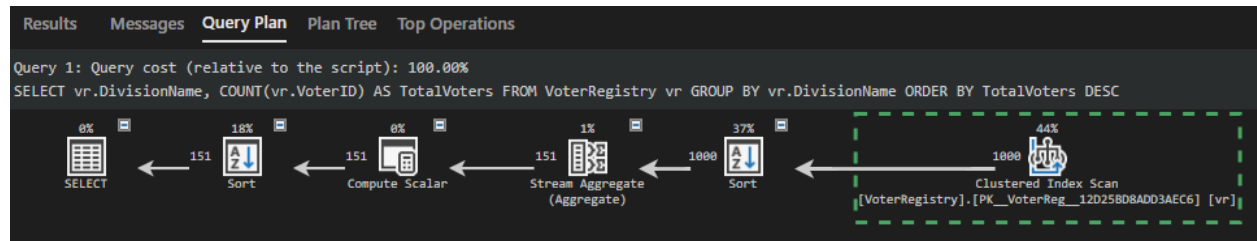
- **Type:** Non-clustered index
- **Purpose:** This index will improve query performance by speeding up the **GROUP BY** and **ORDER BY** operations that rely on the **DivisionName** column. Non-clustered indexes are ideal here as they allow fast lookups and aggregations without reordering the actual data in the table, enabling quicker access to rows with a specific **DivisionName**.

### Index SQL code:

```
--Create Index on VoterRegistry(DivisionName)
CREATE INDEX idx_VoterRegistry_DivisionName ON VoterRegistry (DivisionName);
```

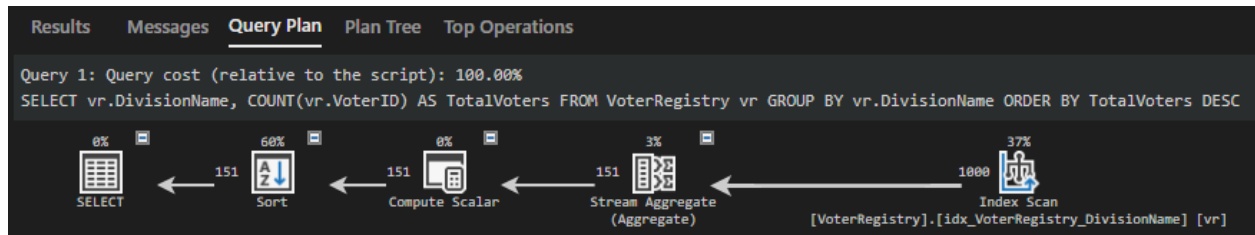


### Query execution plans before the index is added:



Clustered Index Scan	
[VoterRegistry].[PK_VoterReg_12D25BD8ADD3AEC6] [...]	
Scanning a clustered index, entirely or only a range.	
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	0.0312731
Estimated CPU Cost	0.001257
Estimated Number of Executions	1
Estimated Number of Rows Per Execution	1000
Estimated Number of Rows to be Read	1000
Estimated Number of Rows for All Executions	1000
Estimated Row Size	24 B
Ordered	False
Node ID	4
Object	
[s4002485].[dbo].[VoterRegistry].[PK_VoterReg_12D25BD8ADD3AEC6] [vr]	
Output List	
[s4002485].[dbo].[VoterRegistry].DivisionName	

Query execution plans after adding the index:



Index Scan	
[VoterRegistry].[idx_VoterRegistry_DivisionName] [...]	
Scan a nonclustered index, entirely or only a range.	
Logical Operation	Index Scan
Estimated Execution Mode	Row
Actual Execution Mode	Row
Storage	RowStore
Actual Number of Rows for All Executions	1000
Number of Rows Read	1000
Actual Number of Batches	0
Estimated I/O Cost	0.0068287
Estimated CPU Cost	0.001257
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows Per Execution	1000
Estimated Number of Rows to be Read	1000
Estimated Number of Rows for All Executions	1000
Estimated Row Size	24 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Actual Time Statistics	
Actual I/O Statistics	
Node ID	3
Object	
[s4002485].[dbo].[VoterRegistry].[idx_VoterRegistry_DivisionName] [vr]	
Output List	
[s4002485].[dbo].[VoterRegistry].DivisionName	

Explain how the index was utilised (or not) and why?

- **Before the Index was Added:**
  - **Logical Operation:** Clustered Index Scan
  - **IO Cost:** 0.031
  - **CPU Cost:** 0.001257
  - The query required scanning the entire clustered index of the **VoterRegistry** table. This operation involved reading all rows in the table, which contributed to higher I/O costs and slower performance, particularly for large tables.
- **After the Index was Added:**
  - **Logical Operation:** Index Scan on **idx\_VoterRegistry\_DivisionName**
  - **IO Cost:** 0.0068

- **CPU Cost:** 0.001257
- The introduction of the non-clustered index allowed the query to efficiently access rows based on the **DivisionName** column. This resulted in a significant reduction in I/O cost from 0.031 to 0.0068, indicating improved query performance. The CPU cost remained the same, suggesting that the index mainly benefited I/O operations.
- Therefore it seems that this index was properly utilized

## Join Algorithms

No Join algorithms were used for this task as the query did not involve joining any tables

## 2.

### SQL code used for task 2 part 2:

```
--Randomized Candidate List per Electoral Division

SELECT
    ed.DivisionName,
    c.Name AS CandidateName,
    pp.PartyName
FROM
    Contests ct
JOIN
    ElectionEvent ee ON ct.ElectionEventID = ee.ElectionEventID
JOIN
    Candidate c ON ct.CandidateID = c.CandidateID
JOIN
    ElectoralDivision ed ON ee.DivisionName = ed.DivisionName
JOIN
    PoliticalParty pp ON c.PartyCode = pp.PartyCode
WHERE
    ee.ElectionSerialNo = 20220521
ORDER BY
    ed.DivisionName, NEWID(); -- NEWID() randomizes the candidates within each
division
```

### Above code explanation:

The SQL query retrieves a randomized list of candidates for each electoral division in the 2022 federal election by joining relevant tables and ordering the results by division name and a random value generated by **NEWID()** which randomizes the order of the candidates within each electoral division.

### Indexes used for task 2 part 2:

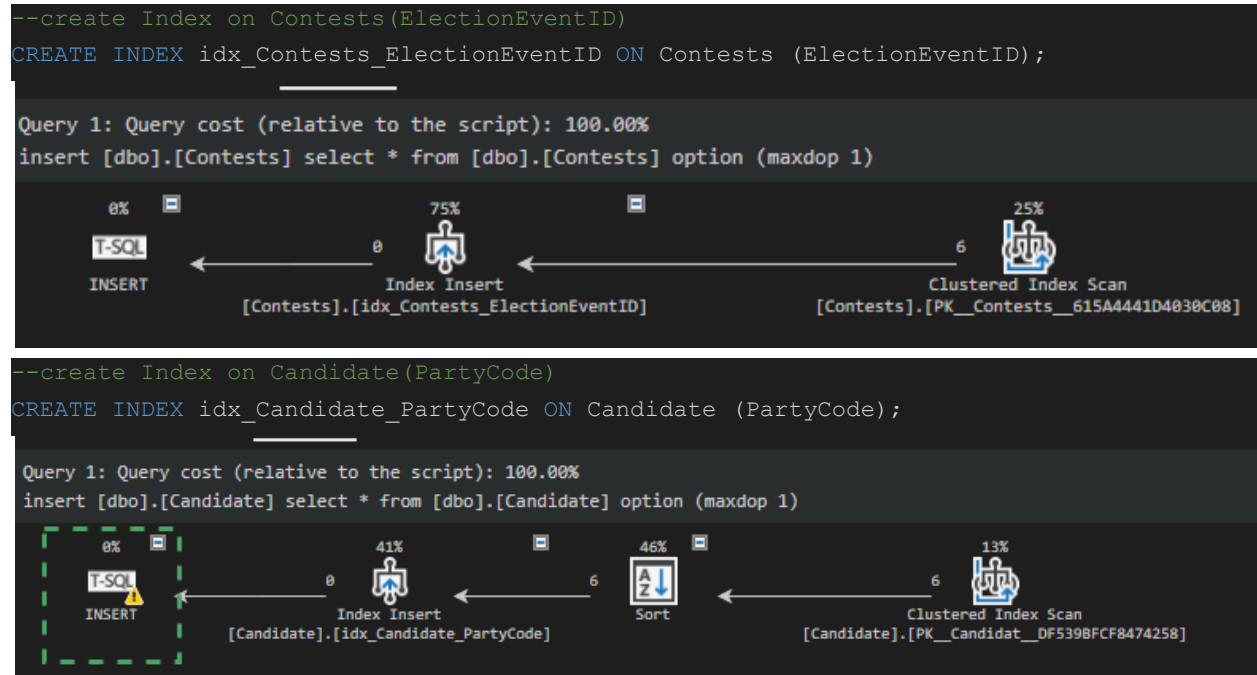
Index on **Contests(ElectionEventID)**:

- **Type:** Non-clustered index
- **Purpose:** This index will help speed up filtering operations that target a specific **ElectionEventID** in the **Contests** table. Since the query frequently filters by **ElectionEventID**, this non-clustered index allows faster retrieval without affecting the physical order of the data.

#### Index on **Candidate(PartyCode)**:

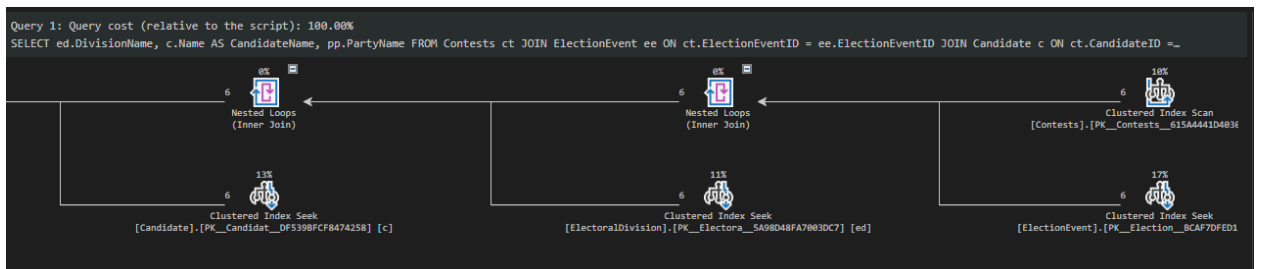
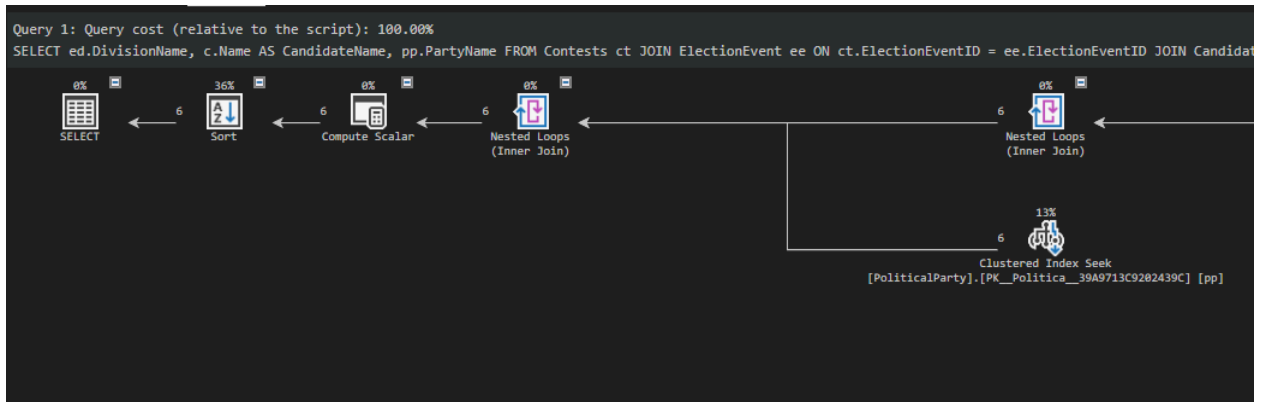
- **Type:** Non-clustered index
- **Purpose:** This index will improve the performance of join operations between the **Candidate** and **PoliticalParty** tables, particularly when **PartyCode** is used as a join condition. By indexing **PartyCode**, it speeds up lookups and join performance, enhancing query efficiency when matching candidates to their respective political parties.

#### Index SQL code:

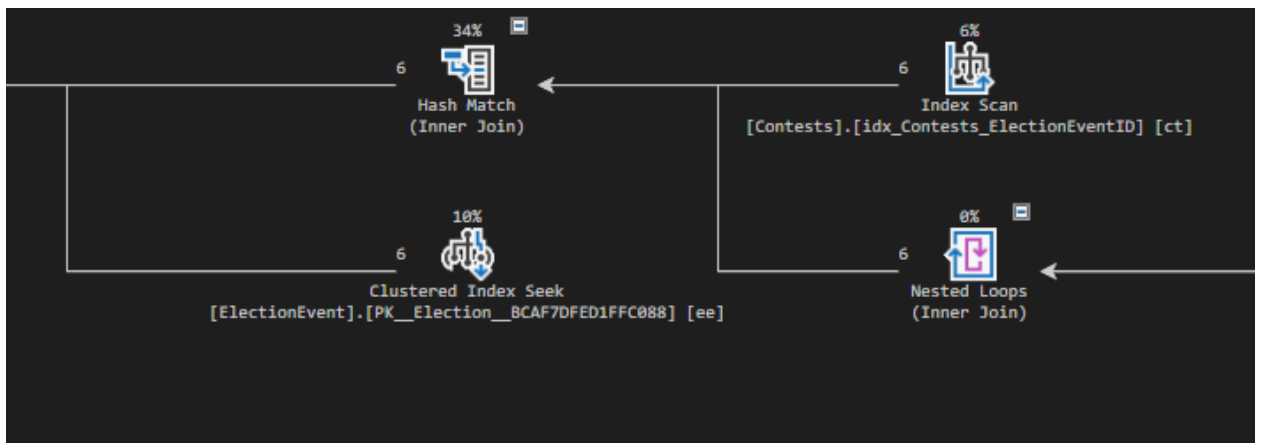
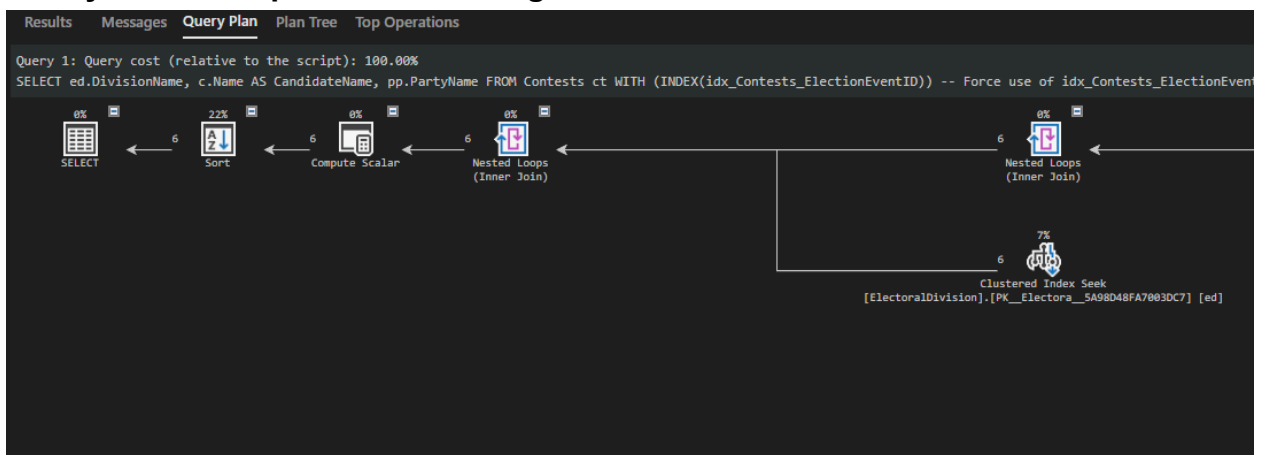


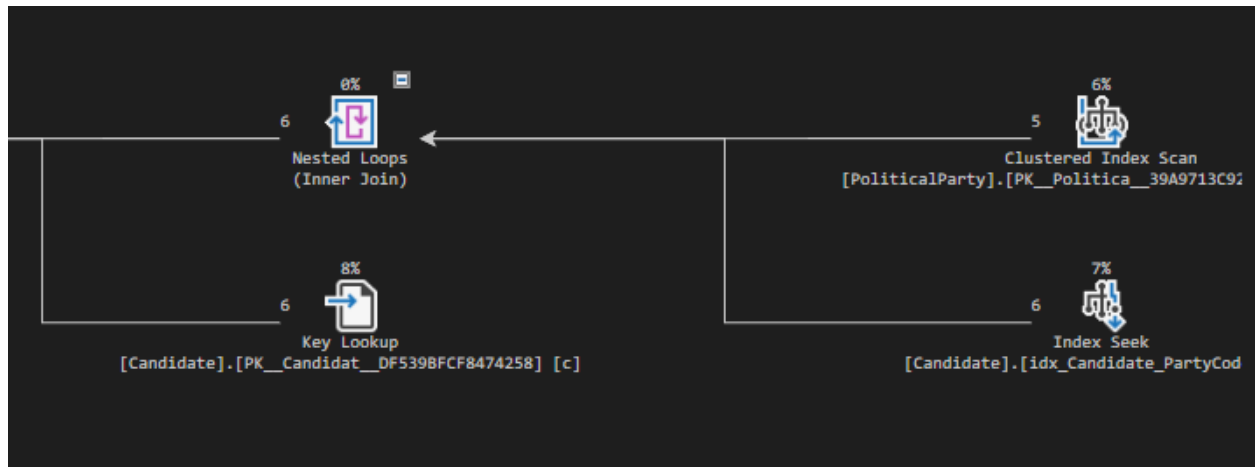
#### Query execution plans before the index is added:





## Query execution plans after adding the index:





## Explain how the index was utilised (or not) and why?

From the query plan, we can see that my 2 indexes for this task were used, with `idx_Contests_ElectionEventID` having a 6% contribution and `idx_Candidate_PartyCode` having a 7% contribution. The `idx_Contests_ElectionEventID` had an overall io cost of 0.003125 and the `idx_Candidate_PartyCode` had the same estimated io cost of 0.003125.

## Join Algorithms

For this query, multiple nested loop inner joins were used with both the index implemented and without it implemented. However, in both cases it had a 0% contribution to the overall query execution, meaning that the percentage of total cost contributed by these joins is negligible.

## 3.

## SQL code used for task 2 part 3:

```
--3
--generate a report that lists the names and addresses of registered voters who did
not vote in 2022 general election
--(election event id: 20220521) and also not voted in 2019 general election (election
event id: 20190518).
SELECT
    vr.FirstName,
    vr.LastName,
    vr.Address
FROM
    VoterRegistry vr
WHERE
    vr.VoterID NOT IN (
        SELECT ir.VoterID
        FROM IssuanceRecord ir
        JOIN ElectionEvent ee ON ir.ElectionEventID = ee.ElectionEventID
        WHERE ee.ElectionSerialNo IN (20220521, 20190518)
```

```
);
```

### Above code explanation:

This query generates a report listing the first name, last name, and address of registered voters who did not vote in both the 2022 and 2019 general elections by using a **NOT IN** subquery to exclude voters present in the **IssuanceRecord** table for those election events.

### Indexes used for task 2 part 3:

#### Index on **IssuanceRecord(VoterID, ElectionEventID)**:

- **Type:** Composite Index
- **Justification:** This will speed up the filtering by linking the voter with the election event.

#### Index on **ElectionEvent(ElectionSerialNo)**:

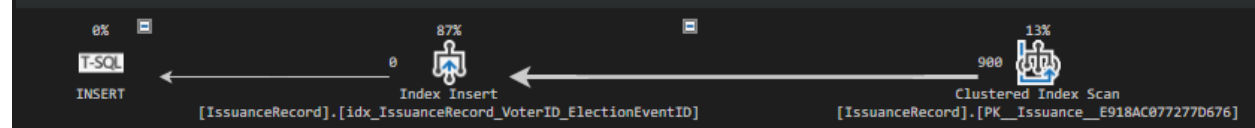
- **Type:** Simple Index
- **Justification:** Speeds up filtering election events by **ElectionSerialNo**

### Index SQL code:

```
--create index on IssuanceRecord table
CREATE INDEX idx_IssuanceRecord_VoterID_ElectionEventID
ON IssuanceRecord (VoterID, ElectionEventID);
```

Query 1: Query cost (relative to the script): 100.00%

insert [dbo].[IssuanceRecord] select \* from [dbo].[IssuanceRecord] option (maxdop 1)

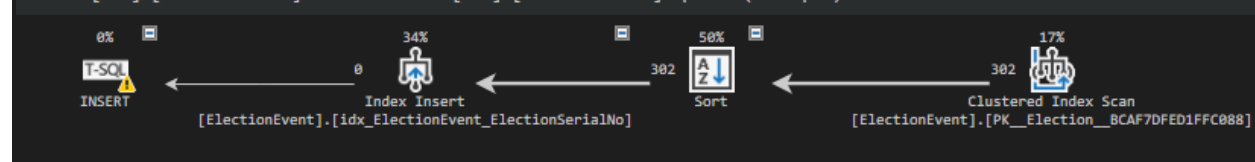


The execution plan for the IssuanceRecord insert query shows a T-SQL INSERT operation at 0% cost. It is followed by an Index Insert operation at 87% cost, which is linked to a Clustered Index Scan operation at 13% cost. The Clustered Index Scan is on the primary key [IssuanceRecord].[PK\_Issuance\_\_E918AC077277D676].

```
--create index on ElectionEvent table
CREATE INDEX idx_ElectionEvent_ElectionSerialNo
ON ElectionEvent (ElectionSerialNo);
```

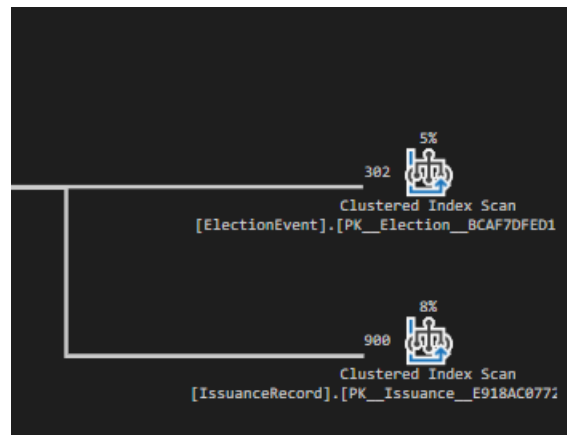
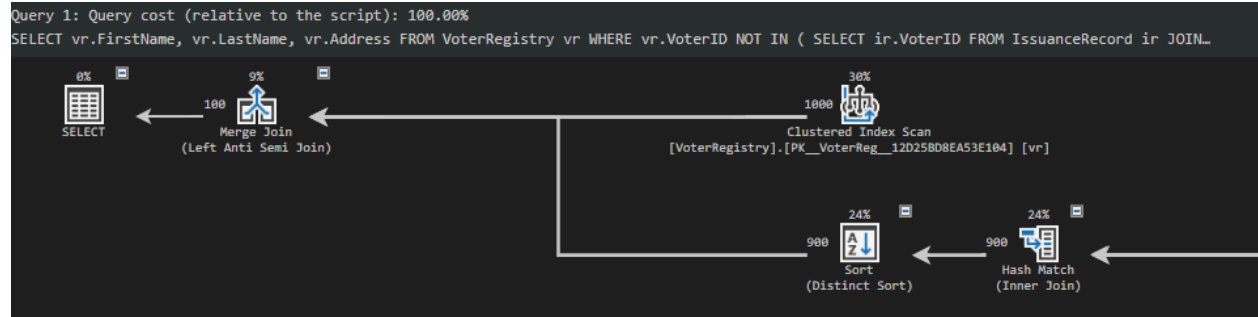
Query 1: Query cost (relative to the script): 100.00%

insert [dbo].[ElectionEvent] select \* from [dbo].[ElectionEvent] option (maxdop 1)

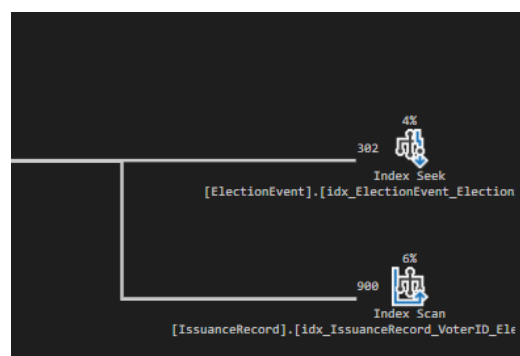
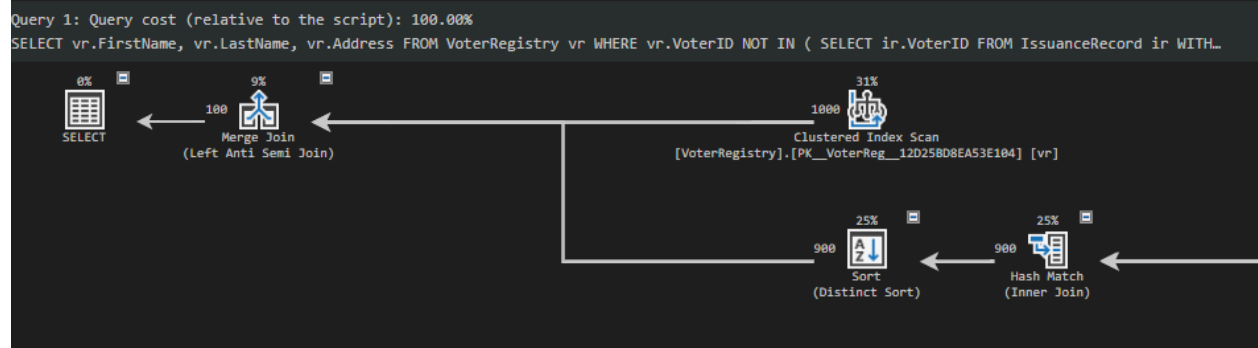


The execution plan for the ElectionEvent insert query shows a T-SQL INSERT operation at 0% cost. It is followed by an Index Insert operation at 34% cost, which is linked to a Sort operation at 50% cost, which is in turn linked to a Clustered Index Scan operation at 17% cost. The Clustered Index Scan is on the primary key [ElectionEvent].[PK\_Election\_\_BCAF7DFED1FFC088].

### Query execution plans before the index is added:



### Query execution plans after adding the index:



### Explain how the index was utilised (or not) and why?

From the query plan, we can see that my 2 indexes for this task were used, with `idx_IssuanceRecord_VoterID_ElectionEventID` having a 6% contribution and

idx\_Candidate\_PartyCode having a 4% contribution. The idx\_IssuanceRecord\_VoterID\_ElectionEventID had an overall io cost of 0.00460648 and the idx\_ElectionEvent\_ElectionSerialNo had a similar estimated io cost of 0.003125.

## Join Algorithms

Hash match join was used for this query which contributed 25% to the overall query execution, meaning that the percentage of total cost contributed by this join is somewhat high. Merge join was also used with a contribution of 9%.

## Task 3

### 1. VoterRegistry

- **Partition Column:** `DivisionName`
- **Partition Type:** Range Partitioning
- **Rationale:**
  - Since each voter is tied to an electoral division, partitioning by `DivisionName` allows us to segment the voters based on their divisions. This improves query performance, especially when aggregating voter counts per division or when retrieving data for specific divisions.
  - Partition pruning will come into play when queries specify conditions on `DivisionName`, allowing SQL Server to ignore partitions that don't meet the criteria. This is useful for queries like Task 2 (Query 1), which groups by `DivisionName` to count voters.
- **SQL DDL for Partition:**

```
--table 1, VoterRegistry

-- Create Partition Function
CREATE PARTITION FUNCTION pf_VoterRegistry (NVARCHAR(100))
AS RANGE LEFT FOR VALUES (
    'Adelaide', 'Aston', 'Ballarat', 'Banks', 'Barker', 'Barton', 'Bass', 'Bean',
    'Bendigo', 'Bennelong', 'Berowra', 'Blair', 'Blaxland',
    'Bonner', 'Boothby', 'Bowman', 'Braddon', 'Bradfield', 'Brand', 'Brisbane',
    'Bruce', 'Burt', 'Calare', 'Calwell', 'Canberra',
    'Canning', 'Capricornia', 'Casey', 'Chifley', 'Chisholm', 'Clark', 'Cook',
    'Cooper', 'Corangamite', 'Corio', 'Cowan', 'Cowper',
    'Cunningham', 'Curtin', 'Dawson', 'Deakin', 'Dickson', 'Dobell', 'Dunkley',
    'Durack', 'Eden-Monaro', 'Fadden', 'Fairfax', 'Farrer',
    'Fenner', 'Fisher', 'Flinders', 'Flynn', 'Forde', 'Forrest', 'Fowler', 'Franklin',
    'Fraser', 'Fremantle', 'Gellibrand', 'Gilmore',
```

```

    'Gippsland', 'Goldstein', 'Gorton', 'Grayndler', 'Greenway', 'Grey', 'Griffith',
    'Groom', 'Hasluck', 'Hawke', 'Herbert', 'Higgins',
    'Hindmarsh', 'Hinkler', 'Holt', 'Hotham', 'Hughes', 'Hume', 'Hunter', 'Indi',
    'Isaacs', 'Jagajaga', 'Kennedy', 'Kingsford Smith',
    'Kingston', 'Kooyong', 'La Trobe', 'Lalor', 'Leichhardt', 'Lilley', 'Lindsay',
    'Lingiari', 'Longman', 'Lyne', 'Lyons', 'Macarthur',
    'Mackellar', 'Macnamara', 'Macquarie', 'Makin', 'Mallee', 'Maranoa',
    'Maribyrnong', 'Mayo', 'McEwen', 'McMahon', 'McPherson',
    'Melbourne', 'Menzies', 'Mitchell', 'Monash', 'Moncrieff', 'Moore', 'Moreton',
    'New England', 'Newcastle', 'Nicholls', 'North Sydney',
    'O'Connor', 'Oxley', 'Page', 'Parkes', 'Parramatta', 'Paterson', 'Pearce',
    'Perth', 'Petrie', 'Rankin', 'Reid', 'Richmond', 'Riverina',
    'Robertson', 'Ryan', 'Scullin', 'Shortland', 'Solomon', 'Spence', 'Sturt', 'Swan',
    'Sydney', 'Tangney', 'Wannon', 'Warringah',
    'Watson', 'Wentworth', 'Werriwa', 'Whitlam', 'Wide Bay', 'Wills', 'Wright'
);

-- Create Partition Scheme
CREATE PARTITION SCHEME ps_VoterRegistry
AS PARTITION pf_VoterRegistry
ALL TO ([PRIMARY]);

-- Create the Partitioned Table with Partitioned Primary Key
CREATE TABLE DivisionNamePartitionedVoterRegistry (
    VoterID NVARCHAR(50),
    FirstName NVARCHAR(50),
    MiddleNames NVARCHAR(50),
    LastName NVARCHAR(50),
    Address NVARCHAR(255),
    DoB DATE,
    Gender CHAR(1),
    ResidentialAddress NVARCHAR(255),
    PostalAddress NVARCHAR(255),
    ContactPhone NVARCHAR(20),
    ContactMobile NVARCHAR(20),
    ContactEmail NVARCHAR(100),
    DivisionName NVARCHAR(100),
    PRIMARY KEY (VoterID, DivisionName), -- Including DivisionName in the primary key
    FOREIGN KEY (DivisionName) REFERENCES ElectoralDivision(DivisionName)
)
ON ps_VoterRegistry(DivisionName);

--now populate the new partitioned table with the data
INSERT INTO DivisionNamePartitionedVoterRegistry
SELECT * FROM VoterRegistry;

--view the new partitioned table

```

```
SELECT * FROM DivisionNamePartitionedVoterRegistry
```

**Which of the task 2 queries the VoterRegistry partition will improve the performance of and how it helps?**

### Partition Pruning:

Partition pruning allows SQL Server to skip scanning partitions that don't match the query criteria. Instead of scanning the entire table, the database will only focus on the relevant partitions.

#### VoterRegistry - Task 2, Query 1:

- **Query:** Counting the total number of voters per electoral division.
- **How partition pruning helps:** Since the table is partitioned by **DivisionName**, when you query for the voter count per division, SQL Server will prune the partitions that don't match the specified division. For example, if the query is counting voters in the division 'Adelaide', only the partition related to 'Adelaide' will be scanned.
  - **Result:** This drastically reduces the number of rows scanned, making aggregation operations faster.

### Parallel SQL:

Partitioning allows the query engine to process different partitions in parallel. This means that multiple partitions can be scanned or processed simultaneously, reducing overall query time.

#### VoterRegistry - Task 2, Query 1:

- **Query:** Counting voters by electoral division.
- **How parallel SQL helps:** Each partition (which represents a division) can be scanned in parallel. If you are running a query across all divisions, SQL Server can scan each division's partition concurrently, rather than sequentially.
  - **Result:** This parallel processing reduces query execution time, especially for aggregations like **COUNT**.

## 2. ElectionEvent

- **Partition Column:** **ElectionSerialNo**
- **Partition Type:** Range Partitioning
- **Rationale:**

- Partitioning the **ElectionEvent** table by **ElectionSerialNo** is ideal because each election is a distinct event, and queries typically focus on a specific election or a range of them. This strategy improves performance for queries filtering by **ElectionSerialNo**, such as Task 2 (Query 2 and 3), which involve election-specific data.
- Partition pruning helps SQL Server to scan only the necessary partitions for the elections being queried.
- **SQL DDL for Partition:**

```
--table 2, ElectionEvent

-- Create Partition Function
CREATE PARTITION FUNCTION pf_ElectionEvent (INT)
AS RANGE LEFT FOR VALUES (20190518, 20220521); -- Key elections for 2019 and 2022

-- Create Partition Scheme
CREATE PARTITION SCHEME ps_ElectionEvent
AS PARTITION pf_ElectionEvent
ALL TO ([PRIMARY]);

-- Create the Partitioned Table with Partitioned Primary Key
CREATE TABLE ElectionSerialNoPartitionedElectionEvent (
    ElectionEventID INT,
    TotalVoters INT,
    VotesCast INT,
    VotesReject INT,
    VotesValid INT,
    ElectionSerialNo INT,
    DivisionName NVARCHAR(100),
    TwoCandidatePrefWinnerCandidateID INT,
    WinnerTally INT,
    TwoCandidatePrefLoserCandidateID INT,
    LoserTally INT,
    PRIMARY KEY (ElectionSerialNo, ElectionEventID), -- Including ElectionSerialNo in
the primary key
    FOREIGN KEY (ElectionSerialNo) REFERENCES ElectionMaster(ElectionSerialNo),
    FOREIGN KEY (DivisionName) REFERENCES ElectoralDivision(DivisionName),
    FOREIGN KEY (TwoCandidatePrefWinnerCandidateID) REFERENCES Candidate(CandidateID),
    FOREIGN KEY (TwoCandidatePrefLoserCandidateID) REFERENCES Candidate(CandidateID)
)
ON ps_ElectionEvent(ElectionSerialNo);

--now populate the new partitioned table with the data
INSERT INTO ElectionSerialNoPartitionedElectionEvent
SELECT * FROM ElectionEvent;

--view the new partitioned table
```



```
SELECT * FROM ElectionSerialNoPartitionedElectionEvent
```

**Which of the task 2 queries the ElectionEvent partition will improve the performance of and how it helps?**

#### Partition Pruning:

##### ElectionEvent - Task 2, Query 2:

- **Query:** Randomizing candidate lists for the 2022 election (`ElectionSerialNo = 20220521`).
- **How partition pruning helps:** Since the table is partitioned by `ElectionSerialNo`, SQL Server will scan only the partition related to `20220521`, skipping irrelevant partitions (such as `20190518` for the 2019 election).
  - **Result:** This minimizes the scanned data to only one election, speeding up the query.

#### Parallel SQL:

##### ElectionEvent - Task 2, Query 2:

- **Query:** Randomizing candidate lists for the 2022 election.
- **How parallel SQL helps:** Since partitioning isolates each election, parallel processing can occur within the `ElectionSerialNo = 20220521` partition. If multiple divisions are queried, their corresponding partitions can also be processed in parallel.
  - **Result:** Faster retrieval and sorting of candidate lists within divisions, particularly with the randomization function.

#### Partition Joins:

##### ElectionEvent and Contests - Task 2, Query 2:

- **Query:** Randomizing candidate lists for the 2022 election.
- **How partition joins help:** Since both `ElectionEvent` and `Contests` tables can be partitioned by `ElectionSerialNo`, SQL Server can join matching partitions (e.g., partition for `20220521`), avoiding unnecessary cross-partition joins.
  - **Result:** This reduces the join complexity, as SQL Server only needs to compare records within matching partitions, leading to faster join performance.

### 3. IssuanceRecord

- **Partition Column:** `ElectionEventID`
- **Partition Type:** Range Partitioning
- **Rationale:**
  - Issuance records are linked directly to elections, with each voter receiving an issue record for a specific election. Partitioning by `ElectionEventID` helps queries that focus on specific election events, such as determining voters who didn't participate in certain elections (Task 2, Query 3). Partition pruning will allow SQL Server to ignore partitions that are unrelated to the queried elections.
- **SQL DDL for Partition:**

```
--table 3, IssuanceRecord

-- Create Partition Function

CREATE PARTITION FUNCTION pf_IssuanceRecord (INT)

AS RANGE LEFT FOR VALUES (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101,
102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118,
119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135,
136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151); --
Partition on the unique ElectionEventID values (1 to 151)

-- Create Partition Scheme

CREATE PARTITION SCHEME ps_IssuanceRecord

AS PARTITION pf_IssuanceRecord

ALL TO ('PRIMARY') ;

-- Create Partitioned Table

CREATE TABLE ElectionEventIDPartitionedIssuanceRecord (

    VoterID NVARCHAR(50),

    ElectionEventID INT,

    IssueDate DATE,
```

```

Timestamp DATE,

PollingStation NVARCHAR(100),

PRIMARY KEY (VoterID, ElectionEventID),

FOREIGN KEY (VoterID) REFERENCES VoterRegistry(VoterID),

FOREIGN KEY (ElectionEventID) REFERENCES ElectionEvent(ElectionEventID)
)

ON ps_IssuanceRecord(ElectionEventID);

--now populate the new partitioned table with the data

INSERT INTO ElectionEventIDPartitionedIssuanceRecord

SELECT * FROM IssuanceRecord;

--view the new partitioned table

SELECT * FROM ElectionEventIDPartitionedIssuanceRecord

```

**Which of the task 2 queries the IssuanceRecord partition will improve the performance of and how it helps?**

**Partition Pruning:**

**ElectionEvent - Task 2, Query 3:**

- **Query:** Checking which voters did not vote in both the 2019 and 2022 elections, using the **ElectionSerialNo**.
- **How it helps:** When **ElectionEventID** is used to join the **IssuanceRecord** table with the **ElectionEvent** table, SQL Server can prune away irrelevant partitions based on the **ElectionEventIDs** corresponding to the 2019 and 2022 elections. Only the partitions containing the necessary **ElectionEventIDs** will be scanned, minimizing the data read and speeding up the query execution.

**Partition Joins**

## IssuanceRecord and ElectionEvent - Task 2, Query 3::

- **Query:** The join between **IssuanceRecord** and **ElectionEvent** on **ElectionEventID** will only operate on the relevant partitions. This ensures that the database doesn't perform a full-table join but instead works efficiently within the partitions where the join keys are located.

## Parallel SQL:

- **How it helps:** Partitioning allows SQL Server to process multiple partitions simultaneously, which improves performance when processing election events in parallel. For instance, if multiple election events are involved, SQL Server can query the corresponding partitions simultaneously, which reduces the overall query time.
- 

## Task 4

For Task 4, I developed a stored function called **previouslyVoted()** that ensures the integrity of the election system by checking whether a voter has already voted in a specific election. This function takes three inputs—**ElectionSerialNo**, **DivisionName**, and **VoterID**—and returns a boolean value indicating whether the voter has voted before (true) or not (false). The function checks if the voter belongs to the specified division from the **VoterRegistry** and if there is a corresponding voting record for the election in the **IssuanceRecord** table, ensuring an accurate voting status for the given voter.

## SQL code for my stored function previouslyVoted():

```
--Task 4 code

--create the stored function
CREATE FUNCTION previouslyVoted (
    @ElectionSerialNo INT,
    @DivisionName NVARCHAR(100),
    @VoterID NVARCHAR(50)
)
RETURNS BIT
AS
BEGIN
    DECLARE @HasVoted BIT;

    -- First, ensure the voter belongs to the specified division in VoterRegistry
    IF EXISTS (
```

```

        SELECT 1
        FROM VoterRegistry VR
        WHERE VR.VoterID = @VoterID
        AND VR.DivisionName = @DivisionName
    )
    BEGIN
        -- If the voter belongs to the division, check if they have a record in
        IssuanceRecord for the given election serial number
        IF EXISTS (
            SELECT 1
            FROM IssuanceRecord IR
            INNER JOIN ElectionEvent EE ON IR.ElectionEventID = EE.ElectionEventID
            WHERE EE.ElectionSerialNo = @ElectionSerialNo
            AND IR.VoterID = @VoterID
        )
        BEGIN
            SET @HasVoted = 1; -- Voter has already voted
        END
        ELSE
        BEGIN
            SET @HasVoted = 0; -- Voter has not voted yet
        END
    END
    ELSE
    BEGIN
        -- Voter doesn't belong to the input division
        SET @HasVoted = 0;
    END

    RETURN @HasVoted;
END;

```

## Testing previouslyVoted() stored function:

```

--test function to see if voter with id 6 has voted in the 2022 election
SELECT dbo.previouslyVoted(20220521, 'Adelaide', '133') AS HasVoted;
--returns 1, meaning they have. I double checked this, and it is correct

--test function to see if voter with id 1000 has voted in the 2022 election
SELECT dbo.previouslyVoted(20220521, 'Werriwa', '1000') AS HasVoted;
--returns 0, meaning they have. I double checked this, and it is correct

```

## Task 5

For Task 5, I developed a stored procedure, `primaryVoteCount`, which counts the first preference votes for candidates in an election event and updates the relevant tables. The procedure takes two input parameters: `@ElectionSerialNo` and `@DivisionName`, which identify the specific election event.

### SQL code for my T-SQL stored procedure `primaryVoteCount`

```
--Task 5 code

-- Create the stored procedure

CREATE PROCEDURE primaryVoteCount

    @ElectionSerialNo INT,

    @DivisionName NVARCHAR(100)

AS

BEGIN

    -- Ensure we're working with the correct election event

    DECLARE @ElectionEventID INT;

    SET @ElectionEventID = (SELECT ElectionEventID

                            FROM ElectionEvent

                            WHERE ElectionSerialNo = @ElectionSerialNo

                            AND DivisionName = @DivisionName);

    -- Ensure the election event exists

    IF @ElectionEventID IS NULL

    BEGIN

        PRINT 'No matching election event found.';

        RETURN;

    END
```

```

-- Clear previous tally records for this election event and round 1

DELETE FROM PreferenceTallyPerRoundPerCandidate

WHERE ElectionEventID = @ElectionEventID AND RoundNo = 1;

-- Insert the first preference counts into PreferenceTallyPerRoundPerCandidate

INSERT INTO PreferenceTallyPerRoundPerCandidate (ElectionEventID, RoundNo,
CandidateID, PreferenceTally)

SELECT

    @ElectionEventID AS ElectionEventID,

    1 AS RoundNo,

    BP.CandidateID,

    COUNT(*) AS PreferenceTally

FROM BallotPreferences BP

INNER JOIN Ballot B ON BP.BallotID = B.BallotID

INNER JOIN ElectionEvent EE ON B.ElectionEventID = EE.ElectionEventID

WHERE EE.ElectionSerialNo = @ElectionSerialNo

AND EE.DivisionName = @DivisionName

AND BP.Preference = 1

GROUP BY BP.CandidateID;

-- insert/update records in PrefCountRecord and aggregate counts in a way suitable
for PrefCountRecord

-- Clear previous counts for this round

DELETE FROM PrefCountRecord

WHERE ElectionEventID = @ElectionEventID AND RoundNo = 1;

```

```

-- Insert or update the aggregate counts

INSERT INTO PrefCountRecord (ElectionEventID, RoundNo, EliminatedCandidateID,
CountStatus, PreferenceAggregate)

SELECT

    @ElectionEventID AS ElectionEventID,

    1 AS RoundNo,

    NULL AS EliminatedCandidateID,

    'Active' AS CountStatus,

    SUM(PreferenceTally) AS PreferenceAggregate

FROM PreferenceTallyPerRoundPerCandidate

WHERE ElectionEventID = @ElectionEventID

AND RoundNo = 1

GROUP BY ElectionEventID;

PRINT 'Primary vote counts updated successfully.';

END;

```

## Test the T-SQL stored procedure primaryVoteCount

```

-- Execute the stored procedure for a specific election serial number and division
name

EXEC primaryVoteCount @ElectionSerialNo = 20220521, @DivisionName = 'Deakin';

```

**View updated preference tallies in PreferenceTallyPerRoundPerCandidate table after executing the stored procedure:**

```

-- View updated preference tallies

SELECT *

```



```

FROM PreferenceTallyPerRoundPerCandidate

WHERE ElectionEventID = (SELECT ElectionEventID

                        FROM ElectionEvent

                        WHERE ElectionSerialNo = 20220521

                        AND DivisionName = 'Macarthur')

AND RoundNo = 1;

```

Results		Messages		
	ElectionEventID	RoundNo	CandidateID	PreferenceTally
1	97	1	10268	1
2	97	1	17846	1
3	97	1	20553	3

View updated count records in PrefCountRecord table after executing the stored procedure:

```

-- View updated preference count records

SELECT *

FROM PrefCountRecord

WHERE ElectionEventID = (SELECT ElectionEventID

                        FROM ElectionEvent

                        WHERE ElectionSerialNo = 20220521

                        AND DivisionName = 'Macarthur')

AND RoundNo = 1;

```

Results		Messages			
	ElectionEventID	RoundNo	EliminatedCandidateID	CountStatus	PreferenceAggregate
1	97	1	NULL	Active	5

Why do only 3 candidates appear to have a number of primary votes with the stored procedure for a specific election serial number and division name ?

In the Ballot table, the BallotID table is the primary key, linked to the BallotPreferences table

which holds the actual preferences for each Candidate (which I populated with the sample data on canvas for 1000 ballot papers cast for a fake election event for a fake electorate with 6 candidates). The Ballot table is also linked to the ElectionEvent table with ElectionEventID which tells you the DivisionName the vote in the ballot paper was cast in. I randomly generated my Ballot table, keeping all unique BallotID's from the provided sample data and adding a random ElectionEventID for each Ballot from among all the ElectionEventID's stored in my ElectionEvent table. With only 1000 ballots in the sample data provided on canvas, all Candidates did not end up with a number 1 preference in each DivisionName after random generation in the Ballot table. So it is normal to see less than 6 Candidates appear as having a number of primary votes with the stored procedure for a specific election serial number and division name.

## References

1. W3Schools.com 2024, W3schools.com, viewed 15 September 2024, <[https://www.w3schools.com/sql/sql\\_stored\\_procedures.asp](https://www.w3schools.com/sql/sql_stored_procedures.asp)>.
2. GeeksforGeeks 2019, *MySQL | Creating stored function*, GeeksforGeeks, GeeksforGeeks, viewed 15 September 2024, <<https://www.geeksforgeeks.org/mysql-creating-stored-function/#>>.
3. *SQL Partitioning: A Step-by-Step Guide for Database Optimization* | PingCAP 2024, PingCAP, PingCAP, viewed 15 September 2024, <<https://www.pingcap.com/article/sql-partition-demystified-from-concept-to-implementation/>>.
4. W3Schools.com 2024, W3schools.com, viewed 15 September 2024, <[https://www.w3schools.com/sql/sql\\_create\\_index.asp](https://www.w3schools.com/sql/sql_create_index.asp)>.