Data Definition Language (DDL) Commands

1. UPDATED WITH FOREIGN KEYs

```
CREATE TABLE countrymedal (
  RankNOC INTEGER,
  NOC VARCHAR(26) PRIMARY KEY,
  Gold INTEGER,
  Silver INTEGER,
  Bronze INTEGER
);

CREATE TABLE athlete (
  AthleteID VARCHAR(5) NOT NULL PRIMARY KEY,
  Name VARCHAR(35) NOT NULL,
  NOC VARCHAR(34) NOT NULL,
  Discipline VARCHAR(21) NOT NULL,
  FOREIGN KEY (NOC) REFERENCES countrymedal(NOC)
);

CREATE TABLE iplmatch (
   MatchID   VARCHAR(7) NOT NULL PRIMARY KEY
  ,MatchName VARCHAR(53) NOT NULL
  ,HomeTeam  VARCHAR(28) NOT NULL
  ,AwayTeam  VARCHAR(28) NOT NULL
  ,Venue     VARCHAR(69) NOT NULL
  ,City      VARCHAR(14) NOT NULL
  ,HomeTeamID INTEGER, -- Foreign key column
  ,AwayTeamID INTEGER, -- Foreign key column
  FOREIGN KEY (HomeTeamID) REFERENCES team(ID)
  FOREIGN KEY (AwayTeamID) REFERENCES team(ID)
);

CREATE TABLE team (
   ID      INTEGER  NOT NULL PRIMARY KEY
  ,Year    INTEGER  NOT NULL
  ,Teams   VARCHAR(27) NOT NULL
  ,Captain VARCHAR(19) NOT NULL
);
```

```
CREATE TABLE bowling (
   ID          INTEGER  NOT NULL PRIMARY KEY
  ,season      INTEGER  NOT NULL
  ,bowling_team VARCHAR(5) NOT NULL
  ,fullName    VARCHAR(26) NOT NULL
  ,overs       INTEGER  NOT NULL
  ,wickets     INTEGER  NOT NULL
  ,economyRate  INTEGER  NOT NULL
  ,PlayerID    INTEGER  NOT NULL, -- Added foreign key column
  FOREIGN KEY (PlayerID) REFERENCES batting_bowling_combined(PlayerID)
);

CREATE TABLE batting (
   ID            INTEGER  NOT NULL PRIMARY KEY
  ,season         INTEGER
  ,current_innings VARCHAR(5) NOT NULL
  ,fullName       VARCHAR(26) NOT NULL
  ,runs           INTEGER  NOT NULL
  ,ballsFaced     INTEGER  NOT NULL
  ,strikeRate     INTEGER  NOT NULL
  ,PlayerID       INTEGER  NOT NULL, -- Added foreign key column
  FOREIGN KEY (PlayerID) REFERENCES batting_bowling_combined(PlayerID)
);

CREATE TABLE batting_bowling_combined (
   PlayerID    INTEGER NOT NULL PRIMARY KEY
  ,FullName    VARCHAR(26) NOT NULL,
  ,TeamName    VARCHAR(5) NOT NULL,
  ,RunScore    INTEGER  NOT NULL,
  ,WicketsTaken INTEGER NOT NULL,
  ,StrikeRate   VARCHAR(11) NOT NULL,
  ,EconomyRate  VARCHAR(10) NOT NULL
);
```

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| SportsUniverse     |
| SportsUniverse2023 |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
6 rows in set (0.00 sec)

mysql> use SportsUniverse2023;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+------------------------------+
| Tables_in_SportsUniverse2023 |
+------------------------------+
| athlete                      |
| batting                      |
| bowling                      |
| countrymedal                 |
| iplmatch                     |
| team                         |
+------------------------------+
6 rows in set (0.00 sec)
```

The following screenshot shows the number of rows per table in the database

```
mysql> SELECT 'athlete' AS table_name, COUNT(*) AS row_count FROM athlete
    -> UNION ALL
    -> SELECT 'batting' AS table_name, COUNT(*) AS row_count FROM batting
    -> UNION ALL
    -> SELECT 'bowling' AS table_name, COUNT(*) AS row_count FROM bowling
    -> UNION ALL
    -> SELECT 'countrymedal' AS table_name, COUNT(*) AS row_count FROM countrymedal
    -> UNION ALL
    -> SELECT 'iplmatch' AS table_name, COUNT(*) AS row_count FROM iplmatch
    -> UNION ALL
    -> SELECT 'team' AS table_name, COUNT(*) AS row_count FROM team;
+--------------+-----------+
| table_name   | row_count |
+--------------+-----------+
| athlete      |     11085 |
| batting      |     15376 |
| bowling      |     11797 |
| countrymedal |        93 |
| iplmatch     |      1032 |
| team         |       137 |
+--------------+-----------+
6 rows in set (0.01 sec)
```

ADVANCED QUERY 1

Top 3 Teams with the Highest Total Runs - Leaderboard

```
SELECT t.Teams, SUM(c.RunScore) AS TotalRuns
FROM team t
JOIN batting_bowling_combined c ON t.Teams = c.TeamName
WHERE t.Year = 2023  -- Specify the desired season (e.g., 2023)
GROUP BY t.Teams
ORDER BY TotalRuns DESC
LIMIT 3;
```

OUTPUT:

```
mysql> SELECT t.Teams, SUM(c.RunScore) AS TotalRuns
    -> FROM team t
    -> JOIN batting_bowling_combined c ON t.Teams = c.TeamName
    -> WHERE t.Year = 2023  -- Specify the desired season (e.g., 2023)
    -> GROUP BY t.Teams
    -> ORDER BY TotalRuns DESC
    -> LIMIT 3;
+----------------------+-----------+
| Teams                | TotalRuns |
+----------------------+-----------+
| Chennai Super Kings  |   1153508 |
| Mumbai Indians       |   1137261 |
| Kolkata Knight Riders |    941362 |
+----------------------+-----------+
3 rows in set (0.01 sec)
```

*Our output is only 3 because we wanted to find the top three teams

INDEXING

WITHOUT INDEXING

```
--------------------------------------------------------------------------------+
| EXPLAIN
                                                                                |
+---------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------+
| -> Limit: 3 row(s)  (actual time=1.799..1.800 rows=3 loops=1)
    -> Sort: TotalRuns DESC, limit input to 3 row(s) per chunk  (actual time=1.799..1.799 rows=3 loops=1)
        -> Table scan on <temporary>  (actual time=1.780..1.781 rows=10 loops=1)
            -> Aggregate using temporary table  (actual time=1.779..1.779 rows=10 loops=1)
                -> Inner hash join (t.Teams = c.TeamName)  (cost=1527.56 rows=1511) (actual time=0.103..1.086 rows=867 loops=1)
                    -> Table scan on c  (cost=0.99 rows=1103) (actual time=0.011..0.725 rows=1103 loops=1)
                    -> Hash
                        -> Filter: (t.`Year` = 2023)  (cost=13.95 rows=14) (actual time=0.046..0.077 rows=10 loops=1)
                            -> Table scan on t  (cost=13.95 rows=137) (actual time=0.045..0.068 rows=137 loops=1)
 |
+---------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

CREATE INDEX idx_team_year ON team (Year, Teams);

- This index includes both the “Year” and “Teams columns which is used in the WHERE clause for filtering. By using the index, it would help the database quickly locate the relevant rows.

```
--------------------------------------------------------------------------------+
| EXPLAIN
                                                                                |
+---------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------+
| -> Limit: 3 row(s)  (actual time=6.225..6.225 rows=3 loops=1)
    -> Sort: TotalRuns DESC, limit input to 3 row(s) per chunk  (actual time=6.224..6.224 rows=3 loops=1)
        -> Table scan on <temporary>  (actual time=6.203..6.205 rows=10 loops=1)
            -> Aggregate using temporary table  (actual time=6.201..6.201 rows=10 loops=1)
                -> Nested loop inner join  (cost=503.08 rows=1145) (actual time=0.038..5.488 rows=867 loops=1)
                    -> Filter: (c.TeamName is not null)  (cost=112.80 rows=1103) (actual time=0.022..0.803 rows=1103 loops=1)
                        -> Table scan on c  (cost=112.80 rows=1103) (actual time=0.021..0.710 rows=1103 loops=1)
                    -> Filter: (t.Teams = c.TeamName)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=1103)
                        -> Covering index lookup on t using idx_team_year (Year=2023, Teams=c.TeamName)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=1103)
 |
+---------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

CREATE INDEX idx_combined_teamname_runscore ON batting_bowling_combined (TeamName, RunScore);

- This index uses the “TeamName” and “RunScore” and they are used in the JOIN and the SUM. This index might help speed up the retrieval of the batting and the bowling data.

```
| EXPLAIN
|
+----------------------------------------------------------------------------------------------------
| -> Limit: 3 row(s)  (actual time=1.459..1.459 rows=3 loops=1)
    -> Sort: TotalRuns DESC, limit input to 3 row(s) per chunk  (actual time=1.458..1.458 rows=3 loops=1)
        -> Table scan on <temporary>  (actual time=1.444..1.446 rows=10 loops=1)
            -> Aggregate using temporary table  (actual time=1.444..1.444 rows=10 loops=1)
                -> Nested loop inner join  (cost=132.18 rows=1007) (actual time=0.099..0.845 rows=867 loops=1)
                    -> Filter: (t.`Year` = 2023)  (cost=13.95 rows=14) (actual time=0.072..0.102 rows=10 loops=1)
                        -> Table scan on t  (cost=13.95 rows=137) (actual time=0.070..0.092 rows=137 loops=1)
                    -> Filter: (t.Teams = c.TeamName)  (cost=1.81 rows=74) (actual time=0.005..0.068 rows=87 loops=10)
                        -> Covering index lookup on c using idx_combined_teamname_runscore (TeamName=t.Teams)  (cost=1.81 rows=74) (actual time=0.005..0.057 rows=87 loops=10)
 |
+----------------------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

CREATE INDEX idx_combined_teamname ON batting_bowling_combined (TeamName);

- This index might also reduce the time because it uses the “TeamName” column which uses the JOIN condition to match teams between the tables. This could help optimize the JOIN operation.

```
| EXPLAIN
|
+----------------------------------------------------------------------------------------------------
| -> Limit: 3 row(s)  (actual time=2.739..2.739 rows=3 loops=1)
    -> Sort: TotalRuns DESC, limit input to 3 row(s) per chunk  (actual time=2.738..2.738 rows=3 loops=1)
        -> Table scan on <temporary>  (actual time=2.717..2.720 rows=10 loops=1)
            -> Aggregate using temporary table  (actual time=2.715..2.715 rows=10 loops=1)
                -> Nested loop inner join  (cost=217.44 rows=1007) (actual time=0.095..1.994 rows=867 loops=1)
                    -> Filter: (t.`Year` = 2023)  (cost=13.95 rows=14) (actual time=0.049..0.096 rows=10 loops=1)
                        -> Table scan on t  (cost=13.95 rows=137) (actual time=0.047..0.081 rows=137 loops=1)
                    -> Index lookup on c using idx_combined_teamname (TeamName=t.Teams), with index condition: (t.Teams = c.TeamName)  (cost=8.04 rows=74) (actual time=0.009..0.180 rows=87 loops=10)
 |
+----------------------------------------------------------------------------------------------------
1 row in set (0.01 sec)
```

2. **UPDATE - So the cost decreased from 13.95 for each of the indexes so using indexes would be useful as it comes down to (8.04). Further, time goes down too.**

ADVANCED QUERY 2

Top Batting Averages for each Team in the 2023 Season

```
SELECT
   t.Year,
   t.Teams,
   t.Captain,
   COUNT(bbc.PlayerID) AS TotalPlayers,
   SUM(bbc.RunScore) AS TotalRunsScored,
   SUM(bbc.WicketsTaken) AS TotalWicketsTaken
FROM team AS t
LEFT JOIN batting_bowling_combined AS bbc ON t.Teams = bbc.TeamName
GROUP BY t.Year, t.Teams, t.Captain
ORDER BY t.Year
LIMIT 15;
```

OUTPUT:

Year	Teams	Captain	TotalPlayers	TotalRunsScored	TotalWicketsTaken
2009	Chennai Super Kings	MS Dhoni	84	1153508	56609
2009	Deccan Chargers	Adam Gilchrist	0	NULL	NULL
2009	Delhi Daredevils	Gautam Gambhir	0	NULL	NULL
2009	Kings XI Punjab	Yuvraj Singh	106	579054	36641
2009	Kolkata Knight Riders	Sourav Ganguly	108	941362	52575
2009	Mumbai Indians	Sachin Tendulkar	113	1137261	60663
2009	Rajasthan Royals	Shane Warne	140	709451	35279
2009	Royal Challengers Bangalore	Kevin Pietersen	134	762346	35288
2010	Chennai Super Kings	MS Dhoni	84	1153508	56609
2010	Deccan Chargers	Adam Gilchrist	0	NULL	NULL
2010	Delhi Daredevils	Gautam Gambhir	0	NULL	NULL
2010	Kings XI Punjab	Kumar Sangakkara	106	579054	36641
2010	Kings XI Punjab	Mahela Jayawardene	106	579054	36641
2010	Kolkata Knight Riders	Sourav Ganguly	108	941362	52575
2010	Mumbai Indians	Sachin Tendulkar	113	1137261	60663

15 rows in set (0.04 sec)

* our output was well over 15 so we limited it

3. UPDATE BEFORE INDEXING

```
--------------------------+
| EXPLAIN
                          |
+-------------------------------------------------------------------------------------------------------------------------------
--------------------------+
| -> Limit: 15 row(s)  (actual time=16.280..16.283 rows=15 loops=1)
    -> Sort: t.`Year`, t.Teams, t.Captain, limit input to 15 row(s) per chunk  (actual time=16.278..16.280 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=15.282..15.326 rows=135 loops=1)
            -> Aggregate using temporary table  (actual time=15.280..15.280 rows=135 loops=1)
                -> Left hash join (bbc.TeamName = t.Teams)  (cost=15141.12 rows=151385) (actual time=0.697..1.670 rows=12120 loops=1)
                    -> Covering index scan on t using idx_team_year_teams_captain  (cost=13.95 rows=137) (actual time=0.035..0.082 rows=137 loops=1)
                    -> Hash
                        -> Table scan on bbc  (cost=0.82 rows=1105) (actual time=0.064..0.435 rows=1105 loops=1)
 |
+-------------------------------------------------------------------------------------------------------------------------------
--------------------------+
1 row in set (0.03 sec)
```

INDEXING

CREATE INDEX idx_teams ON team (Teams);

- This index could speed up the JOIN operation by creating a better lookup for the "Teams" column

```
-------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=35.302..35.304 rows=15 loops=1)
    -> Sort: t.`Year`, t.Teams, t.Captain, limit input to 15 row(s) per chunk  (actual time=35.301..35.302 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=35.182..35.244 rows=135 loops=1)
            -> Aggregate using temporary table  (actual time=35.180..35.180 rows=135 loops=1)
                -> Nested loop left join  (cost=2048.86 rows=10074) (actual time=0.088..18.224 rows=12100 loops=1)
                    -> Table scan on t  (cost=13.95 rows=137) (actual time=0.047..0.095 rows=137 loops=1)
                    -> Filter: (t.Teams = bbc.TeamName)  (cost=7.55 rows=74) (actual time=0.003..0.126 rows=88 loops=137)
                        -> Index lookup on bbc using idx_combined_teamname (TeamName=t.Teams)  (cost=7.55 rows=74) (actual time=0.003..0.114 rows=88 loops=137)
 |
+-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------+
1 row in set (0.04 sec)
```

CREATE INDEX idx_teamname ON batting_bowling_combined (TeamName);

- This index could help performance on the JOIN operation also by creating better lookup

```
-------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=36.364..36.367 rows=15 loops=1)
    -> Sort: t.`Year`, t.Teams, t.Captain, limit input to 15 row(s) per chunk  (actual time=36.363..36.365 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=36.248..36.302 rows=135 loops=1)
            -> Aggregate using temporary table  (actual time=36.245..36.245 rows=135 loops=1)
                -> Nested loop left join  (cost=2048.86 rows=10074) (actual time=0.080..18.965 rows=12100 loops=1)
                    -> Table scan on t  (cost=13.95 rows=137) (actual time=0.046..0.106 rows=137 loops=1)
                    -> Filter: (t.Teams = bbc.TeamName)  (cost=7.55 rows=74) (actual time=0.003..0.130 rows=88 loops=137)
                        -> Index lookup on bbc using idx_combined_teamname (TeamName=t.Teams)  (cost=7.55 rows=74) (actual time=0.003..0.118 rows=88 loops=137)
 |
+-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------+
1 row in set (0.04 sec)
```

CREATE INDEX idx_team_year_teams_captain ON team (Year, Teams, Captain);

- This index could improve the GROUP BY and the ORDER BY operations by grouping and sorting faster

```
-------------------------------------------------------------------------------------------------------------------------------
| -> Limit: 15 row(s)  (cost=1248.29 rows=15) (actual time=0.323..2.616 rows=15 loops=1)
    -> Group aggregate: count(bbc.PlayerID), sum(bbc.RunScore), sum(bbc.WicketsTaken)  (cost=1248.29 rows=1103) (actual time=0.322..2.614 rows=15 loops=1)
        -> Nested loop left join  (cost=1137.99 rows=1103) (actual time=0.068..2.196 rows=1207 loops=1)
            -> Covering index scan on t using idx_team_year_teams_captain  (cost=0.19 rows=15) (actual time=0.027..0.032 rows=16 loops=1)
            -> Filter: (t.Teams = bbc.TeamName)  (cost=7.55 rows=74) (actual time=0.006..0.130 rows=75 loops=16)
                -> Index lookup on bbc using idx_combined_teamname (TeamName=t.Teams)  (cost=7.55 rows=74) (actual time=0.005..0.119 rows=75 loops=16)
 |
+-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

Results

Looking at the indexing results above, it seems like there is not that much of a timing difference between the original time and the indexing times. The biggest time difference in the first query was 0.01 and the second query was 0.04. This difference is not significant enough. This is probably because our queries are simpler than most and do not take a lot of time in general. Additionally, a lot of what we are finding are primary keys with maybe some foreign and adding an index would not do much for these queries. Furthermore, we do not have heaps of data that it needs more time. Thus, we have decided not to use indexing for these queries.

4. **UPDATE -** So the timing difference doesn't make any difference, but the cost reduces as we add indexes as clearly seen from the Screenshots above, (Before Indexing goes from 13.95 to 7.55).