ISM 6218.003F23 -- Advanced Database Management Systems

University of South Florida

Prof. Don Berndt

Project: H1B Petitions Database

Team Members:

Kabir- U81255830

Kashyap -

Satya Adda - U13506965

Shreedevi Olekar - U77226431

Yashwanth Danda - U117804078

Topic Area	Description	Points
Database Design	This part should include a logical database design (for the relational model), using normalization to control redundancy and integrity constraints for data quality.	Default: 25 Range: 20 - 30
Query Writing	This part is another chance to write SQL queries, explore transactions, and even do some database programming for stored procedures.	Default: 25 Range: 20 - 30
Performance Tuning	In this section, you can capitalize and extend your prior experiments with indexing, optimizer modes, partitioning, parallel execution and any other techniques you want to further explore.	Default: 25 Range: 20 - 30
Data Visualization	Here you are free to explore any other topics of interest. Suggestions include DBA scripts, database security, interface design, data visualization, data mining, and NoSQL databases.	Default: 25 Range: 10 - 40

Background & Summary:

The H-1B visa program, administered by the United States Citizenship and Immigration Services (USCIS), is designed for the employment of foreign workers in specialty occupations which typically require a high degree of expertise in specialized fields such as IT, engineering, mathematics, science, and finance. The program allows U.S. employers to temporarily employ foreign professionals when qualified Americans are not available. This visa is widely used in sectors like technology and has become a critical pathway for skilled foreign professionals seeking employment in the United States.

By designing and building the H1B petitions database, we aim to analyze employer information, application outcomes, industry categories (NAICS), and geographic locations of the petitioners. It helps individuals and companies to deep dive into the approvals and rejections data to make data-driven decisions for their H1B visa applications.

Table of Contents

1. Database Design

1.1 Data Acquisition & Table Definitions:

USCIS has launched the H-1B Employer Data Hub in the year 2019, offering information on employers petitioning for H-1B workers. This initiative by USCIS allows public or applicants to search for H-1B petitioners based on various criteria and download data from as far back as FY 2009. USCIS will release cumulative quarterly data updates and annual releases on the H-1B Employer Data Hub, ensuring that the latest information is available, each fiscal year.

The data is made available in the USCIS website in the flat file format. One csv for year is provided on the website for the download. We have created a python script to automate the download of all csv files. The script also merges these individual files into a single excel file which is later used to import the data into the database tables easily. From the preliminary analysis of the data available, we have come up with following tables/entities.

- **Cities:** With columns such as Name of the city, State and Zip Code, this table helps to store employer's geographic information in a normalized form.
- **Employers:** This table consists of unique employer details, along with employer name and location mapping as foreign key reference.
- **Applications:** This table gives us the overall information of the H1B filing status of employers. It consists of data related to initial approval, Initial Denials, Initial Denials, Continuing approvals, Continuing denials, along with foreign key references NACIS ID and Employer ID.

USCIS has used the NAICS ID to categorize each employer. The North American Industry Classification System (NAICS) is the standard used by Federal statistical agencies in classifying business establishments by type of economic activity. Hence, we have downloaded this information from the census website in the excel format.

- **NAICS:** This table stores the NAICS ID to Industrial Category mapping, which can be joined with employers or applications table to get Employers Industry Name.

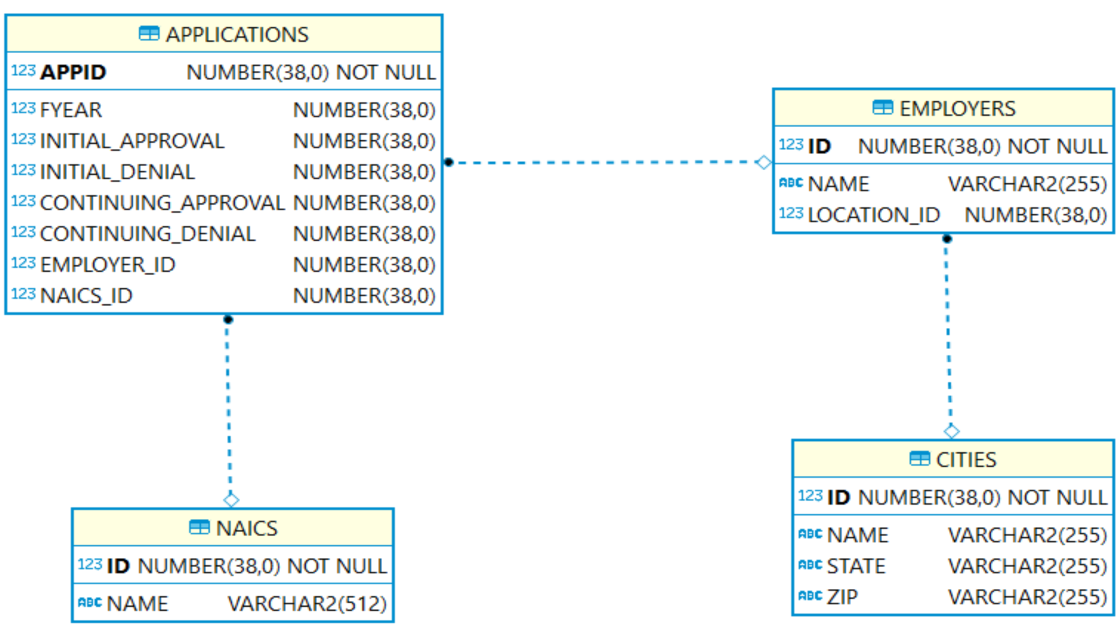
We have defined our entities as defined above to optimize the storage process and made sure that the data is normalized without any redundancies. The goal is to allow easy updates to this existing database to add new data releases as and when USCIS releases new information.

1.2 Data Integrity & Constraints:

During our data acquisition process, we made sure that there are no duplicates in the cities, employers and NAICS data by writing data quality tests in the python scripts. The script also generates a unique identifier for each of these entities. Further, we have come up with the following constraints collectively to maintain the accuracy and consistency of the data within the database.

- **Primary Keys**: Each table (APPLICATIONS, EMPLOYERS, NAICS, CITIES) has a primary key (APPID, ID, ID, ID respectively), which ensures that each record within a table is unique and identifiable.
- **Foreign Keys**: Foreign key constraints linking EMPLOYER_ID in the APPLICATIONS to ID in the EMPLOYERS, and LOCATION_ID in the EMPLOYERS to ID in the CITIES table. These ensure referential integrity, meaning an employer or city must exist before they can be referenced in an application.
- **Data Types and Not Null Constraints**: Attributes like FYEAR and NAICS_ID in APPLICATIONS is specified as numbers with a precision, and attributes like NAME in EMPLOYERS and CITIES are specified as varying character strings with a maximum length. The NOT NULL constraint on essential fields ensures that critical information must be included for a record to be created.
- **Relational Integrity**: The dashed lines indicate relationships between tables. For example, one EMPLOYER can have many APPLICATIONS, but each APPLICATION is associated with one EMPLOYER.

1.3 Entity Relationship Diagram (ERD):



2. Query Writing

2.1 DDL Queries

Data Definition Language (DDL), is a subset of SQL used for defining and managing the structure of a database. It includes commands like CREATE, DROP, and, ALTER allowing users to create, modify, or delete database objects such as tables and indexes. DDL focuses on schema and structure rather than data manipulation.

2.1.1 Create Table for cities.

CREATE TABLE Cities (id int PRIMARY KEY, name varchar (255), state varchar(255), zip varchar(255));

A new table named "Cities" is being created.

id int PRIMARY KEY: This creates an integer column named "id" that serves as the primary key for the table. The primary key uniquely identifies each record in the table and ensures its uniqueness.

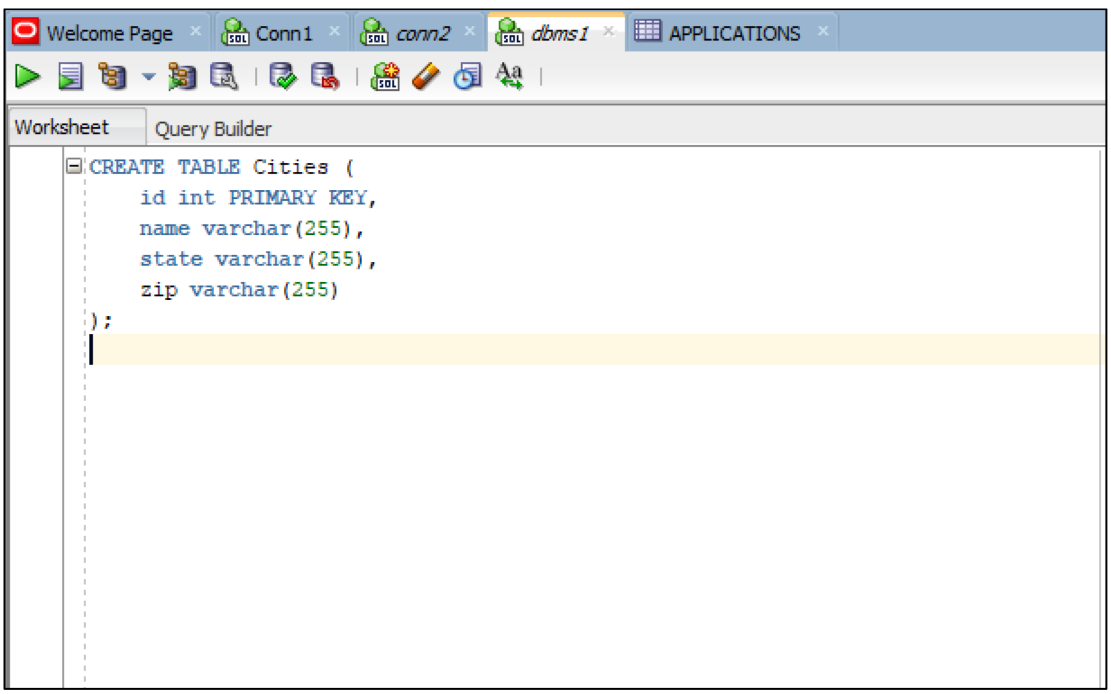


Fig 2.1.1 Table for cities

2.1.2 Create table for NAICS

CREATE TABLE NAICS (id int PRIMARY KEY, name varchar(512));

A new table named "NACIS" is being created.

id int PRIMARY KEY: This creates an integer column named "id" that serves as the primary key for the table. The primary key uniquely identifies each record in the table and ensures its uniqueness.

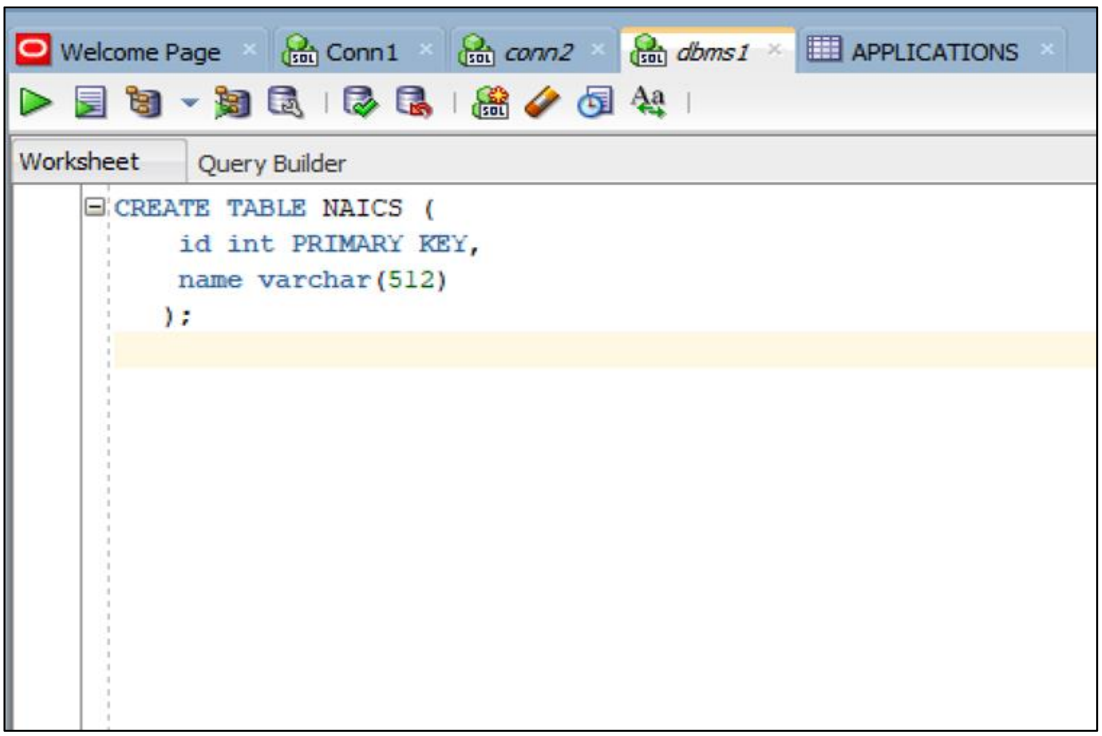


Fig 2.1.2 Table for NACIS

2.1.3 Create Table for Employers

CREATE TABLE Employers (id int PRIMARY KEY, name varchar(255), location_id int,

FOREIGN key(location_id) REFERENCES cities(id));

A new table named "Employers" is being created.

id int PRIMARY KEY: This creates an integer column named "id" that serves as the primary key for the table. The primary key uniquely identifies each record in the table.

FOREIGN KEY (location_id) REFERENCES cities(id): This establishes a foreign key constraint on the "location_id" column. It indicates that the values in the "location_id" column of the "Employers" table must correspond to values in the "id" column of the "cities" table. This enforces referential integrity between the two tables.

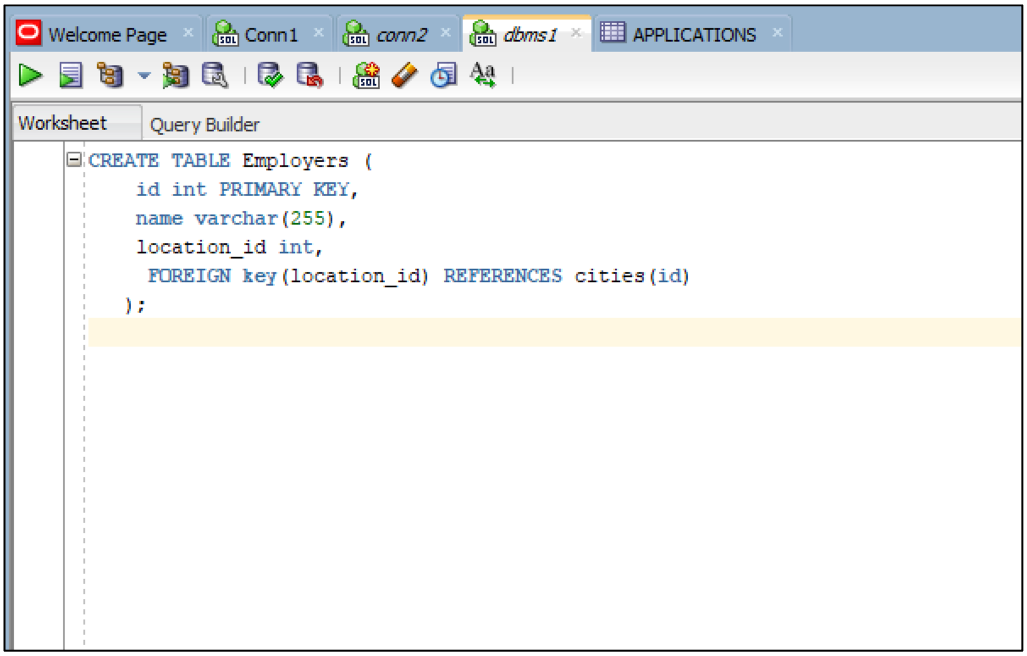


Fig 2.1.3 Table for Employers

2.1.4 Create Table for Applications

CREATE TABLE Applications (appid int PRIMARY KEY, fyear int, initial_approval int,

initial_denial int, continuing_approval int, continuing_denial int, employer_id int, naics_id int, FOREIGN key(employer_id) REFERENCES Employers(id),

FOREIGN key(naics_id) REFERENCES NAICS(id)

);

A new table named "Applications" is being created.

appid int PRIMARY KEY: This creates an integer column named "appid" that serves as the primary key for the table. The primary key uniquely identifies each record in the table.

Foreign Key Constraints:

FOREIGN KEY (employer_id) REFERENCES Employers(id): This establishes a foreign key constraint on the "employer_id" column, indicating that the values in this column must correspond to valid values in the "id" column of the "Employers" table.

FOREIGN KEY (naics_id) REFERENCES NAICS(id): This establishes a foreign key constraint on the "naics_id" column, indicating that the values in this column must correspond to valid values in the "id" column of the "NAICS" table.

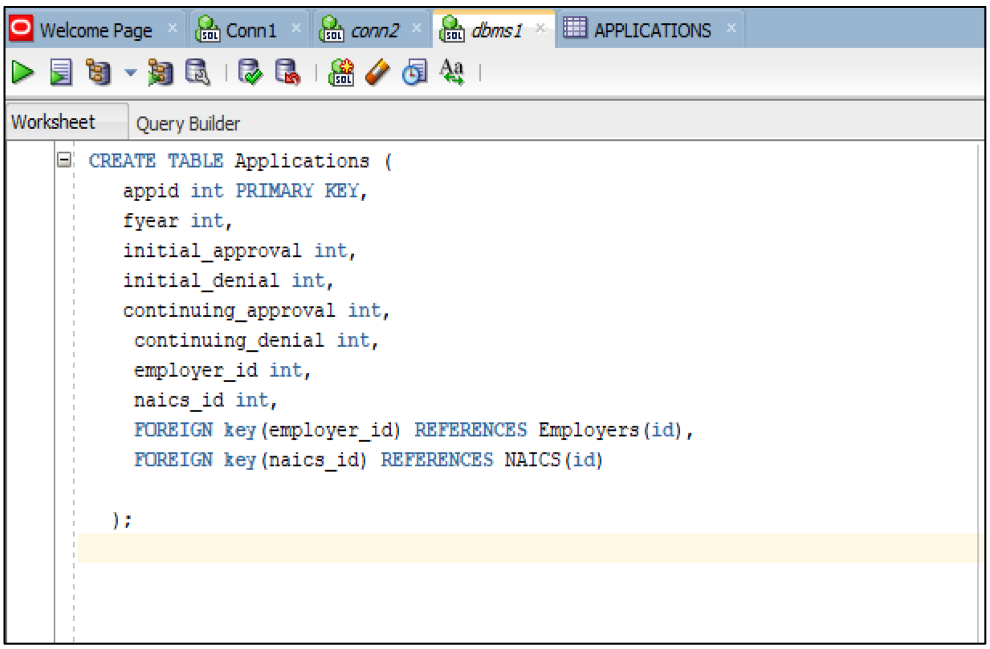


Fig 2.1.4 Table for Applications

2.2 DML Queries

DML, or Data Manipulation Language, is a subset of SQL that deals with the modification and retrieval of data in a database. It includes commands such as SELECT (retrieve data), INSERT (insert new data), UPDATE (modify existing data), and DELETE (remove data). DML is focused on interacting with the actual records and content within database tables.

2.2.1

Selecting citi id,name and zipcode from cities where state is VT

SELECT id, name,zip from cities where state= 'VT';

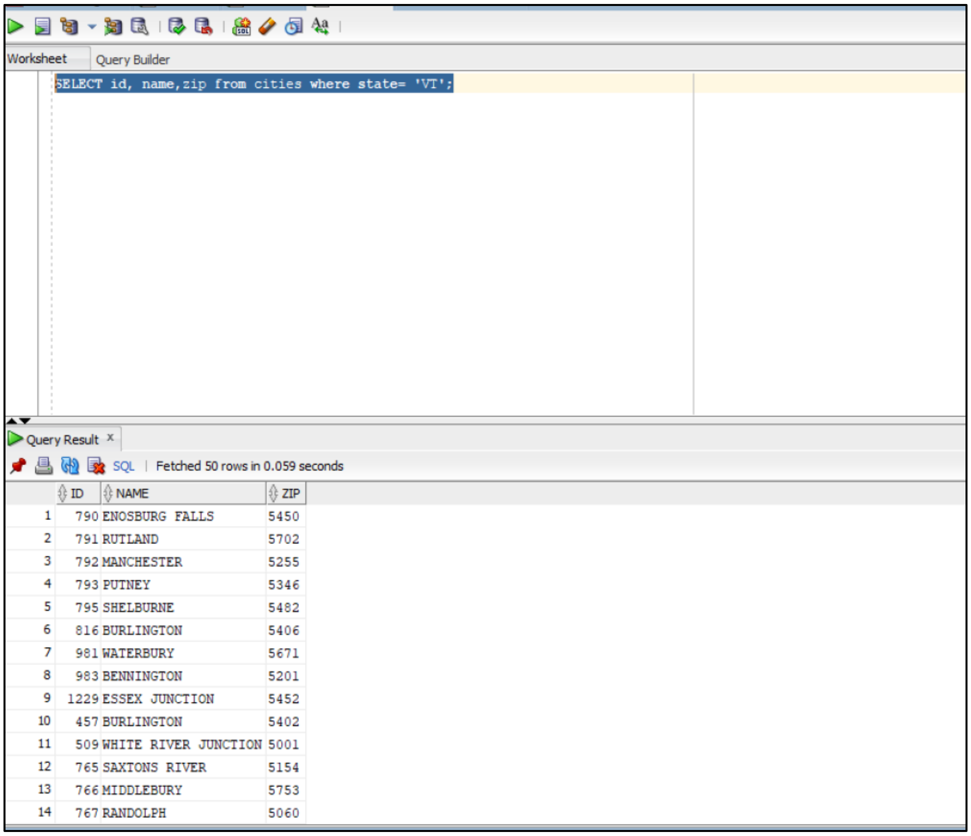


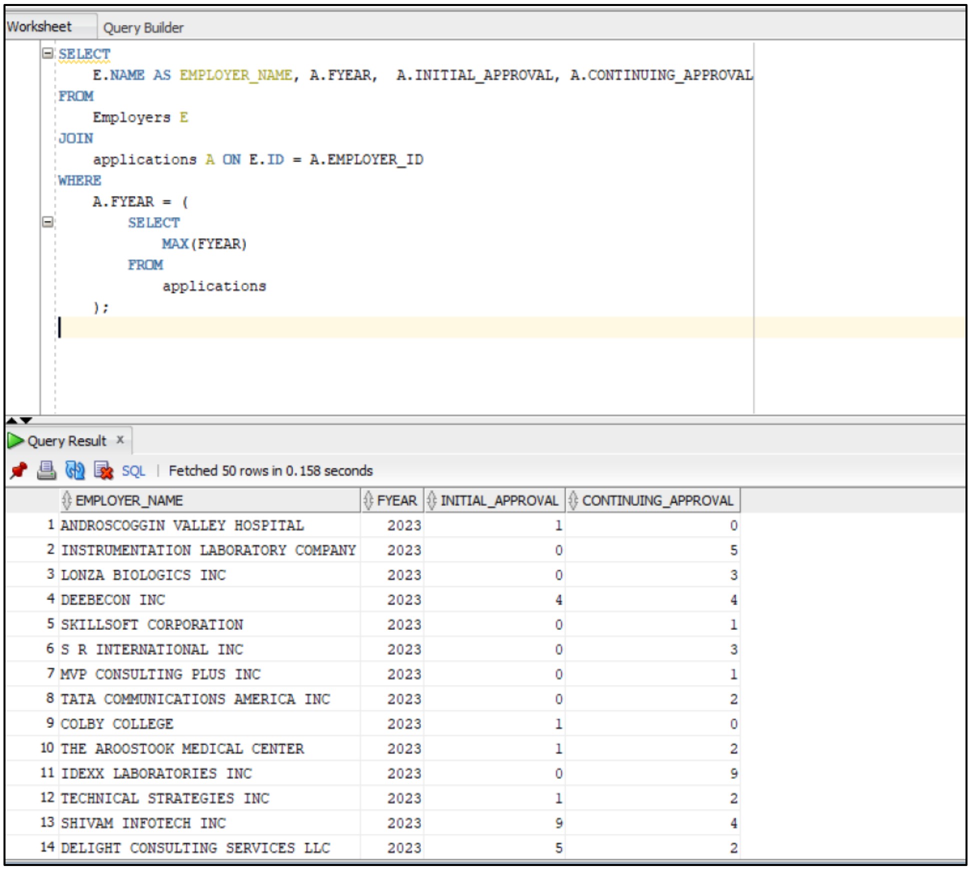
Fig 2.2.1 Select statement query

2.2.2

Select employee name, Fyear, Initial Approval, and continuing approval

```
SELECT
   E.NAME AS EMPLOYER_NAME, A.FYEAR,  A.INITIAL_APPROVAL, A.CONTINUING_APPROVAL
FROM
   Employers E
JOIN
   applications A ON E.ID = A.EMPLOYER_ID
WHERE
   A.FYEAR = (
      SELECT
         MAX(FYEAR)
      FROM
         applications
   );
```

This query retrieves the employer name, fiscal year, initial approval, and continuing approval for the latest fiscal year recorded in the "applications" table. The subquery is used to find the maximum fiscal year in the "applications" table.



```
SELECT
    E.NAME AS EMPLOYER_NAME, A.FYEAR,  A.INITIAL_APPROVAL, A.CONTINUING_APPROVAL
FROM
    Employers E
JOIN
    applications A ON E.ID = A.EMPLOYER_ID
WHERE
    A.FYEAR = (
        SELECT
            MAX(FYEAR)
        FROM
            applications
    );
```

Query Result

Fetched 50 rows in 0.158 seconds

	EMPLOYER_NAME	FYEAR	INITIAL_APPROVAL	CONTINUING_APPROVAL
1	ANDROSCOGGIN VALLEY HOSPITAL	2023	1	0
2	INSTRUMENTATION LABORATORY COMPANY	2023	0	5
3	LONZA BIOLOGICS INC	2023	0	3
4	DEEBECON INC	2023	4	4
5	SKILLSOFT CORPORATION	2023	0	1
6	S R INTERNATIONAL INC	2023	0	3
7	MVP CONSULTING PLUS INC	2023	0	1
8	TATA COMMUNICATIONS AMERICA INC	2023	0	2
9	COLBY COLLEGE	2023	1	0
10	THE AROOSTOOK MEDICAL CENTER	2023	1	2
11	IDEXX LABORATORIES INC	2023	0	9
12	TECHNICAL STRATEGIES INC	2023	1	2
13	SHIVAM INFOTECH INC	2023	9	4
14	DELIGHT CONSULTING SERVICES LLC	2023	5	2

Fig 2.2.2 Select query

2.2.3

```
SELECT
   E.NAME AS EMPLOYER_NAME,  COUNT(A.APPID) AS TOTAL_APPLICATIONS
FROM Employers E
LEFT JOIN
   applications A ON E.ID = A.EMPLOYER_ID
WHERE
   A.FYEAR = (
      SELECT
         MAX(FYEAR FROM applications )
GROUP BY
   E.NAME;
```

This query retrieves the employer name and the total number of applications submitted by each employer for the latest fiscal year recorded in the "applications" table. The subquery is used to find the maximum fiscal year.

```sql
SELECT
    E.NAME AS EMPLOYER_NAME,  COUNT(A.APPID) AS TOTAL_APPLICATIONS
FROM
    Employers E
LEFT JOIN
    applications A ON E.ID = A.EMPLOYER_ID
WHERE
    A.FYEAR = (
        SELECT
            MAX(FYEAR)
        FROM
            applications
    )
GROUP BY
    E.NAME;
```

uery Result ×

SQL | Fetched 50 rows in 0.219 seconds

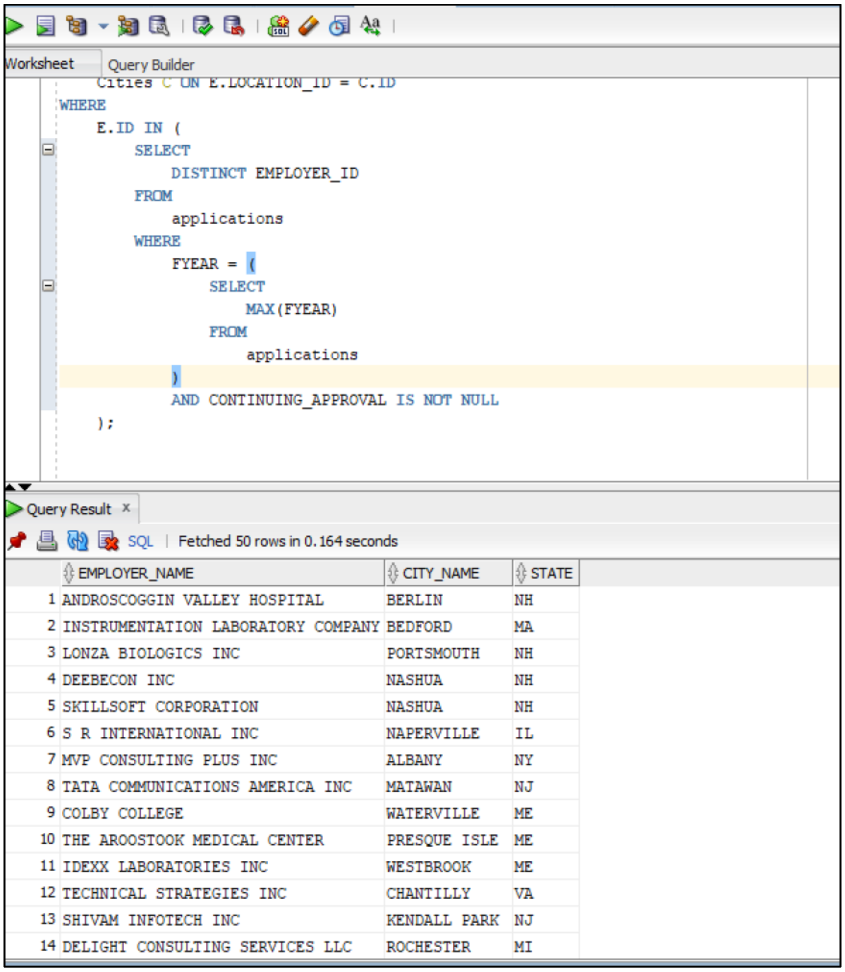
	EMPLOYER_NAME	TOTAL_APPLICATIONS
1	MVP CONSULTING PLUS INC	1
2	SMK SOFT INC	2
3	SAINT ANSELM COLLEGE	1
4	HYPERTHERM INC	1
5	BAIN & COMPANY INC	1
6	DISCO INTERNATIONAL INC	1
7	PRIMEON INC	1
8	L E K CONSULTING LLC	1
9	EAST WEST SYSTEMS INC	1
10	DELOITTE CONSULTING LLP	1
11	UNIVERSITY OF OREGON	1
12	MASSACHUSETTS MEDICAL SOCIETY	1
13	BOSTON COLLEGE	1
14	MILTON ACADEMY	1

Fig 2.2.3 Select query

2.2.4

```
SELECT
  E.NAME AS EMPLOYER_NAME,  C.NAME AS CITY_NAME, C.STATE
FROM
  Employers E
JOIN
  Cities C ON E.LOCATION_ID = C.ID
WHERE
  E.ID IN (
    SELECT
      DISTINCT EMPLOYER_ID
    FROM
      applications
    WHERE
      FYEAR = (
        SELECT
          MAX(FYEAR)
        FROM
          applications
      )
      AND CONTINUING_APPROVAL IS NOT NULL
  );
```

This query retrieves the employer name, city name, and state for employers who have applications with continuing approval in the latest fiscal year recorded in the "applications" table. The subquery is used to find the maximum fiscal year.




```sql
    Cities C ON E.LOCATION_ID = C.ID
WHERE
    E.ID IN (
        SELECT
            DISTINCT EMPLOYER_ID
        FROM
            applications
        WHERE
            FYEAR = (
                SELECT
                    MAX(FYEAR)
                FROM
                    applications
            )
            AND CONTINUING_APPROVAL IS NOT NULL
    );
```

	EMPLOYER_NAME	CITY_NAME	STATE
1	ANDROSCOGGIN VALLEY HOSPITAL	BERLIN	NH
2	INSTRUMENTATION LABORATORY COMPANY	BEDFORD	MA
3	LONZA BIOLOGICS INC	PORTSMOUTH	NH
4	DEEBECON INC	NASHUA	NH
5	SKILLSOFT CORPORATION	NASHUA	NH
6	S R INTERNATIONAL INC	NAPERVILLE	IL
7	MVP CONSULTING PLUS INC	ALBANY	NY
8	TATA COMMUNICATIONS AMERICA INC	MATAWAN	NJ
9	COLBY COLLEGE	WATERVILLE	ME
10	THE AROOSTOOK MEDICAL CENTER	PRESQUE ISLE	ME
11	IDEXX LABORATORIES INC	WESTBROOK	ME
12	TECHNICAL STRATEGIES INC	CHANTILLY	VA
13	SHIVAM INFOTECH INC	KENDALL PARK	NJ
14	DELIGHT CONSULTING SERVICES LLC	ROCHESTER	MI

Fig 2.2.4 Select Query

3. Performance Tuning

3.1 Indexing:

3.1.1 Point Query

Query Explanation & SQL Development:

This SQL query is SELECT statement that retrieves all columns (* stands for "all columns") from the table named **employers** where the **name** column matches the employer's name 'A2Z INC'. The = symbol is used for exact pattern matching in SQL.

```
SELECT * FROM employers
WHERE name = 'A2Z INC';
```

Query-Specific Indexing Plans:

This query is ordered by the employer's name. Creating an index on employer_name enhances database query speed. The index helps in allowing for quick retrieval of specific rows, particularly in SELECT queries and WHERE clause conditions involving the indexed column. This enhancement improves overall database performance.

```
create index employer_name on employers(name);
```

Assess and Compare Query performance:

Before indexing the cost is 651 after indexing cost got reduced to 5. Before Indexing the cell, physical indexing was 19038208 but after indexing it got reduced to 114688. Indexing not only increases the speed but also makes the database more efficient in retrieval and resource utilization.

Before Indexing:

SQL HotSpot | 1.36 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				651	2332	34858
TABLE ACCESS	EMPLOYERS	FULL	1	651	2332	34858
Filter Predicates						
NAME='A2Z INC'						
Other XML						

```
Statistics
-----------------------------------------------------------
                 3  DB time
                 4  Requests to/from client
          19038208  cell physical IO interconnect bytes
              2332  consistent gets
              2324  consistent gets direct
                 8  consistent gets from cache
                 8  consistent gets pin
                 8  consistent gets pin (fastpath)
                35  non-idle wait count
                 4  non-idle wait time
                 2  opened cursors cumulative
                 1  opened cursors current
                34  physical read total IO requests
          19038208  physical read total bytes
                18  physical read total multi block requests
                 1  pinned cursors current
              2332  session logical reads
                 3  user I/O wait time
                 5  user calls
```

After Indexing:

SQL HotSpot | 1.372 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				5	5	48
TABLE ACCESS	EMPLOYERS_IX	BY INDEX RO...	1	5	5	48
INDEX	EMPLOYER_NAME	RANGE SCAN	1	3	3	29
Access Predicates						
NAME='A2Z INC'						
Other XML						

```
Statistics
----------------------------------------------------------
                1  CPU used by this session
                1  CPU used when call started
                2  DB time
                5  Requests to/from client
           114688  cell physical IO interconnect bytes
               25  consistent gets
                3  consistent gets examination
                3  consistent gets examination (fastpath)
               25  consistent gets from cache
               22  consistent gets pin
               22  consistent gets pin (fastpath)
                1  enqueue releases
                1  enqueue requests
               11  non-idle wait count
                4  opened cursors cumulative
                1  opened cursors current
                7  physical read total IO requests
           114688  physical read total bytes
                1  pinned cursors current
                7  recursive calls
               25  session logical reads
                5  user calls
```

3.1.2 Range Query

Query Explanation & SQL Development:

```sql
SELECT *
FROM applications
WHERE fyear = 2020 AND initial_approval > 500;
```

This SQL query retrieves all columns (*) from the **applications** table where the initial visa approvals value in greater than 500 and the fiscial year (**fyear)** equals to 2020. The **applications** table has columns like **appid**, **fyear**,**initial_approval**, **initial_denial**, **continuing_approval**, **continuing_denial**, **employer_id**(foreign key), and naics_id.

Query-Specific Indexing Plans:

When we conduct sorting operations on the fyear column, the index can help us out. This query contains aggregations or computations based on the fyear column (such as computing the total or average of values for a certain year), which the index can speed up. Indexing eliminates the need for Full Table Scans. Indexing can improve the efficiency of operations involving the fyear column when it is used as part of a foreign key connection (e.g., linking to another table).

```sql
create index app_idx on applications_ix(fyear, initial_approval);
```

Assess and Compare Query performance:

Here before indexing the cost for running the query is 961 post that it is got reduced to 5. Earlier session logical reads were 3527 but now it has been reduced to 22. Lower logical reads indicate that the database engine is scanning fewer data pages to fulfill the query, resulting in more efficient and faster query execution.

Before Indexing:

SQL HotSpot | 1.308 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				961	3527	18420
TABLE ACCESS	APPLICATIONS	FULL	20	961	3527	18420
Filter Predicates						
AND						
INITIAL_APPROVAL>500						
FYEAR=2020						

```
Statistics
----------------------------------------------------------
          2  CPU used by this session
          2  CPU used when call started
          2  DB time
          4  Requests to/from client
       3527  consistent gets
       3527  consistent gets from cache
       3527  consistent gets pin
       3527  consistent gets pin (fastpath)
          4  non-idle wait count
          2  opened cursors cumulative
          1  opened cursors current
          1  pinned cursors current
       3527  session logical reads
          5  user calls
```

After Indexing:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				5	22	81
TABLE ACCESS	APPLICATIONS_IX	BY INDEX RO...	22	5	22	81
INDEX	APP_IDX	RANGE SCAN	20	3	3	13
Access Predicates						
AND						
FYEAR=2020						
INITIAL_APPROVAL>500						
INITIAL_APPROVAL IS NOT NULL						

```
Statistics
----------------------------------------------------------
          5  Requests to/from client
         22  consistent gets
          2  consistent gets examination
          2  consistent gets examination (fastpath)
         22  consistent gets from cache
         20  consistent gets pin
         20  consistent gets pin (fastpath)
          4  non-idle wait count
          2  opened cursors cumulative
          1  opened cursors current
          1  pinned cursors current
         22  session logical reads
          5  user calls
```

3.1.3 Scan Query

Query Explanation & SQL Development:

```sql
SELECT e.name, sum(a.initial_approval) AS new_approvals
FROM applications a
join employers e on a.employer_id = e.id
join cities c on e.location_id = c.id
WHERE fyear = 2020
and c.name = 'AUSTIN'
WHERE fyear = 2020
GROUP BY e.name
Having sum(a.initial_approval) > 0
ORDER BY 2 desc;
```

This SQL query retrieves the names of employers (**e.name**) and the total initial approvals (**sum(a.initial_approval AS new_approvals**)for applications in the year 2020, grouped by employer. It only includes results where the sum of initial approvals is greater than 0. The data is joined from the **applications** table (a) to the **employers** table (**e**) and the **cities** table (c). The results are ordered by the second column (**ORDER BY 2)** in descending order. Note: The **WHERE** clause seems incomplete as it lacks a specific condition after the **=** sign.

Query-Specific Indexing Plans:

```
create index city_name on cities(name);
```

1. Filtering on Indexed Column (c.name = 'Austin'): The query filters results based on the city name (c.name). When an index exists on the name column in the cities table, the database engine can quickly locate the rows corresponding to the specified city name ('Austin') without having to perform a full table scan. This leads to faster retrieval of relevant data.
2. Join Condition (ON e.city_id = c.id): The index on cities(name) might indirectly benefit the join operation between the employers and cities tables. Efficient access to the city information can optimize the join process, especially when dealing with large datasets.
3. Aggregation and Grouping (GROUP BY e.name): If the cities table is involved in a join and there's grouping based on the employer's name, the index can contribute to quicker aggregation. Grouping operations may be more efficient when an index exists on the columns used in the grouping.
4. Ordering (ORDER BY 2 desc): The index might be beneficial for the ordering operation as well. If the ordering is based on the second column (result of SUM(a.initial_approval)), having an index on the involved columns can facilitate a more efficient sorting process.

Assess and Compare Query performance:

Before indexing the cost is 1644, after indexing cost has been reduced to 1034. The logical reeds have been reduced from 5964 to 2764.

Before Indexing:

SQL HotSpot | 1.806 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				1644	5966	80047
SORT		ORDER BY	1	1644	5966	80047
FILTER					5966	79958
Filter Predicates						
SUM(A.INITIAL_APPROVAL)>0						
HASH		GROUP BY	1	1644	5966	79872
HASH JOIN			6	1642	5966	79286
Access Predicates						
A.EMPLOYER_ID=E.ID						
HASH JOIN			44	682	2439	58039
Access Predicates						
E.LOCATION_ID=C.ID						
TABLE A	CITIES	FULL	3	30	107	898
Filter Predicates						
C.NAME='AUSTIN'						
TABLE A	EMPLOYERS	FULL	428605	651	2332	42174
TABLE ACC	APPLICATIONS	FULL	53829	960	3527	17320
Filter Predicates						
A.FYEAR=2020						

```
Statistics
-----------------------------------------------------------
              3  CPU used by this session
              3  CPU used when call started
              6  DB time
              5  Requests to/from client
       19038208  cell physical IO interconnect bytes
           5966  consistent gets
           2324  consistent gets direct
           3642  consistent gets from cache
           3642  consistent gets pin
           3642  consistent gets pin (fastpath)
             28  non-idle wait count
              2  non-idle wait time
              2  opened cursors cumulative
              1  opened cursors current
             34  physical read total IO requests
       19038208  physical read total bytes
             18  physical read total multi block requests
              1  pinned cursors current
           5966  session logical reads
              3  user I/O wait time
              6  user calls

214 rows selected.
```

After Indexing:

SQL HotSpot | 1.465 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				1036	2764	81680
SORT		ORDER BY	6	1036	2764	81680
FILTER					2764	81599
Filter Predicates						
HASH		GROUP BY	6	1036	2764	81526
HASH JOIN			6	1034	2764	80936
Access Predicates						
NESTED LO			6	1034	2396	30283
NESTED					2396	29547
STA					2396	28839
			44	656	2396	28018
Access Predicates						
E.LOCATION_ID=C.ID						
	CITIES_IX	BY INDEX RO...	3	4	49	240
	CITY_NAME	RANGE SCAN	3	1	3	48
Access Predicates						
C.NAME='AUSTIN'						
	EMPLOYERS_IX	FULL	428605	651	2347	13327
IND	APP_YEAR	RANGE SCAN	53829	115	0	0
TABLE /	APPLICATIONS_IX	BY INDEX RO...	1	378	0	0
Filter Predicates						
A.EMPLOYER_ID=E.ID						
TABLE ACC	APPLICATIONS_IX	BY INDEX RO...	53829	378	368	27990
INDEX	APP_YEAR	RANGE SCAN	53829	115	116	9700
Access Predicates						

```
Statistics
-----------------------------------------------------------
              8  CPU used by this session
              8  CPU used when call started
              8  DB time
              5  Requests to/from client
           2764  consistent gets
              3  consistent gets examination
              3  consistent gets examination (fastpath)
           2764  consistent gets from cache
           2761  consistent gets pin
           2761  consistent gets pin (fastpath)
              5  non-idle wait count
              2  opened cursors cumulative
              1  opened cursors current
              1  pinned cursors current
           2764  session logical reads
              6  user calls
```

3.2.0 Partitioning

Range Partitioning: It's particularly useful for time-based data where each partition can correspond to a period like a day, month, or year. But the current data is for 14 years, and partitioning based on fiscal year will only add redundancy. Hence this option is not chosen.

List Partitioning: This approach is suitable for data with a discrete set of values, such as regions or categories. But there are too many cities in the current data and partition by this only create unnecessarily small partitions, adding overhead and resulting in slow running queries.

Hash Partitioning: This is useful when the goal is balanced distribution of data for even performance across partitions. Hash partitioning can be used with a smaller number of partitions (2 or 4), as this will divide the applications table into partitions of size 10 to 25 MB resulting in improved query performance.

To conclude, from various partitioning schemes available like List Partitioning, Range Partitioning, Hash Partitioning, we decided to go with Hash partitions on fiscal year column in applications table as our data size is relatively small (=~50MB) and having too many partitions based on list or range might not be useful.

Below is the SQL used that can be used to create tablespaces and partitioned tables.

```sql
CREATE TABLESPACE p1 DATAFILE 'p1.dbf' SIZE 10M AUTOEXTEND ON;
CREATE TABLESPACE p2 DATAFILE 'p2.dbf' SIZE 10M AUTOEXTEND ON;

CREATE TABLE applications_pt
(
    appid int PRIMARY KEY,
    fyear int,
    initial_approval int,
    initial_denial int,
    continuing_approval int,
     continuing_denial int,
     employer_id int,
     naics_id int,
     FOREIGN key(employer_id) REFERENCES Employers(id),
     FOREIGN key(naics_id) REFERENCES NAICS(id)

 )
PARTITION BY HASH (fyear)
PARTITIONS 2
STORE IN (p1, p2);
```

Assesing Query Performance before and aftering partitioning:

As we can observed from the below auto trace statistics, clearly there is reduction in the number of logical reads by 50% when queried on paritioned table.

DML Query:

```sql
SET AUTOTRACE TRACEONLY STATISTICS;
select * from applications where fyear = 2020;
```

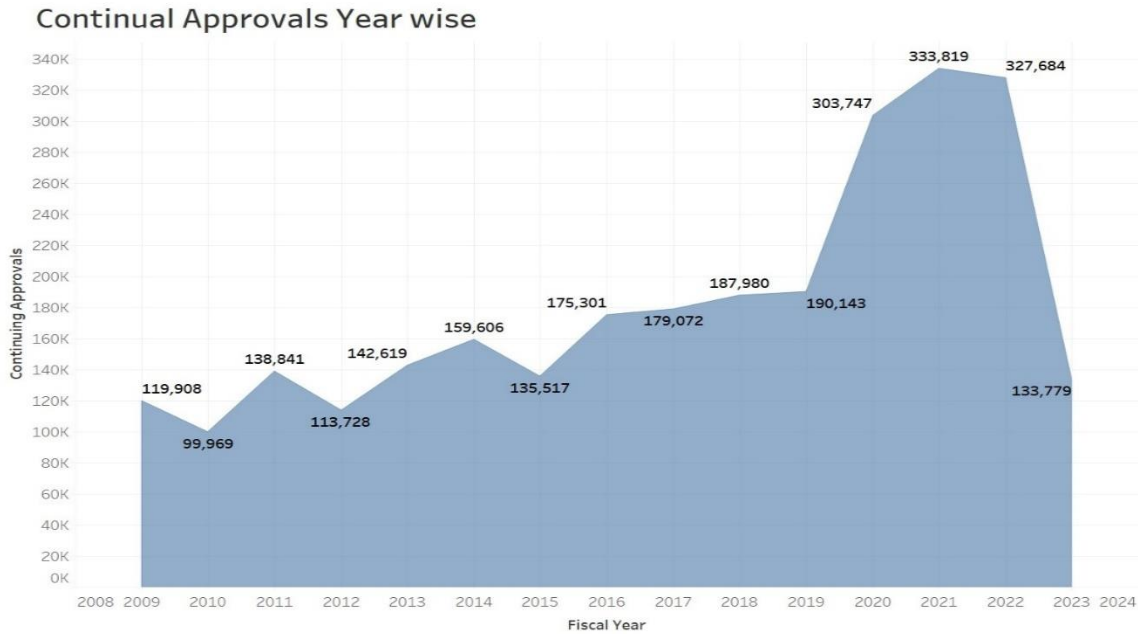
Auto trace statistics before parititioning:

```
Statistics
----------------------------------------------------------
              2  CPU used by this session
              2  CPU used when call started
              6  DB time
            112  Requests to/from client
       28835840  cell physical IO interconnect bytes
           3631  consistent gets
           3628  consistent gets direct
              3  consistent gets from cache
              3  consistent gets pin
              3  consistent gets pin (fastpath)
            251  non-idle wait count
              4  non-idle wait time
              2  opened cursors cumulative
             43  physical read total IO requests
       28835840  physical read total bytes
             27  physical read total multi block requests
           3631  session logical reads
              4  user I/O wait time
            114  user calls
```

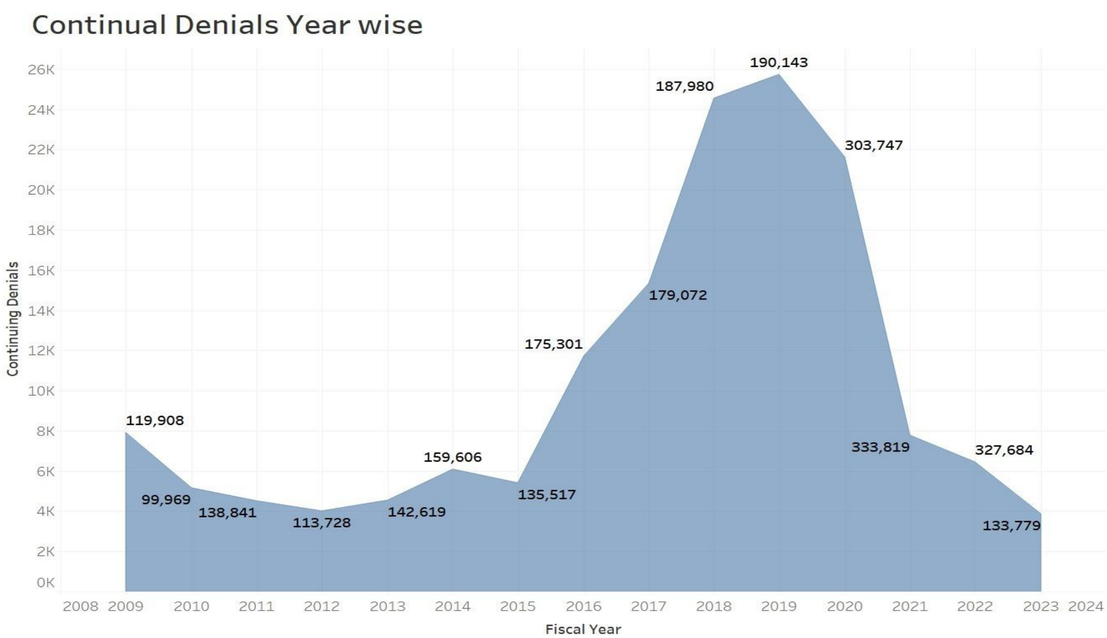
Auto trace statistics after parititioning:

```
Statistics
----------------------------------------------------------
              4  CPU used by this session
              4  CPU used when call started
              4  DB time
            133  Requests to/from client
            133  SQL*Net roundtrips to/from client
           1334  bytes received via SQL*Net from client
        1103575  bytes sent via SQL*Net to client
              2  calls to get snapshot scn: kcmgss
             11  calls to kcmgcs
           1699  consistent gets
           1699  consistent gets from cache
           1699  consistent gets pin
           1699  consistent gets pin (fastpath)
              2  execute count
       13918208  logical read bytes from cache
           1690  no work - consistent read gets
            240  non-idle wait count
              1  non-idle wait time
              2  opened cursors cumulative
              1  opened cursors current
              2  parse count (total)
              1  session cursor cache hits
           1699  session logical reads
              1  sorts (memory)
           1162  sorts (rows)
           1690  table scan blocks gotten
         379436  table scan disk non-IMC rows gotten
         379436  table scan rows gotten
```

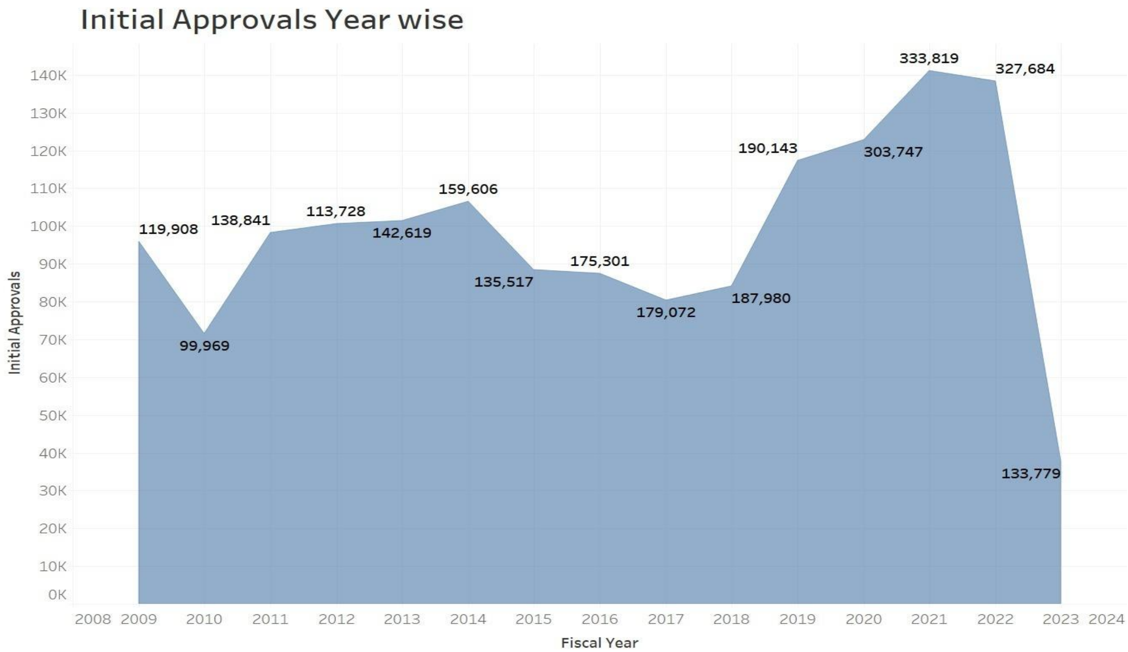
4. Data Visualization



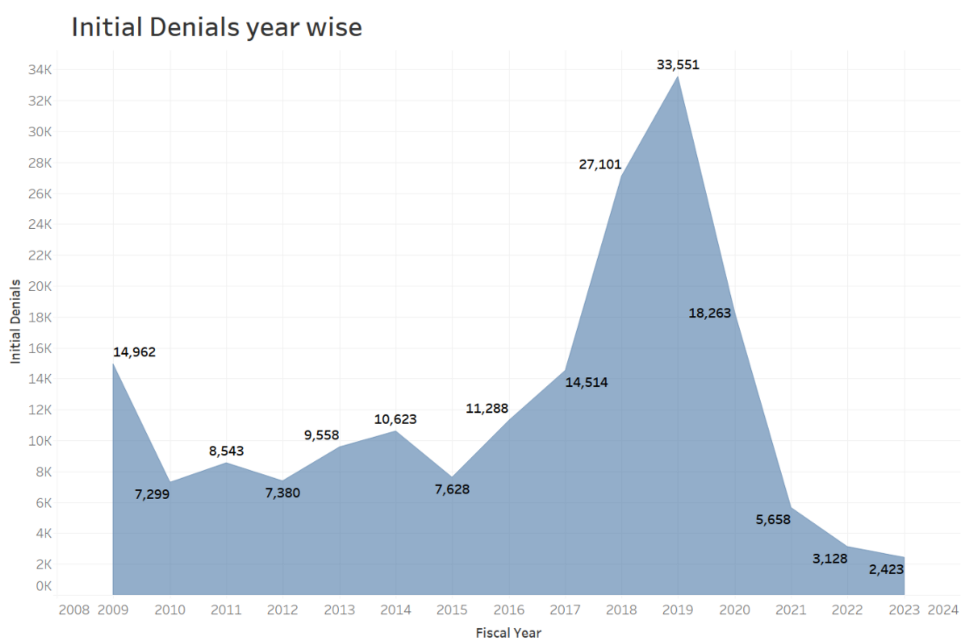
- The highest no of continual approvals can be seen in 2021 with 33819 approvals.



- The highest no of continual denials can be seen in 2019 with 187,980 denials.

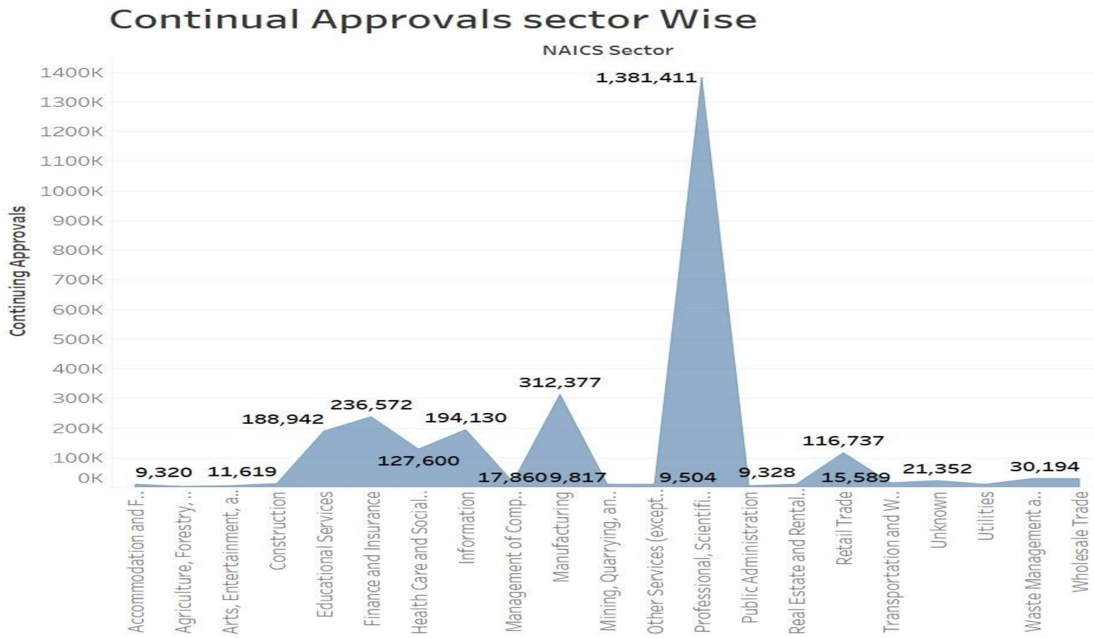


- The highest no of initial approvals can be seen in 2021 with 333,819 approvals.

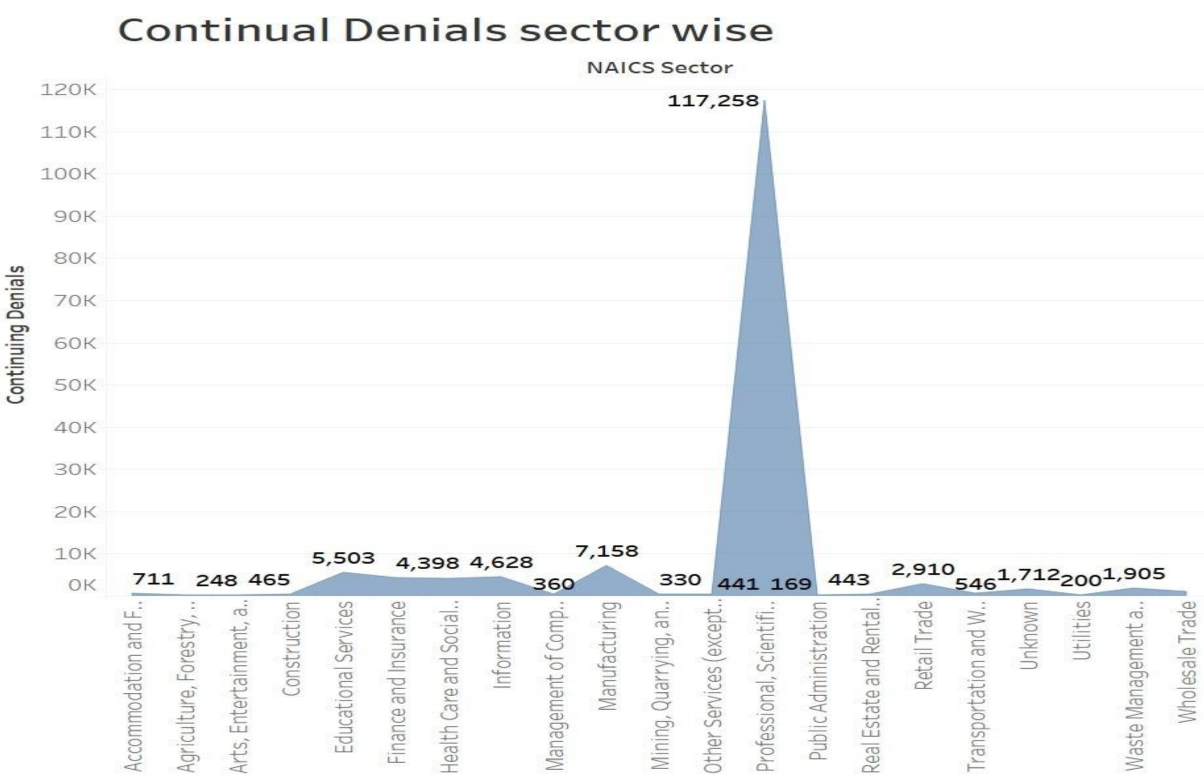


The plot of sum of Initial Denials for Fiscal Year. The marks are labeled by sum of Initial Denials.

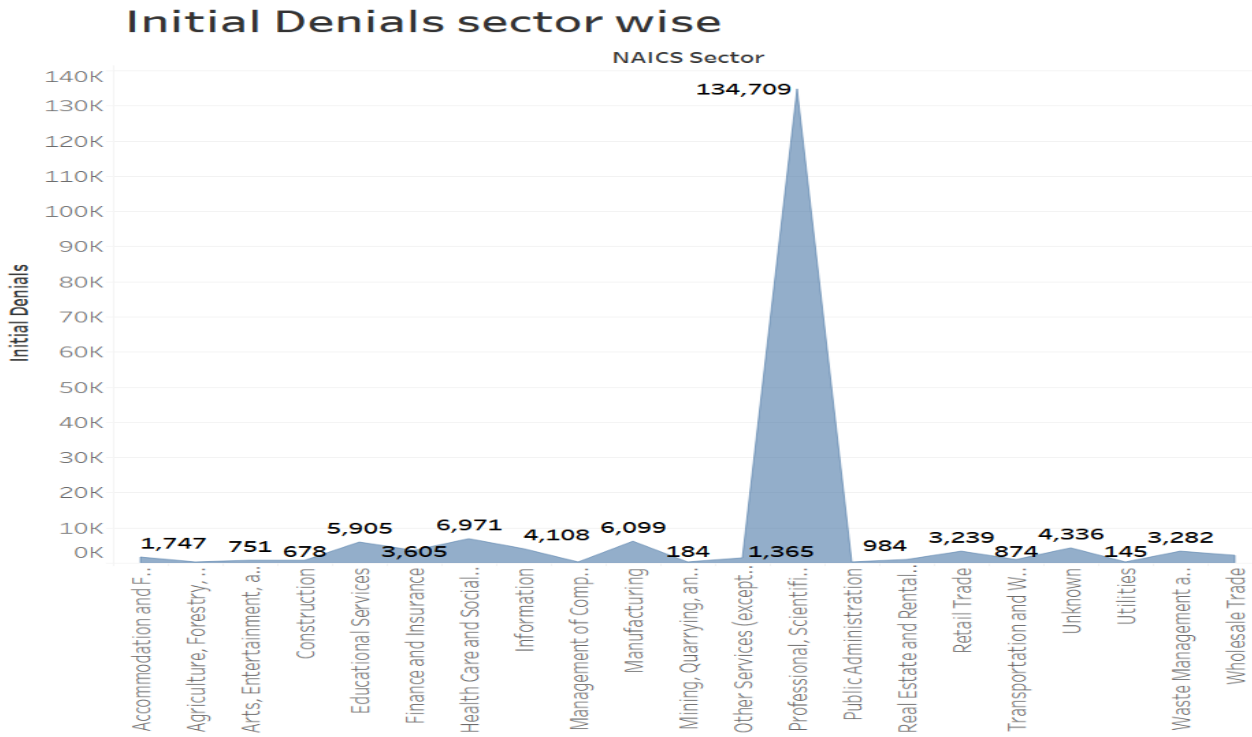
- The highest no of initial denials can be seen in 2019 with 33551 denials.



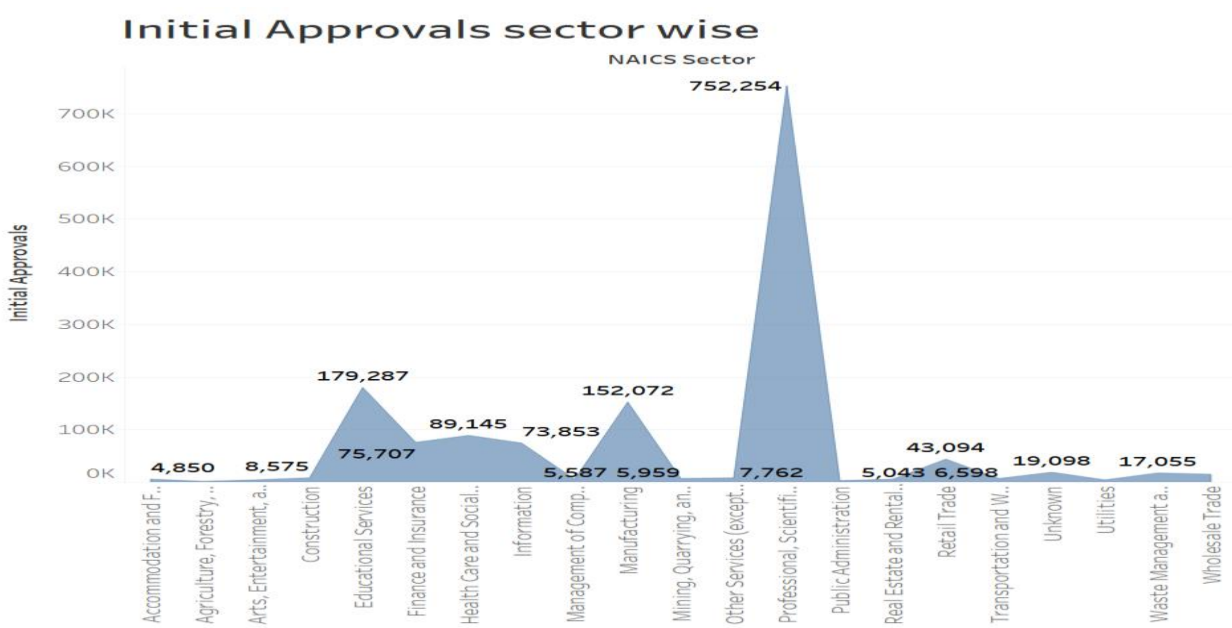
- The highest no of Continual approvals can be seen in professional, scientific and Technological sector from 2009 t0 2023.



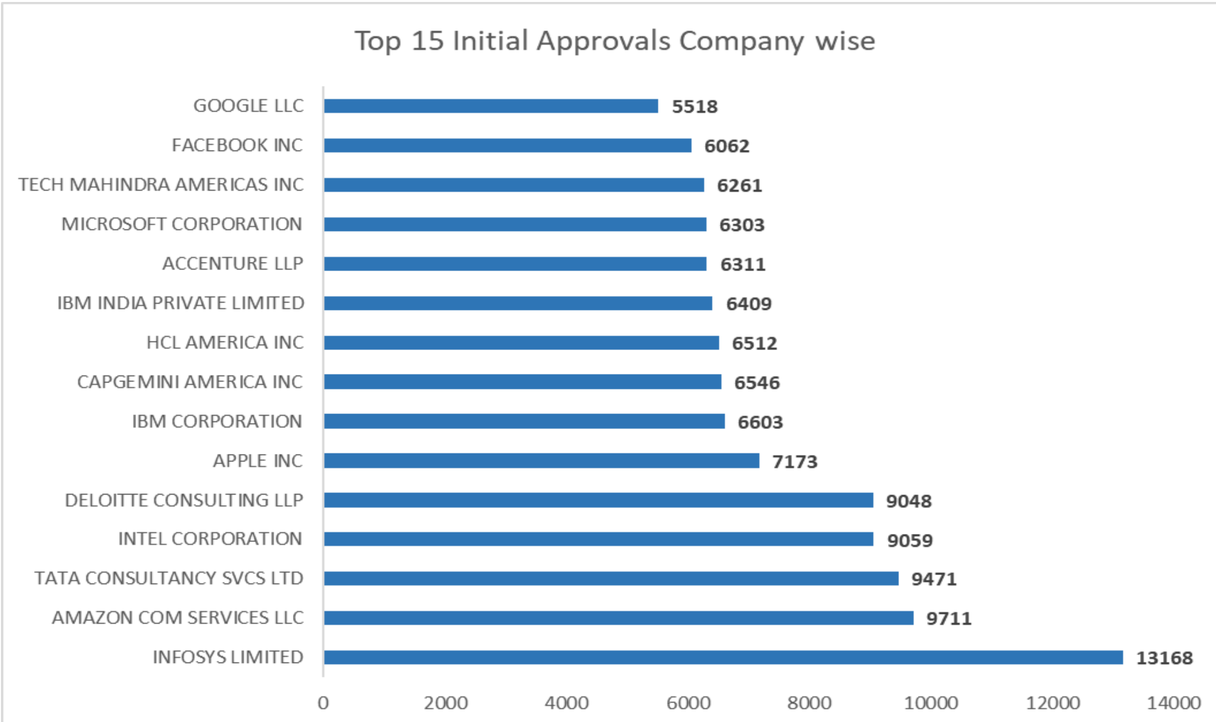
- The highest continual denials can be seen in Professional, scientific and Technological sector with 117258 denials from 2009 to 2023.



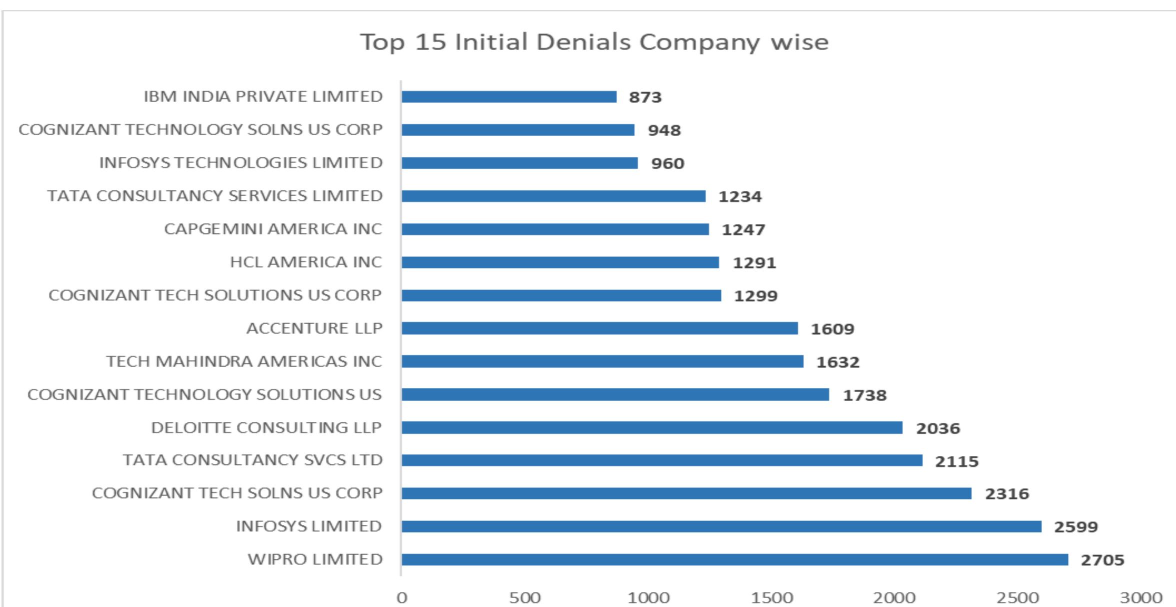
- The highest no of initial denials can be seen in professional, scientific and technological sectors with 134,709 denials from 2009 to 2023.



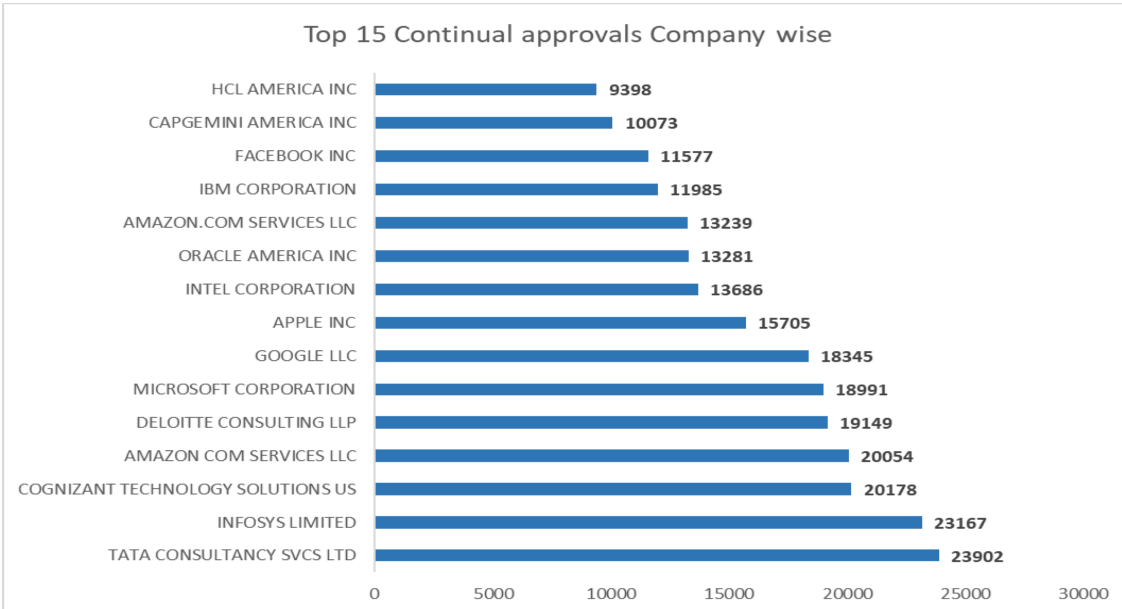
- Highest can be seen in professional, scientific and technological sector with 753,254 approvals from 2009 to 2023.



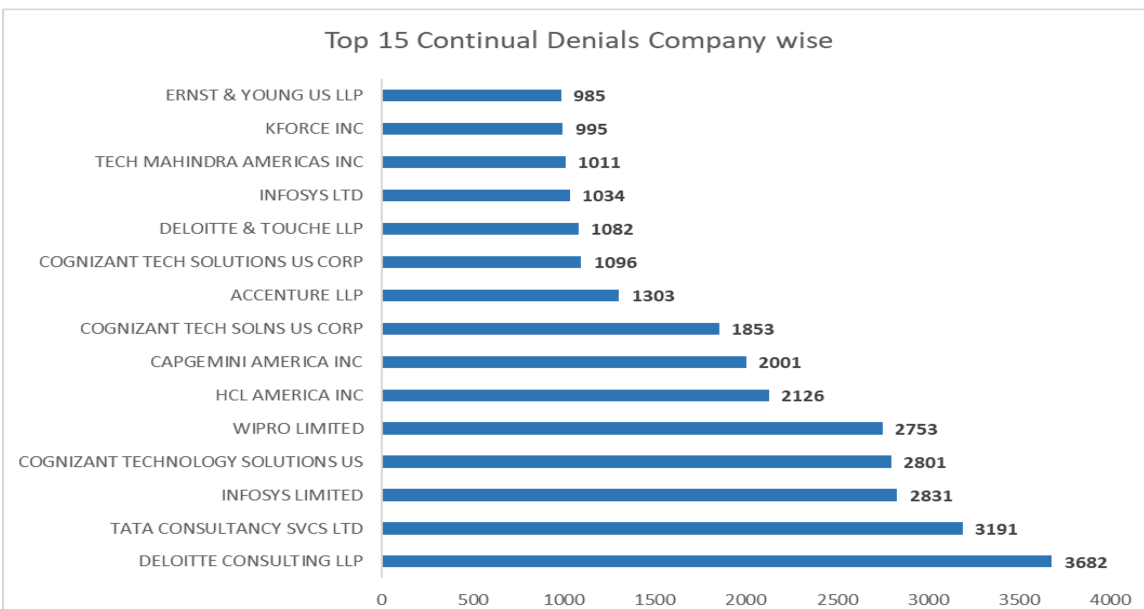
- The highest no of approvals can be seen in Infosys with 1318 approvals followed by Amazon from 2009 to 2023.



- The highest initial denials can be seen in Wipro with 2705 denials followed by Infosys from 2009 to 2023.



- The highest continual approvals can be seen in TCS with 23902 approvals followed by Infosys from 2009 to 2023.



- The highest number of continual denials can be seen in Deloitte with 3682 denials followed by TCS from 2009 to 2023.