

Indices (or Indexes)

In this lecture

- Types of indexes in Teradata.
- Primary indexes.
- Partitioned primary indexes.
- Join indexes.

Motivation: Why do we care?

- Indexes are the most important part of designing the database structure.
- Indexes not only provide an effective way to store data, but also help in determining effective access paths to data.

Indexes in Teradata

- Unique/non-unique primary index (**UPI/NUPI**)
- Partitioned primary index (**PPI**)
- Unique/non-unique secondary index (**USI/NUSI**)
- Partitioned primary index (**PPI**)
- Join index (**JI**)

Index families

- **Primary indexes:** To distribute and retrieve data rows in a table. Storage and maintenance are free.
- **Partitioned primary index:** A table organization to optimize the physical database design for range constraint queries. Storage is 2 bytes per row.
- **Raw data extensions:** Any structure that duplicates or points to primary data for purposes of better performance. These are like secondary indexes (SI) or join indexes (JI). Storage and maintenance are not free.

Primary Indexes

- Cannot be modified once the table is non-empty...
- but you can either:
 - create a new table with the index structure you need.
 - copy the data out to another table (volatile or permanent), modify the index and move the data back.

Primary Indexes (cont.)

- You can modify the index of an empty table:

```
ALTER TABLE <table> MODIFY  
PRIMARY INDEX Index_Name(col1, col2, ...)
```

- Should you modify primary indexes? That depends.
- Usually other workarounds are possible, since PI determines the location of the data, so re-indexing involves moving data around (often unwanted).
- Workarounds: partitioned indexes or secondary indexes.

Choosing a good primary index

- A good primary index should satisfy the following three properties:
 - Access
 - Distribution
 - Volatility

Choosing a good primary index (cont.)

- **Access:** Choose the column that provides the best access path to the data.
 - Are the individual records commonly selected? Then use the PK.
 - Do you typically join this table? Then consider rather the join columns.

Choosing a good primary index (cont.)

- **Distribution:** Help Teradata distribute data evenly choosing columns that would have a regular distribution. Avoid implementing numeric values as `VARCHAR/CHAR` to avoid hash collisions.

Choosing a good primary index (cont.)

- **Volatility:** Choose a column with stable data values.
- This helps to reduce overhead of data maintenance (moving data around).

Example: Choosing a good index

- Suppose our client has the following tables:
- `Order(PRIMARY KEY(OrderNumber))` .
- `LineItem(PRIMARY KEY(OrderNumber, ItemNumber))` .

Example: Choosing a good index (cont.)

- Orders are commonly looked up by `orderNumber` in the `Order` table.
- Line items are typically accessed by joining the `Order` table to the `LineItem` table on `OrderNumber`.

Example: Choosing a good index (cont.)

- `OrderNumber` s are unique in the `order` table, and non-unique in the `LineItem` table.
- Since `OrderNumber` is the key to the `order` table
`OrderNumber` values do not typically change.
- **Trivia:** What should be the primary indices on each table, and why?
- **Hint:** think in terms of access, distribution and volatility.

Example (cont.)

- `OrderNumber` is the only choice for PI in `order`, and it satisfies access, distribution, volatility.
- The PK of `LineItem` satisfies distribution and volatility, but not access: the table has to be re-distributed every time it is joined with `order` !
- `OrderNumber` satisfies the three conditions in `LineItem`, hence this should be selected as PI of both tables.

Partitioned primary index

Partitioned primary index

- With a PPI, rows are sent to different AMPs, but also *local partitions* within each AMP are created.
- Normal PI access remains unchanged, but in the case of a range query, each AMP is able to localize the search on specific partitions within its workspace.
- This means that Teradata Optimizer knows the portions of a range of values stored and scan only those parts in the table.

Example

- 04 Indexes.sql
- Create EMP_SAL_NONPPI and run EXPLAIN on the query:
 - SELECT * from EMP_SAL_NONPPI where dob <= 2017-08-01;
- Red flags:
 - all-rows scan : polite synonym for "really slow".
 - no confidence : Optimizer has not collected statistics on this table. Here this is the least of our problems.

Example (cont.)

- So how do we get rid of that?
- Partition by range! (DDL for `EMP_SAL_PPI`).
- Four different types of

Case partitioning

```
/*CASE partition*/  
CREATE TABLE SALES_CASEPPI  
(  
    ORDER_ID INTEGER,  
    CUST_ID INTERGER,  
    ORDER_DT DATE,  
)  
PRIMARY INDEX(ORDER_ID)  
PARTITION BY CASE_N(ORDER_ID < 101,  
ORDER_ID < 201,  
ORDER_ID < 501,  
NO CASE, UNKNOWN);
```

Range-based partitioning

```
/*Range Partition table*/  
CREATE volatile TABLE EMP_SAL_PPI  
(  
  id INT,  
  Sal int,  
  dob date,  
  bonus int  
) primary index( id)  
PARTITION BY RANGE_N (dob BETWEEN DATE '2017-01-01'  
AND DATE '2017-12-01' EACH INTERVAL '1' DAY)  
on commit preserve rows;
```

Multi-level partitioning

```
CREATE TABLE SALES_MLPPI_TABLE
(
ORDER_ID INTEGER NOT NULL,
CUST_ID INTERGER,
ORDER_DT DATE,
)
PRIMARY INDEX(ORDER_ID)
PARTITION BY (
    RANGE_N(ORDER_DT BETWEEN DATE '2017-08-01'
    AND DATE '2017-12-31'
    EACH INTERVAL '1' DAY)

CASE_N (ORDER_ID < 1001,
ORDER_ID < 2001,
ORDER_ID < 3001,
NO CASE, UNKNOWN));
```

Character-based partitioning:

```
/*CHAR Partition*/  
CREATE TABLE SALES_CHAR_PPI (  
  ORDR_ID INTEGER,  
  EMP_NAME VARCHAR (30) CHARACTER,  
  PRIMARY INDEX (ORDR_ID)  
  PARTITION BY CASE_N (  
    EMP_NAME LIKE 'A%', EMP_NAME LIKE 'B%',  
    EMP_NAME LIKE 'C%', EMP_NAME LIKE 'D%',  
    EMP_NAME LIKE 'E%', EMP_NAME LIKE 'F%',  
    NO CASE, UNKNOWN);
```


Secondary Index

Secondary Index

- A table can contain **only one** primary index.
- More often, you will come across scenarios where the table contains other columns, using which the data is frequently accessed.
- Secondary indexes are used to avoid full table scan in those cases.
 - Optional and not involved in data distribution.
 - Stored in sub tables. These tables are built in all AMPs.
 - They can be created during table creation or after a table is created.
 - They also require maintenance since the sub-tables need to be updated for each new row.

The truth...

- Teradata runs extremely well **without** secondary indexes.
- Only recommended when queries that are run over and over.
- If the tables are modified, need to be recreated! Maintenance overhead.

```
CREATE UNIQUE INDEX (Column/Columns) ON <tablename >;
```

```
CREATE INDEX (Column/Columns) ON <tablename >;
```

Join Indexes

Join Index

A **JOIN INDEX** is a materialized view. Its definition is permanently stored and the data is updated whenever the base tables referred in the join index is updated. JOIN INDEX may contain one or more tables and also contain pre-aggregated data. Join indexes are mainly used for improving the performance. There are different types of join indexes available.

- Single Table Join Index (STJI)
- Multi Table Join Index (MTJI)
- Aggregated Join Index (AJI)

Example

Suppose we have the following tables:

```
CREATE SET TABLE EMPLOYEE, Fallback
(
EmployeeNo INTEGER,
FirstName VARCHAR(30) ,
LastName VARCHAR(30) ,
DOB DATE FORMAT 'YYYY-MM-DD',
JoinedDate DATE FORMAT 'YYYY-MM-DD',
DepartmentNo BYTEINT
)
UNIQUE PRIMARY INDEX ( EmployeeNo );
```

Example (cont.)

```
CREATE SET TABLE SALARY, Fallback
(
EmployeeNo INTEGER,
Gross INTEGER,
Deduction INTEGER,
NetPay INTEGER
)
PRIMARY INDEX ( EmployeeNo )
UNIQUE INDEX (EmployeeNo);
```

STJI: Example

Let's create a JOIN index on the `Employee` table.

```
CREATE JOIN INDEX Employee_JI  
AS  
SELECT EmployeeNo, FirstName, LastName,  
BirthDate, JoinedDate, DepartmentNo  
FROM Employee  
PRIMARY INDEX(FirstName);
```

- When the user submits a query with a `WHERE` clause on `EmployeeNo` then the UPI is used.
- If the query is on `FirstName`, then the system may access it on `Employee_JI`.
- On other columns (e.g. `LastName`), then full table scan is necessary.

MTJI: Example

```
CREATE JOIN INDEX Employee_Salary_JI
AS
SELECT a.EmployeeNo,a.FirstName,a.LastName,
a.BirthDate,a.JoinedDate, a.DepartmentNo
,b.Gross,b.Deduction,b.NetPay
FROM Employee a
INNER JOIN Salary b
ON(a.EmployeeNo=b.EmployeeNo)
PRIMARY INDEX(FirstName);
```

- If you run a query joining these tables, then Optimizer may choose to access the data from the join index directly.
- You can verify what will happen with `EXPLAIN` .

AJI: Example

```
CREATE JOIN INDEX Employee_Salary_JI
AS
SELECT a.DepartmentNo, SUM(b.NetPay) AS TotalPay
FROM Employee a
INNER JOIN Salary b
ON(a.EmployeeNo=b.EmployeeNo)
GROUP BY a.DepartmentNo
Primary Index(DepartmentNo);
```

- If a table is consistently aggregated on certain columns, you can create an aggregate join index.
- It supports only `SUM` and `COUNT` .

Your turn!

Exercise

In the `tutorial` database, we want to retrieve regularly:

- Total sold per customer.
 - Total sold per region.
1. Write queries to calculate this.
 2. Run `EXPLAIN` on your queries.
 3. Propose and implement an index structure to make this queries more efficient.
 4. Run `EXPLAIN` with the new indexed tables and compare.