

Optimización - SQL Server

Dr. Gerardo Rossel



Bases de Datos

2024

Estructura de las Tablas e Indices

SQL Server almacena datos en tablas e índices.

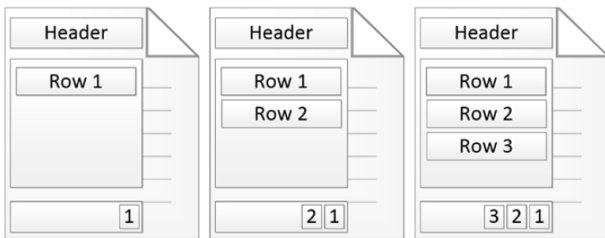
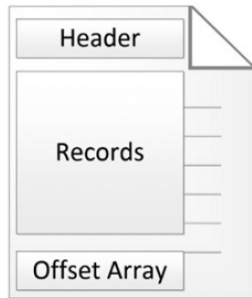
- Heap Tables
- Clustered Indexes
- Nonclustered Indexes
- Columnstore
- In Memory

Heap vs Clustered

- Una tabla se puede organizar de una de dos maneras: como un *heap* o como un *árbol B* (B-Tree). Técnicamente, la tabla se organiza como un *árbol B* cuando tiene un *clustered index* definido y como un *heap* cuando no.
- Cuando agrega una restricción de *Primary Key* a una tabla, SQL Server la aplicará utilizando un *clustered index* a menos que especifique explícitamente la palabra clave *NONCLUSTERED* o ya exista un *clustered index* en la tabla.
- Cuando agrega una restricción única a una tabla, SQL Server la aplicará utilizando un *nonclustered index* a menos que especifique la palabra clave *CLUSTERED*

Páginas

- El área de almacenamiento más básica es una página.
- Cada página ocupa 8KB.
- Cuando SQL Server interactúa con los archivos de la base de datos, la unidad más pequeña en la que puede ocurrir una operación de E / S está en el nivel de página
- Las páginas se agrupan de ocho en ocho en estructuras llamadas extents.



- SQL Server opera con páginas almacenadas en el *buffer pool*
- La interacción con las páginas ocurre principalmente dentro del *buffer pool*, no directamente en el disco.
- Cuando SQL Server lee una página, verifica si ya está en la caché de datos (*buffer pool*).
 - Si la página está en memoria, SQL Server realiza una lectura lógica (lee desde la memoria).
 - Si la página no está en memoria, SQL Server realiza una lectura física (extrae la página del disco a la memoria) seguida de una lectura lógica.
- Cuando SQL Server escribe una página, verifica si ya está en la caché de datos.
 - Si la página está en memoria, SQL Server realiza una escritura lógica.
 - El indicador *dirty* en el encabezado de la página indica que la página en memoria es más reciente que en el disco.

Páginas

Diferentes tipos de páginas



- File header page. Contiene información de metadatos para el file en cuestión.
- Boot page. Idem File Header pero para toda la Base de Datos.
- Page Free Space (PFS)
 - Es la segunda página y después se ubica cada 8088 páginas.
 - Cada byte en la página PFS representa una página subsiguiente en el archivo de datos y proporciona cierta información de asignación simple sobre la página; es decir, determina la cantidad aproximada de espacio libre en la página.

Páginas

Diferentes tipos de páginas



- Global Allocation Map (GAM) page

- La página GAM determina si un conjunto de *extents* ha sido designado para su uso como un *extent* uniforme. Un propósito secundario de la página GAM es ayudar a determinar si el *extent* está libre y disponible para la asignación.
- Cada página GAM proporciona un mapa de todos los *extent* subsiguientes en cada intervalo GAM. Un intervalo GAM consiste en las 64,000 *extents*, o 4 GB.
- Cada bit en la página GAM representa un *extent* a continuación de la página GAM.

Páginas

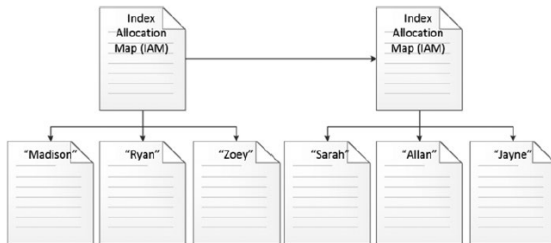
Diferentes tipos de páginas



- Shared Global Allocation Map (SGAM) page
 - Casi idéntica a la página GAM es la página de Asignación Global Compartida (SGAM).
 - La diferencia principal entre las páginas es que la página SGAM determina si un *extent* está asignado como un *extent* mixto.
- Differential Changed Map (DCM) page
 - Sirve para determinar cuando un *extent* ha cambiado.
 - Cuando ocurre un *full backup*, todos los bits en la página DCM se restablecen a 0. El bit luego vuelve a cambiar a 1 cuando se produce un cambio dentro del *extent* asociado.
- Minimally Logged (ML)
 - se utiliza para indicar cuándo una extensión en un intervalo GAM ha sido modificada por una operación de registro mínimo

- Index Allocation Map (IAM) page
 - SQL Server necesita saber si la información en una página está asociada a una tabla o índice específico.
 - Toda tabla o índice comienza con una página IAM específica que indica cuales *extent* dentro del intervalo GAM específico están asociados con la tabla o índice
 - Asocia 4 tipos de páginas: *data*, *index*, *large object*, y *small-large objects*.
 - La cabecera cuenta con un puntero al inicio de un rango de 4 GB mapeado por la página IAM en el archivo de datos
 - Tiene un bit por cada *extent* indicando si pertenece o no al objeto asociado al IAM

Heap Tables

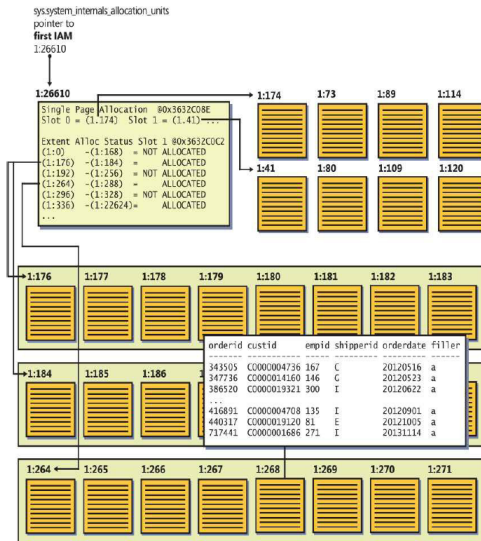


- Un *HEAP* está compuesto por una o más páginas de asignación de índices (IAM) que señalan las páginas de datos que constituyen el *heap*.
- La primera página disponible en un *heap* es la primera página que se encuentra en el archivo de la base de datos para ese *heap*.

```
select * from sys.system_internals_allocation_units
```

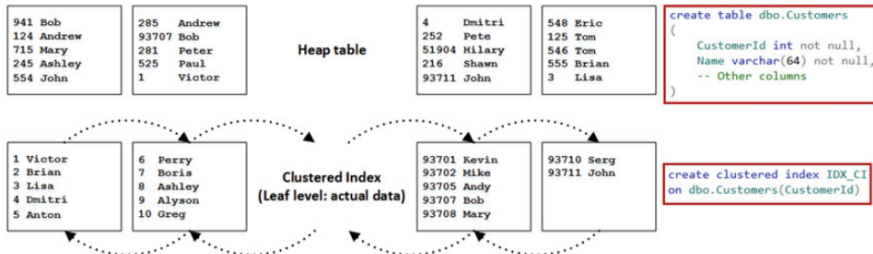
	allocation_unit_id	type	type_desc	container_id	filegroup_id	total_pages	used_pages	data_pages	root_page	first_page	first_iam_page
1	196608	1	IN_ROW_DATA	196608	1	33	26	24	0x9E00000000000000	0x1300000000000000	0x9C00000000000000
2	327680	1	IN_ROW_DATA	327680	1	13	6	4	0xA000000000000000	0x1100000000000000	0x8300000000000000
3	458752	1	IN_ROW_DATA	458752	1	12	7	5	0x9B00000000000000	0x1400000000000000	0x1500000000000000
4	524288	1	IN_ROW_DATA	524288	1	2	2	1	0x2000000000000000	0x2000000000000000	0x0C00000000000000
5	281474977103872	1	IN_ROW_DATA	281474977103872	1	0	0	0	0x0000000000000000	0x0000000000000000	0x0000000000000000
6	281474977300480	1	IN_ROW_DATA	281474977300480	1	0	0	0	0x0000000000000000	0x0000000000000000	0x0000000000000000

Heap Tables

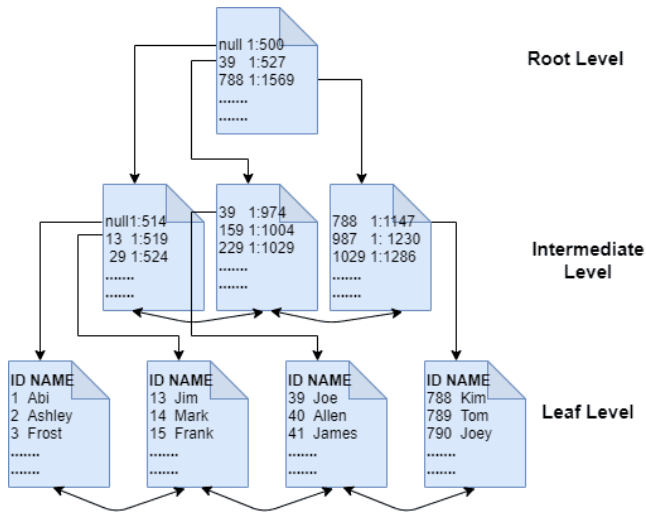


Clustered Indexes

- 1 Un *clustered index* dicta el **orden físico** de los datos en una tabla
- 2 Una tabla puede tener **sólo un *clustered index*** definido

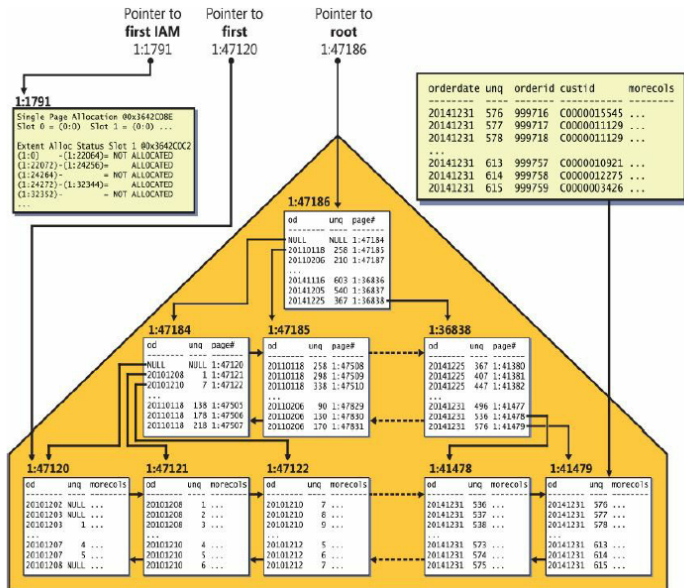


Árbol B



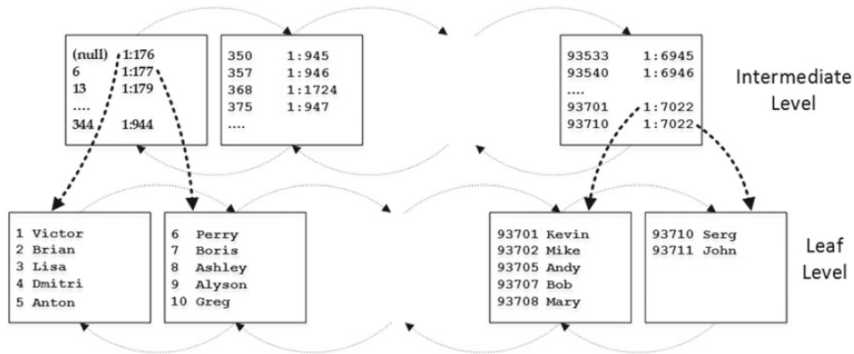
B-Tree Clustered Index on ID

Clustered Indexes

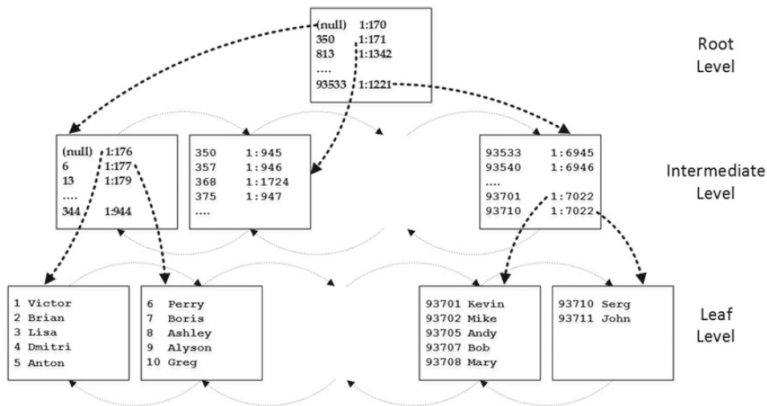


Clustered Indexes

- 1 El nivel intermedio almacena una fila por cada página del nivel de hoja.
- 2 Almacena: dirección física de la página y el valor mínimo de la clave del índice de la hoja referenciada, con excepción del primera fila de la primer página donde almacena NULL (optimización para insertar una fila con clave mas baja en la tabla)

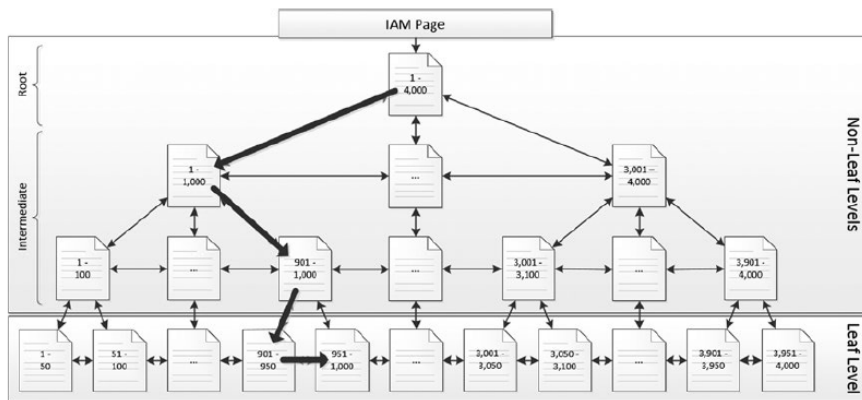


Clustered Indexes



Siempre hay un nivel hoja, cero o mas niveles intermedios y un nivel raíz. Excepto cuando la tabla entra en una única página, en este caso solo hay una página con los datos.

Clustered Indexes - buscando entre 925 y 3025



Non Clustered Indexes

- ❶ Non clustered indexes definen un orden que es almacenado en una estructura separada de los datos.
- ❷ Es similar al clustered index en su estructura, pero las páginas del nivel de hoja incluyen el valor de la clave y un *rowid*
- ❸ El ***rowid***
 - Para *heap tables* el rowid representa la locación física de la página
 - Para tablas con *clustered index* representa el *clustered index key* de la fila.
- ❹ Los nodos intermedios almacenan las dirección física del página y el valor mínimo de la clave.

NonClustered Indexes

```
SELECT * FROM dbo.Customers WHERE Name = 'Boris'
```

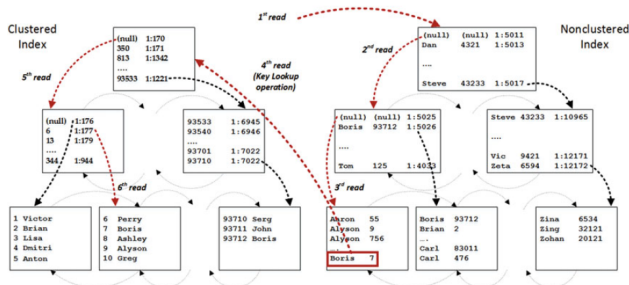


Figura: 1er Paso

NonClusterd Indexes

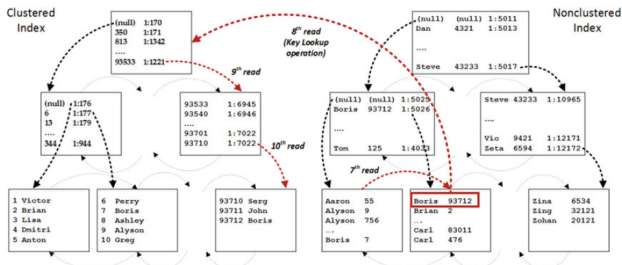
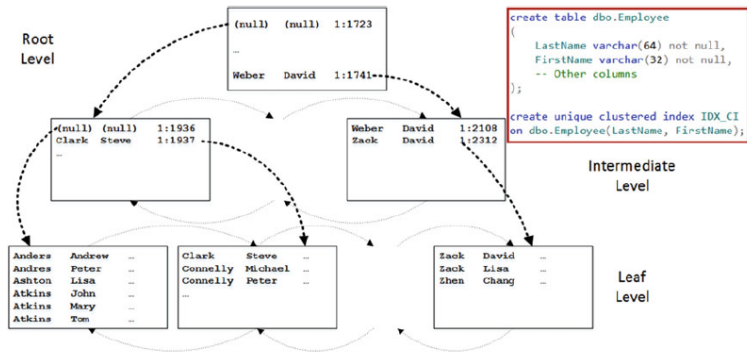
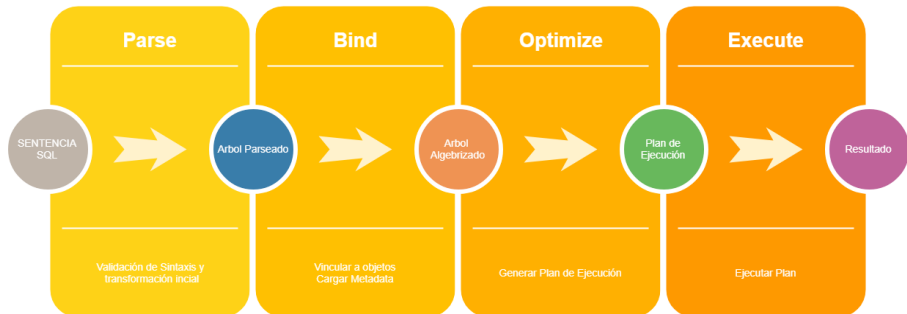


Figura: 2do Paso

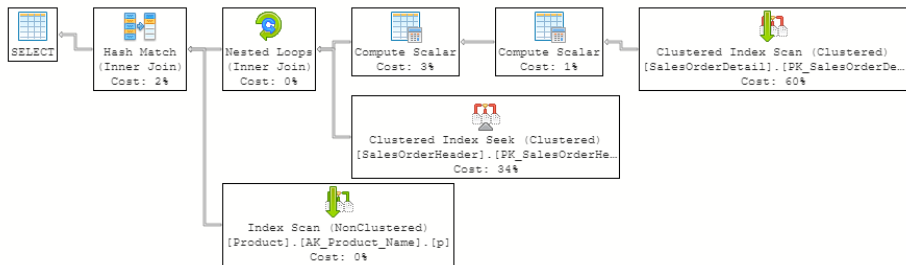
Composite Indexes



El procesamiento de consultas



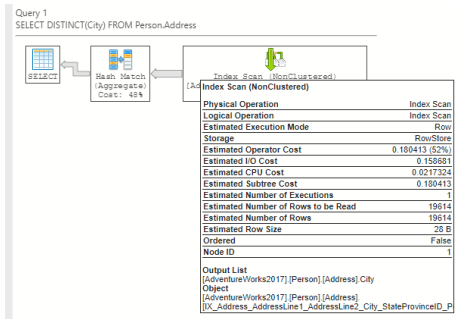
Ejecución de Consultas








Ejecución de Consultas

Cada nodo en un plan de ejecución implementa al menos tres métodos

- Open(): se inicializa el operador y se configuran las estructuras de datos requeridas
- GetRow() : Requiere una fila del operador
- Close() : finaliza el operador limpiando estructuras y datos que sean necesarios.



Operadores de Acceso a Datos

Estructura	Scan	Seek
Heap	 Table Scan	
Clustered Index	 Clustered Index Scan	 Clustered Index Seek
Nonclustered Index	 Index Scan	 Index Seek

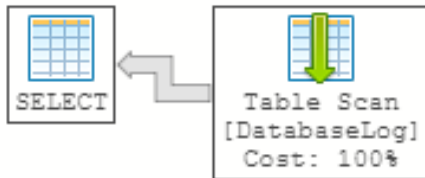
bookmark lookup: RID lookup or Keylookup





Table Scan

Query 1
`SELECT * FROM DatabaseLog`

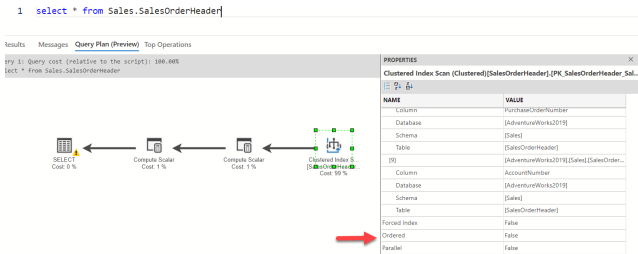


La tabla Databaselog no tiene índice clustered.



Cluster Index Scan

- Unordered Clustered Index Scan
- Ordered Clustered Index Scan





Comparemos el uso de índices

- ❶ `SELECT * FROM Person.Address ORDER BY AddressID;`
- ❷ `SELECT AddressID, City, StateProvinceID FROM Person.
Address ORDER by AddressID;`
- ❸ `SELECT AddressID, City, StateProvinceID FROM Person.
Address;`
- ❹ `SELECT AddressID, City, StateProvinceID FROM Person.
Address ORDER by City;`

Operadores de Agregación

SQL Server utiliza dos operadores:

- 1  Stream aggregate
- 2  Hash aggregate

Se usan para implementar agregados (SUM, AVG, MAX, etc) y para GROUP BY y DISTINCT.

Para algunos operadores se requiere que los datos vengan ordenados, el *Query Optimizer* puede usar un índice existente o puede introducir un *Sort Operator* explícitamente.

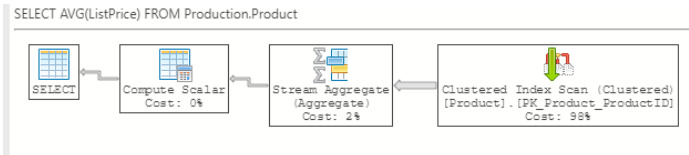
Tanto el hashing como el sort usan memoria, pero si no encuentran espacio suficiente pueden usar *tempdb database*.



Stream Aggregate

Consultas que usan un agregado y no tienen una clausula GROUP BY (***scalar aggregates***) siempre se implementan con Stream Aggregate. En caso de usar GROUP BY los datos deben venir ordenados.

```
SELECT AVG(ListPrice) FROM Production.Product
```

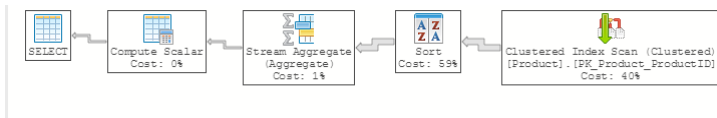


```
--Compute Scalar(DEFINE:([Expr1002]=CASE WHEN [Expr1003]=(0) THEN NULL ELSE [Expr1004]/CONVERT_IMPLICIT(money,[Expr1003],0) END  
--Stream Aggregate(DEFINE:([Expr1003]=Count(*), [Expr1004]=SUM([AdventureWorks2017].[Production].[Product].[ListPrice])))  
--Clustered Index Scan(OBJECT:([AdventureWorks2017].[Production].[Product].[PK_Product_ProductID]))
```



Stream Aggregate

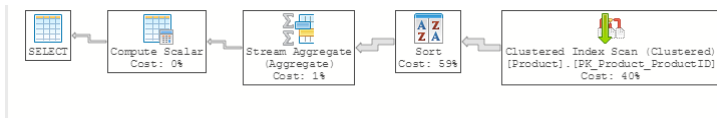
```
SELECT ProductLine, COUNT(*) FROM Production.Product  
GROUP BY ProductLine
```



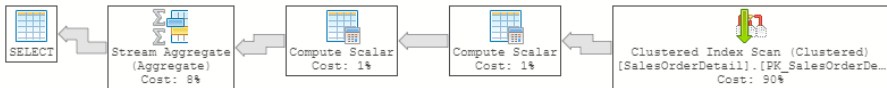


Stream Aggregate

```
SELECT ProductLine, COUNT(*) FROM Production.Product  
GROUP BY ProductLine
```



```
SELECT SalesOrderID, SUM(LineTotal) FROM Sales.  
SalesOrderDetail GROUP BY SalesOrderID
```





Hash Aggregate

Hash Match

El Hash Aggregate cómo el Hash Join se implementa con el operador físico

Hash Match



Hash Aggregate

Hash Match

El Hash Aggregate cómo el Hash Join se implementa con el operador físico

Hash Match

```
SELECT City, COUNT(City) AS CityCount
FROM   Person.Address
GROUP BY City
```

El optimizador de consultas puede seleccionar un *Hash Aggregate* para tablas grandes donde los datos no están ordenados, *no es necesario* ordenarlos y su cardinalidad se estima en solo unos pocos grupos.



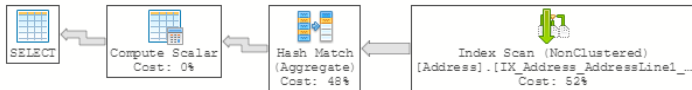
Hash Aggregate

Hash Match

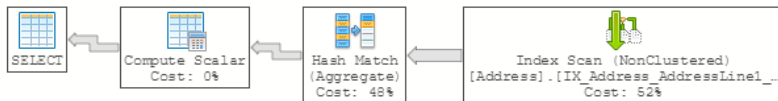
El Hash Aggregate cómo el Hash Join se implementa con el operador físico **Hash Match**

```
SELECT City, COUNT(City) AS CityCount
FROM Person.Address
GROUP BY City
```

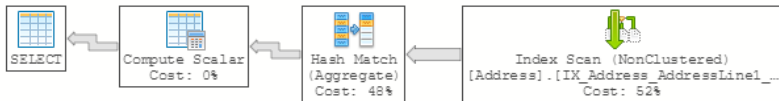
El optimizador de consultas puede seleccionar un *Hash Aggregate* para tablas grandes donde los datos no están ordenados, *no es necesario* ordenarlos y su cardinalidad se estima en solo unos pocos grupos.



Hash Keys Build



Hash Keys Build



```
<HashKeysBuild>  
  <ColumnReference Database="[AdventureWorks2017]" Schema="[Person]" Table="[Address]" Column="City">  
  </ColumnReference>  
</HashKeysBuild>
```

Hash vs Stream Aggregate

- *Hash Aggregate* ayuda cuando los datos no están ordenados. Si se crea un índice que pueda proporcionar datos ordenados, entonces el optimizador de consultas puede seleccionar un *Stream Aggregate*.
- Si la entrada no está ordenada pero se pide el orden explícitamente en una consulta, el optimizador de consultas puede introducir un *Sort operator* y un *Stream Aggregate* o puede decidir usar un *Hash Aggregate* y luego ordenar los resultados

Una consulta que usa la palabra clave **DISTINCT** puede ser implementada por *Stream Aggregate*, *Hash Aggregate* o por un operador *Distinct Sort*.

Distinct Sort

Una consulta que usa la palabra clave **DISTINCT** puede ser implementada por *Stream Aggregate*, *Hash Aggregate* o por un operador *Distinct Sort*.

```
SELECT DISTINCT(JobTitle) FROM HumanResources.Employee;
```

```
SELECT JobTitle FROM HumanResources.Employee  
GROUP BY JobTitle;
```

Query 1

```
SELECT JobTitle FROM HumanResources.Employee GROUP BY JobTitle
```



Juntas

El optimizador de consultas debe tomar dos decisiones importantes con respecto a las juntas:

- la selección de un orden de las juntas
- la selección del algoritmo de junta



- Nested Loops Join



- Merge Join

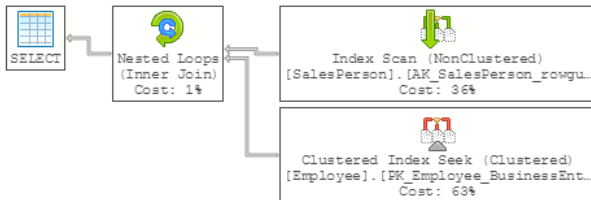


- Hash Join



Nested Loop Join

```
SELECT e.BusinessEntityID FROM HumanResources.Employee AS e
      INNER JOIN Sales.SalesPerson AS s ON
      e.BusinessEntityID = s.BusinessEntityID
```



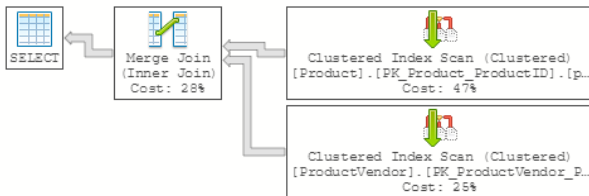
Es muy efectivo si el *outer input* es suficientemente pequeña y el *inner input* es grande pero indexada.

Merge Join y Hash Join requieren que la junta tenga al menos un predicado de junta basado sobre un operador de igualdad (ej. equi join)



Merge Join

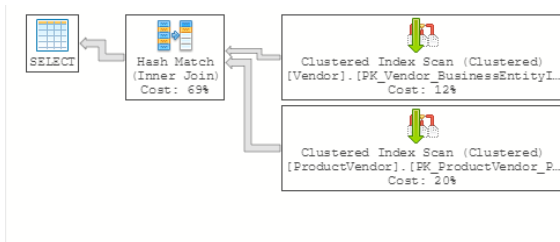
```
SELECT p.ProductID, v.BusinessEntityID
FROM Production.Product AS p
      JOIN Purchasing.ProductVendor AS v
      ON (p.ProductID = v.ProductID);
```





Hash Join

```
SELECT pv.ProductID, v.BusinessEntityID, v.Name
FROM Purchasing.ProductVendor pv JOIN Purchasing.Vendor v
ON (pv.BusinessEntityID = v.BusinessEntityID)
WHERE StandardPrice > 10
```



Hash Join

El optimizador lo usa para procesar entradas largas, no ordenadas, no indexadas eficientemente.

JOIN Resumen

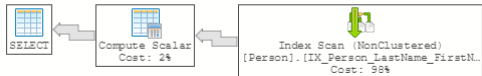
Join Type	Index on Joining Columns	Usual Size of Joining Tables	Presorted	Join Clause
Hash	Inner table: Not indexed Outer table: Optional Optimal condition: Small outer table, large inner table	Any	No	Equi-join
Merge	Both tables: Must Optimal condition: Clustered or covering index on both	Large	Yes	Equi-join
Nested loop	Inner table: Must Outer table: Preferable	Small	Optional	All



Compute Scalar

```
SELECT p.LastName + ', ' + p.FirstName  
FROM Person.Person as p
```

SELECT p.LastName + ', ' + p.FirstName FROM Person.Person as p



Results Top Operations Query Plan

4 RESULTS

	XML Showplan
1	<ShowPlanXML xmlns="http:...



Compute Scalar

```
<RelOp NodeId="0" PhysicalOp="Compute Scalar" LogicalOp="Compute Scalar" EstimateRows="19972" EstimateIO="0" EstimateCPU="0.0019972" AvgRowSize="113" EstimatedTotalSubtreeCost="0.104285" Parallel="0">
  <OutputList>
    <ColumnReference Column="Expr1001"></ColumnReference>
  </OutputList>
  <ComputeScalar>
    <DefinedValues>
      <DefinedValue>
        <ColumnReference Column="Expr1001"></ColumnReference>
        <ScalarOperator ScalarString="[AdventureWorks2017].[Person].[Person].[LastName] as [p].[LastName]+N&apos;; &apos;;+[AdventureWorks2017].[Person].[Person].[FirstName] as [p].[FirstName]">
          <Arithmetic Operation="ADD">
            <ScalarOperator>
              <Arithmetic Operation="ADD">
                <ScalarOperator>
                  <Identifier>
                    <ColumnReference Database="[AdventureWorks2017]" Schema="[Person]" Table="[Person]" Alias="[p]" Column="LastName"></ColumnReference>
                  </Identifier>
                </ScalarOperator>
                <ScalarOperator>
                  <Const ConstValue="N&apos;; &apos;;"></Const>
                </ScalarOperator>
              </Arithmetic>
            </ScalarOperator>
            <ScalarOperator>
              <Identifier>
                <ColumnReference Database="[AdventureWorks2017]" Schema="[Person]" Table="[Person]" Alias="[p]" Column="FirstName"></ColumnReference>
              </Identifier>
            </ScalarOperator>
          </Arithmetic>
        </ScalarOperator>
      </DefinedValue>
    </DefinedValues>
  </ComputeScalar>
</RelOp>
```



Compute Scalar

```
SELECT COUNT(*) cRows  
FROM HumanResources.Shift;
```



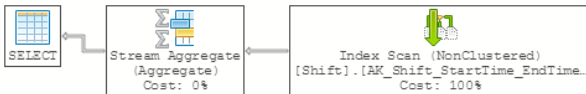


Compute Scalar

```
SELECT COUNT(*) cRows  
FROM HumanResources.Shift;
```



```
SELECT COUNT_BIG(*) cRows  
FROM HumanResources.Shift;
```





Filter

```
SELECT City, COUNT(City) AS CityCount
FROM Person.Address
GROUP BY City
HAVING COUNT(City) > 1
```



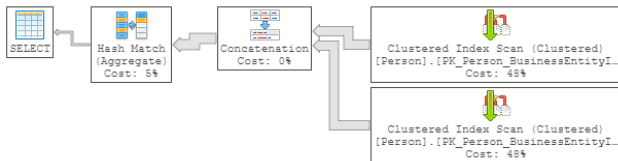


UNION - Concat

OPERADORES

Hay varios operadores que el optimizador utiliza para resolver la unión: merge, hash, concat

```
select Suffix from Person.Person
UNION
select Suffix from Person.Person
```



¿Que es más costoso UNION o UNION ALL?

Integridad Referencial

```
SELECT a.AddressID, sp.StateProvinceID
FROM Person.Address AS a
    JOIN Person.StateProvince AS sp
    ON a.StateProvinceID = sp.StateProvinceID
WHERE a.AddressID = 27234;
```

```
SELECT a.AddressID, a.StateProvinceID
FROM Person.Address AS a
    JOIN Person.StateProvince AS sp
    ON a.StateProvinceID = sp.StateProvinceID
WHERE a.AddressID = 27234;
```

```
ALTER TABLE Person.Address
DROP CONSTRAINT FK_Address_StateProvince_StateProvinceID;
```

```
ALTER TABLE Person.Address WITH CHECK
ADD CONSTRAINT FK_Address_StateProvince_StateProvinceID
    FOREIGN KEY(StateProvinceID)
REFERENCES Person.StateProvince (StateProvinceID);
```

```
SELECT ...  
OPTION (<hint>,<hint>...)
```

- HASH GROUP / ORDER GROUP
- MERGE —HASH —CONCAT UNION
- LOOP—MERGE—HASH JOIN
- FAST N
- FORCE ORDER
- RECOMPILE
- FROM TableName WITH (INDEX ([IndexName]))

Parameter Sniffing / OPTIMIZE FOR

```
SELECT * FROM Person.Address  
WHERE City = 'Mentor';
```

```
SELECT * FROM Person.Address  
WHERE City = 'London';
```

Parameter Sniffing / OPTIMIZE FOR

```
SELECT * FROM Person.Address  
WHERE City = 'Mentor';
```

```
SELECT * FROM Person.Address  
WHERE City = 'London';
```

```
DECLARE @City NVARCHAR(30)  
SET @City = 'Mentor'  
SELECT * FROM Person.Address  
WHERE City = @City
```

```
SET @City = 'London'  
SELECT * FROM Person.Address  
WHERE City = @City;
```

```
OPTION( OPTIMIZE FOR (@City = 'Mentor') )
```

Estadísticas

```
SET STATISTICS TIME ON;
```

```
SELECT soh.AccountNumber, sod.LineTotal,  
sod.OrderQty, sod.UnitPrice, p.Name  
FROM Sales.SalesOrderHeader soh  
JOIN Sales.SalesOrderDetail sod  
ON soh.SalesOrderID = sod.SalesOrderID  
JOIN Production.Product p  
ON sod.ProductID = p.ProductID  
WHERE sod.LineTotal > 1000 ;
```

```
SET STATISTICS TIME OFF;
```

Liberar cache:

```
DBCC FREEPROCCACHE
```

Ver Entradas/Salidas

```
SET STATISTICS IO ON;
```

```
SELECT soh.AccountNumber, sod.LineTotal,  
       sod.OrderQty, sod.UnitPrice, p.Name  
FROM Sales.SalesOrderHeader soh  
JOIN Sales.SalesOrderDetail sod  
     ON soh.SalesOrderID = sod.SalesOrderID  
JOIN Production.Product p  
     ON sod.ProductID = p.ProductID  
WHERE sod.LineTotal > 1000 ;
```

```
SET STATISTICS IO OFF;
```

Liberar cache:

```
DBCC dropcleanbuffers  
DBCC freeproccache
```

SARGable / search-ARGument-able

Comparar

```
SELECT sod.*  
FROM Sales.SalesOrderDetail AS sod  
WHERE sod.SalesOrderID IN ( 51825, 51826, 51827, 51828 );
```

```
SELECT sod.*  
FROM Sales.SalesOrderDetail AS sod  
WHERE sod.SalesOrderID BETWEEN 51825 AND 51828 ;
```

```
SELECT sod.*  
FROM Sales.SalesOrderDetail AS sod  
WHERE sod.SalesOrderID = 51825  
      OR sod.SalesOrderID = 51826  
      OR sod.SalesOrderID = 51827  
      OR sod.SalesOrderID = 51828;
```

Started executing query at Line 1

(150 rows affected)

Table 'SalesOrderDetail'. Scan count 4, logical reads 18, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

(150 rows affected)

Table 'SalesOrderDetail'. Scan count 1, logical reads 6, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Total execution time: 00:00:00.021

Composite Indexes: SARGable

SARGable predicates	Non-SARGable predicates
<code>LastName = 'Clark' and FirstName = 'Steve'</code>	<code>LastName <> 'Clark' and FirstName = 'Steve'</code>
<code>LastName = 'Clark' and FirstName <> 'Steve'</code>	<code>LastName LIKE '%ar%' and FirstName = 'Steve'</code>
<code>LastName = 'Clark'</code>	<code>FirstName = 'Steve'</code>
<code>LastName LIKE 'Cl%'</code>	

La *SARGability* de un índice compuesto depende de la *SARGability* del predicado sobre la primer columna del índice.


```
SELECT *  
FROM Purchasing.PurchaseOrderHeader AS poh  
WHERE poh.PurchaseOrderID * 2 = 3400;
```

```
SELECT *  
FROM Purchasing.PurchaseOrderHeader AS poh  
WHERE poh.PurchaseOrderID * 2 = 3400;
```

```
SELECT *  
FROM Purchasing.PurchaseOrderHeader AS poh  
WHERE poh.PurchaseOrderID = 3400/2;
```

```
SELECT a.PostalCode  
FROM Person.Address AS a  
WHERE a.StateProvinceID = 42;
```

```
CREATE NONCLUSTERED INDEX IX_Address_StateProvinceID  
ON Person.Address(StateProvinceID ASC)  
INCLUDE(PostalCode)  
WITH (DROP_EXISTING = ON);
```

Indice con filtros

```
SELECT soh.PurchaseOrderNumber ,  
       soh.OrderDate ,  
       soh.ShipDate ,  
       soh.SalesPersonID  
FROM Sales.SalesOrderHeader AS soh  
WHERE PurchaseOrderNumber LIKE 'P05%'  
AND soh.SalesPersonID IS NOT NULL;
```

```
CREATE NONCLUSTERED INDEX IX_Test  
ON Sales.SalesOrderHeader (PurchaseOrderNumber ,SalesPersonID)  
INCLUDE (OrderDate ,ShipDate)
```

```
CREATE NONCLUSTERED INDEX IX_Test  
ON Sales.SalesOrderHeader (PurchaseOrderNumber ,SalesPersonID)  
INCLUDE (OrderDate ,ShipDate)  
WHERE PurchaseOrderNumber IS NOT NULL  
AND SalesPersonID IS NOT NULL  
WITH (DROP_EXISTING = ON);
```

- Grant Fritchey. **SQL Server 2017 Query Performance Tuning Troubleshoot and Optimize Query Performance** (Fifth Edition). Apress, Berkely, CA, USA.
- Jason Strate and Grant Fritchey. 2015. **Expert Performance Indexing in SQL Server** (2nd ed.). Apress, Berkely, CA, USA.
- Benjamin Nevarez **Inside the SQL Server Query Optimizer**
- Grant Fritchey 2012 **SQL Server Execution Plans** -Second Edition Simple Talk Publishing September