Advanced Database Design Patterns in Action

<https://www.udemy.com/course/advanced-database-design-patterns-in-action>

Table of Contents

[1.Indexes 2](#_Toc188282299)

[Clustered Index & Non-Clustered indexes. 2](#_Toc188282300)

[Operators 3](#_Toc188282301)

[Scan Operator 3](#_Toc188282302)

[Seek Operator 3](#_Toc188282303)

[Composite Index 6](#_Toc188282304)

[Types of Joins 6](#_Toc188282305)

[Nested Loop Joins 6](#_Toc188282306)

[Merge Join 6](#_Toc188282307)

[Hash Join 6](#_Toc188282308)

[2. DB Patterns 6](#_Toc188282309)

[2.1 Lookup Table 6](#_Toc188282310)

[2.2 Associative Table 7](#_Toc188282311)

[2.3 History Table 7](#_Toc188282312)

[Horizontal Partitioning 8](#_Toc188282313)

[Indexes 8](#_Toc188282314)

[Vertical Partitioning 8](#_Toc188282315)

[Internal Fragmentation 9](#_Toc188282316)

[External Fragmentation 9](#_Toc188282317)

[Examples of issues and optimizations 11](#_Toc188282318)

[Modeling Hierarchical data 12](#_Toc188282319)

[Adjacency List pattern 12](#_Toc188282320)

[Nested Set pattern 12](#_Toc188282321)

[Path Enumeration Pattern 13](#_Toc188282322)

[Modeling Inheritance 13](#_Toc188282323)

[Single Table Inheritance 13](#_Toc188282324)

[Class Table Inheritance 13](#_Toc188282325)

# 1.Indexes

## Clustered Index & Non-Clustered indexes.

Clustered Index is the main index i.e PK. Non-Clustered indexes means any secondary indexes. Clustered Index can change the actual data or has reference of actual page on disk at it’s leaf node. Secondary indexes at their leaf node, hold reference to copy of data on which the index was created and reference to the PK or Clustered Index.

PK should be ever increasing to minimize page splits. If PK is user selected random numbers as below and now a new key 7 is to be added, the same page has to be reorganized, and data might have to be moved out to new page. This will increase fragmentation.

A diagram of a flowchart

AI-generated content may be incorrect.

If the data required by the query is present in the leaf node of the secondary index then there is no need to go to actual physical location of the page to read the page.

Primary Key uniquely identifies each record within the table (Logically).

Clustered Index is the physical index structure that provides efficient data retrieval. PK is logical and it’s physical index structure is called Clustered Index.

90% of cases, clustered index will be created on a single column to pin point a specified record but in some cases it can be created on a group of columns as well.

Example:

|  |  |  |
| --- | --- | --- |
| State | Customer ID (PK) | Name |
| IL | 10 |  |
| NY | 20 |  |

The primary key of the above table is Customer ID but Clustered Index (State, Customer ID). This makes sense if the primary query running on the table is give me customer id in specific state.

## Operators

### Scan Operator

1. Clustered Index Scan

2. Index Scan

3. Table Scan

The worst is Table Scan. That means you don’t have any index and will load all the page files to get any result. If there is an secondary index then Index Scan will be done on the column.

### Seek Operator

1. Clustered Index Seek
2. Index Seek

When you do EXPLAIN <query>, you will see the result with these operators. They will tell you the number of rows read. We should try to minimize the number of rows read. Another parameter to look for is number of pages loaded.

**SELECT \* FROM CUSTOMERS;**

This query will result in Clustered Index Scan

A diagram of a tree

AI-generated content may be incorrect.

**SELECT \* FROM CUSTOMERS WHERE CUSTOMER\_ID = 100;**

This query will result in Clustered Index Seek

**A diagram of a network

AI-generated content may be incorrect.**

**SELECT \* FROM CUSTOMERS WHERE CUSTOMER\_ID >= 10 and CUSTOMER\_ID <= 100;**

This query will result in Clustered Index Seek

**A diagram of a network

AI-generated content may be incorrect.**

For Non-Clustered Index:

**SELECT FIRST\_NAME, LAST\_NAME FROM CUSTOMER WHERE EMAIL = ‘ABC@GMAIL.COM’;**

**A diagram of a diagram

AI-generated content may be incorrect.**

Here in this example, we have a secondary index on email field. So the query planner will first traverse the secondary index, get to leaf node, since leaf node only contains email and not first, last names, it will go to Primary index/ Clustered Index and traverse that to find the first & last name.

The secondary index, on it’s leaf will store (email address, customer\_id). So when it reaches the correct leaf, it gets customer\_id and now it traverses the primary key index to get to the page with that customer id.

If we would have created secondary index as (email, first\_name, last\_name) then the search would have ended at secondary index itself. Once the leaf node of secondary index is found, all the data is present there so we just return that data.

## Composite Index

SELECT ORDER\_ID FROM ORDER WHERE STATUS = ‘OPEN’ AND DATE >= ‘2024-03-01’ AND DATE <= ‘2024-03-03’;

Here we are looking at 2 columns, Status and Date so we can create a composite index **idx(status, date)**. Should we create idx(status, date) or idx(date, status) ? This depends on data. Remember it is going to filter first on the first column that you specify idx(status, date), this will be status column and in idx(date, status) it will be date column. If you have 10k column and idx(date, status) as column, let’s say you are searching between dates and status = ‘OPEN’. Index will first search for date, let’s say this returns 1000 rows, now in these 1000 rows, it will filter out status = ‘OPEN’. If in your data, there are only 10 rows with status=’OPEN’, then it makes sense to create index as idx(status, date) because it will first select only 10 rows and then filter dates from these 10 rows which is more efficient.

Whichever column can be more exclusive, should come first in composite index definition.

We have composite index as idx(status, date). On this index if we search by only date then this will result in Full Index Scan. Because we missed the first status column in the search query. So in case of Composite Indexes, it’s best to search by both columns or the 1st column.

## Types of Joins

### Nested Loop Joins

For record in outer\_table:

For record in inner join table:

### Merge Join

When both tables are sorted, this is the same as merge two sorted array problem.

### Hash Join

Smaller table is converted into hash table so memory requirement is more. If tables are very large then not possible.

# 2. DB Patterns

## 2.1 Lookup Table

Let’s take an example of Orders Table and Order Status Table

|  |  |  |  |
| --- | --- | --- | --- |
| Order ID (PK) | Customer ID | Status | Date |
| 1 | 4 | 1 | 2024-03-05 |

|  |  |
| --- | --- |
| Status Enum | Status Readable Name |
| 1 | Open |
| 2 | Delivered |
| 3 | Delayed |

We just created a lookup table Order Status table

## 2.2 Associative Table

Useful especially to model Many-Many relationships. Singer and Album Tables as example.

One singer can have multiple Albums and an Album can have multiple singers.

|  |  |
| --- | --- |
| Singer ID | Singer Name |
| 1 | Paul |

|  |  |
| --- | --- |
| Album ID | Album Name |
| 5 | Greatest Hits |

Singer-Album Associative Table

|  |  |
| --- | --- |
| Singer ID | Album ID |
| 1 | 5 |
| 1 | 3 |
| 34 | 5 |

## 2.3 History Table

All Audit actions are recorded. Especially useful and needed for DDD patterns. When a column is updated, the previous value is lost. To store this value if required for future references, we create a history table.

|  |  |  |  |
| --- | --- | --- | --- |
| Order ID (PK) | Customer ID | Status | Date |
|  |  |  |  |
| OrderStatus ID (PK) | Order ID (FK) | Timestamp | Status |
| 1 | 1 | 2024-03-01 10:10:00 | OPEN |
| 2 | 3 | 2024-03-01 10:12:00 | OPEN |
| 3 | 1 | 2024-03-01 10:13:00 | PROCESSING |

# Horizontal Partitioning

Most used types of horizontal partitioning are:

1. **Hash Based**

CREATE TABLE Stores (

customer\_name varchar(40),

bill\_number varchar(20) NOT NULL,

store\_id int PRIMARY KEY NOT NULL,

bill\_date date NOT NULL,

amount decimal(8, 2) NOT NULL

)

PARTITION BY HASH (store\_id)

PARTITIONS 4;

1. **Range Based**

CREATE TABLE Sales (

customer\_id int NOT NULL,

customer\_name varchar(40),

store\_id varchar(20) NOT NULL,

bill\_number int NOT NULL,

bill\_date date PRIMARY KEY NOT NULL,

amount decimal(8, 2) NOT NULL

)

PARTITION BY RANGE (YEAR(bill\_date)) (

PARTITION p0 VALUES LESS THAN (2016),

PARTITION p1 VALUES LESS THAN (2017),

PARTITION p2 VALUES LESS THAN (2018),

PARTITION p3 VALUES LESS THAN (2020)

);

**3. List (List of values)**

## Indexes

# Vertical Partitioning

Data page looks like this. This example is from SQLServer, in MySQL the page is 4KB

A diagram of a computer data

AI-generated content may be incorrect.

Two types of index fragmentation can happen

## Internal Fragmentation

Internal fragmentation means in data page, the space is wasted. There is lot of empty space left in data page. Causes include **delete operation, large rows, insert update operations causing page splits**. Example: You have a page with 10 rows, now 3 of the rows are deleted. That space is left free, can a new row be written there? What if the new row being written is bigger and overflowing into the next row? So this space is left empty (I guess ☺).

If you have very large rows or a row with very large column (say 5k) then only one row can fit into a single page. If you have to retrieve 10k rows, means you are reading 10k pages into memory.

## External Fragmentation

Data pages are organized sequentially on disk. In external fragmentation, the physical order of storage changes and affects performance, especially for range queries.

A diagram of a physical page layout

AI-generated content may be incorrect.

**Causes include page splits, not enough contagious space available.**

**A diagram of a diagram

AI-generated content may be incorrect.**

## Examples of issues and optimizations

#### Customer Table with large description field

Customer Table

* ID
* Name
* Description (varchar 2000)

When you do a select from customer table, large number pages are read, although description field is not there in select statement.

Solution

Customer Table Customer Description Table

* ID - ID
* Name - Customer ID (FK to Customer ID)

- Description (Varchar 2000)

This way the description field is stored separately and doesn’t affect Customer Table’s performance.

# Modeling Hierarchical data

## Adjacency List pattern

|  |  |  |
| --- | --- | --- |
| ID | Name | Parent\_ID |
| 1 | Product | NULL |
| 10 | Electronics | 1 |
| 23 | Laptop | 10 |
| 15 | Cloths | 1 |
| 78 | Shirt | 15 |

## Nested Set pattern

A diagram of a product

AI-generated content may be incorrect.

Becomes very easy to find all the children of a parent. Used by Jive in forum use case. Give me all the children of the thread.

## Path Enumeration Pattern

|  |  |  |
| --- | --- | --- |
| ID | Name | Path |
| 1 | Product | NULL |
| 2 | Electronics | 1 |
| 8 | Laptop | 1 | 8 |
| 25 | LG Laptop | 1 | 8 | 23 |

# Modeling Inheritance

### Single Table Inheritance

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | Type | Price | Title | Author | Size |
| 1 | Book | 20 | Test Title | abc | NULL |
| 2 | Shirt | 10 | NULL | NULL | XL |

### Class Table Inheritance

A diagram of a vendor

AI-generated content may be incorrect.