Physical Design

Read Chapter 9 of Riguzzi et al. Sistemi Informativi

Physical Design

- Design of the physical structures of the database
- Last phase of database design
- Input:
  - Database logical schema
  - Information on the load
- Output:
  - Definition of the primary structures of tables
  - Definition of indexes on tables
  - Setting of a number of DBMS specific parameters

Choice of Indexes

- Very important
- Usually one index on the primary key
  - Sometimes compulsory
  - Useful because the primary key is often involved in joins and selections
- Other indexes for easing certain selections and joins
- It is possible to investigate how the indexes are used in queries by the command
  - SHOW PLAN
- If the performances are unsatisfactory, indexes can be added or removed

Index Creation

- Pattern common to different DBMS
  - CREATE [UNIQUE] INDEX *IndexName* ON *TableName*(*AttributeList*)
  - DROP INDEX *IndexName*
- Syntax details vary widely among DBMS
- UNIQUE: AttributeList is a superkey
- No way to specify the type of index (B+Tree, Hash,...)

Physical Structures

- Primary structures (defined when creating the table):
  - heap ("unclustered")
  - ordered ("clustered"), also on a pseudokey
  - hash ("clustered"), also on a pseudokey
  - array ("clustered"), also on a pseudokey
  - clustering of different tables
- Indexes (primary/secondary, defined when creating the index)
  - ISAM
  - B+-tree
  - bitmap

SQL Server Physical Structures

- Primary structures
  - Heap
  - Ordered with sparse B+-tree
- Secondary indexes: dense B+-tree

Relational Index Creation

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX *index_name*

ON <object> **(** *column* [ASC | DESC] [,...*n*] **)**

<object> ::= table_or_view

Sort Order

- You should consider whether the data for the index key column should be stored in ascending or descending order.
- Ascending is the default
- Keywords: ASC (ascending) and DESC (descending)
- Specifying the order in which key values are stored in an index is useful when queries referencing the table have ORDER BY clauses that specify order directions for the columns in that index.
- In these cases, the index can remove the need for a SORT operator in the query plan; therefore, this makes the query more efficient.

Sort Order Example

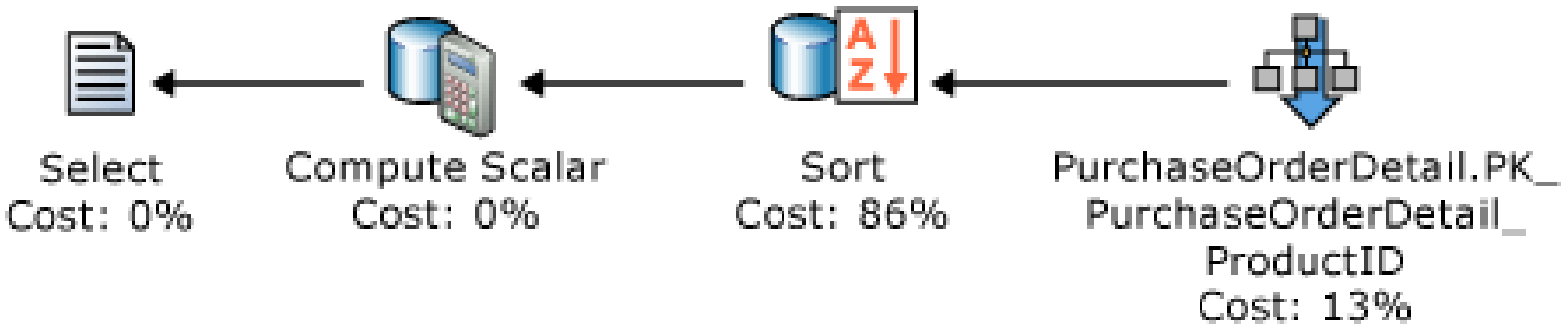
- The buyers in the Adventure Works Cycles purchasing department have to evaluate the quality of products they purchase from vendors. The buyers are most interested in finding products sent by these vendors with a high rejection rate.
- Retrieving the data to meet this criteria requires the RejectedQty column in the Purchasing.PurchaseOrderDetail table to be sorted in descending order (large to small) and the ProductID column to be sorted in ascending order (small to large).

Sort Order Example

```
USE AdventureWorks;
SELECT RejectedQty, ((RejectedQty/OrderQty)*100)
    AS RejectionRate, ProductID, DueDate
FROM Purchasing.PurchaseOrderDetail
ORDER BY RejectedQty DESC, ProductID ASC;
```

Sort Order Example

- The following execution plan for this query shows that the query optimizer used a SORT operator to return the result set in the order specified by the ORDER BY clause.



Try it

- Launch Visual Studio
- Add a new Data connection
- Create a New query
- Press button “Include Actual Execution Plan”
- Run the query

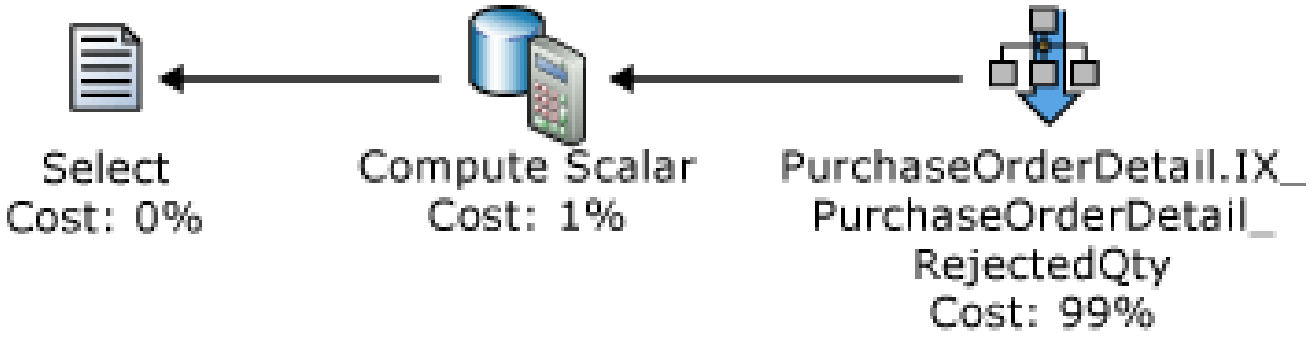
Sort Order Example

- If an index is created with key columns that match those in the ORDER BY clause in the query, the SORT operator can be eliminated in the query plan and the query plan is more efficient:

```
CREATE NONCLUSTERED INDEX
    IX_PurchaseOrderDetail_RejectedQty
ON Purchasing.PurchaseOrderDetail
    (RejectedQty DESC, ProductID ASC);
```

Sort Order Example

- After the query is executed again, the following execution plan shows that the SORT operator has been eliminated and the newly created nonclustered index is used.



Sort Order

- SQL Server can move equally efficiently in either direction. An index defined as (RejectedQty DESC, ProductID ASC) can still be used for a query in which the sort direction of the columns in the ORDER BY clause are reversed.
- For example, a query with the ORDER BY clause ORDER BY RejectedQty ASC, ProductID DESC can use the index.
- A query with the ORDER BY clause ORDER BY RejectedQty ASC, ProductID ASC can not use the index.

Showing the Execution Plan of a Query

SET SHOWPLAN_TEXT ON

After this SET statement is executed, SQL Server returns the execution plan information for each query in text. The Transact-SQL statements or batches are not executed.

Oracle Physical Structures

- Primary structures
  - Heap
  - Ordered with sparse B+-tree
  - Hash
  - Multitable clusters
- Secondary indexes:
  - dense B+-tree
  - bit-map

Ordered with sparse B+-tree

- Also called index-organized table.
- Oracle Database maintains the table rows, both primary key column values and nonkey column values, in an index built on the primary key.
- Index-organized tables are therefore best suited for primary key-based access and manipulation.
- You must specify a primary key for an index-organized table, because the primary key uniquely identifies a row.
- Similar to SQL server

Indexes

- Type of indexes:
  - B+-tree indexes
  - B+-tree cluster indexes (for multitable clusters)
  - Bitmap indexes
  - Bitmap join indexes:
    - Bitmap index for the join of two or more tables
    - Useful for datawarehousing

Unique and Nonunique Indexes

- Unique indexes guarantee that no two rows of a table have duplicate values in the key column (or columns).
- Oracle recommends that unique indexes be created explicitly, using CREATE UNIQUE INDEX.

CREATE INDEX

CREATE [UNIQUE | BITMAP] INDEX [schema.]index
  ON [schema.]table
(column [ASC | DESC]
  [, column [ASC | DESC]]...);

- By default, Oracle Database creates B+-tree indexes.

DB2 Physical Structures

- Primary structure:
  - Heap
  - Array (Range-clustered tables)
- Indexes:
  - dense B+-trees

- Indexes are bidirectional by default: they allow forward and reverse scans

Range-clustered tables

- Array primary structure
- The table should have an integer key that is tightly clustered (dense) over the range of possible values.
- The columns of this integer key must not be nullable, and the key should logically be the primary key of the table.
- The allocation of all the space for the complete set of rows in the defined key sequence range is done during table creation

Secondary indexes

- Secondary indexes contain only keys and record IDs in the index structure.
- The record IDs always point to rows in the data pages.
- Dense indexes

Indexes

- Only B+trees

```
CREATE [ UNIQUE  ] INDEX index
  ON table
(column [ ASC | DESC ]
  [, column [ ASC | DESC ] ]...)
```

Views

CREATE VIEW [*schema_name* **.**] *view_name* [
(*column* [**,**...*n*] **)**]

AS *select_statement* [;]

[WITH CHECK OPTION]

column

- *column*
  - If *column* is not specified, the view columns acquire the same names as the columns in the SELECT statement.
  - If it is specified, it is the name to be used for a column in a view.

Updatable Views

- You can modify the data of an underlying base table through a view, as long as the following conditions are true:
  - Any modifications, including UPDATE, INSERT, and DELETE statements, must reference columns from only one base table.
  - The columns being modified in the view must directly reference the underlying data in the table columns. The columns cannot be derived in any other way

WITH CHECK OPTION

- Forces all data modification statements executed against the view to satisfy the criteria set within *select_statement*. When a row is modified through a view, the WITH CHECK OPTION makes sure the data remains visible through the view after the modification is committed.

View Example (SQL Server)

```
USE AdventureWorks ;
GO
CREATE VIEW hiredate_view
AS
SELECT c.FirstName, c.LastName, e.EmployeeID,
    e.HireDate
FROM HumanResources.Employee e JOIN
    Person.Contact c on e.ContactID = c.ContactID ;
GO
```

An UPDATE that modifies an employee LastName and HireDate returns an error

View Example (SQL Server)

```
USE AdventureWorks ;
GO
CREATE VIEW SeattleOnly
AS
SELECT p.LastName, p.FirstName, p.City,
FROM Person p WHERE p.City = 'Seattle'
WITH CHECK OPTION ;
GO
```

An UPDATE that changes the city of a Person returns an error.