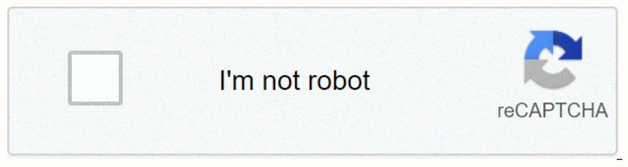

Continue

Inner join vs subquery performance sql server

I was reading at Stackoverflow today and saw an interesting question. SQL inner join vs subquery. The user has compared 3 requeries and wondered why the first took significantly longer than the other 2. Here are the requests: Query 1: SELECT * From TabA INNER JOIN TabA.Id = TabB.Id Query 2: select * From Tabbed where id IN (SELECT ID FROM TabB) Query 3: SELECT TabA.* From TabA INNER JOIN TabB ON TabA.Id = TabB.Id I reply was that the Query 1 will pull all the data from both table and therefore did more it. I also suggested that the user PUT STATISTICS SET STATISTICS IO AND SET TIME STATISTICS TO THE THEORY. Well, I got curious and decided to run a similar test for myself. No tables are OPs I decided to use 2 from AdventureWorks2008. Sales.Customers and People.Person. So first turned on the statistics. SET STATISTICS IO ON SET TIME STATISTICS ON Next, here are the requests using the same format as their OPS. SELECT * FROM Sales.Customer INNER join person. Person on Sales.Customer.PersonID = Person.Person.BusinessEntityID SELECT * FROM Sales.Customer.Customer.PersonID IN (SELECT Person.BusinessEntityID FROM Person.Person) SELECT Sales.Customer.* FROM Sales.Customer INNER join.Person on Sales.Customer.PersonID = Person.BusinessEntityID I am summing down the output for the sake of notice. -- Query 1 (1919 row(s) affected) Table 'Worktable'. Scan counts 0, his logic 0, his physical 0, his-front 0, his logic 0, loses his physical 0, his blob 0, his blob-him 0. 'People' table. Scan counts 1, his logic 3816, his physical 0, his-front 0, his logical 0, his physical command 0, his blob 0, loses it – ahead of him 0. 'Clients' table. Scan counts 1, his logic 123, his physical 0, his-front 0, his logic 0, his logic 0, his blob 0, his blob 0, his blob-it 0. SQL Server Execution Times: Time CPU = 1594 ms, That pass time = 11237 ms. -- Query 2 (19119 row(s) affected) Table 'Worktable'. Scan counts 0, his logic 0, his physical 0, his-front 0, his logic 0, loses his physical 0, his blob 0, his blob-him 0. 'Clients' table. Scan counts 1, his logic 123, his physical 0, his-front 0, his logic 0, his logic 0, his blob 0, his blob 0, his blob-it 0. 'People' table. Scan counts 1, his logic 59, his physical 0, his-front 0, his logic 0, loses his physical 0, his blob 0, his blob-him 0. SQL Server Execution Times: Time CPU = 859 ms, Time Spent = 2193 ms. -- Query 3 (1919 row(s) affected) Table 'Worktable'. Scan counts 0, his logic 0, his physical 0, his-front 0, his logic 0, his logic 0, loses his physical 0, his blob 0, his blob-him 0. 'People' table. Scan counts 1, his logic 59, his physical 0, his-front 0, his logic 0, loses his physical 0, his blob 0, his blob-him 0. 'Clients' table. Scan counts 1, his logic 123, his physical 0, his-front 0, his logic 0, his logic 0, his blob 0, his blob 0, his blob-it 0. SQL, SR Time Executions: Time CPU = 875 ms, Past Time = 2796 ms. If you review IO section you'll notice that all it's physical are 0. This means both tables are hidden at the moment and all it's coming from memory. So right in the fight we can exclude caching problems. Also Queries 2 and 3 are close enough to time to be just normal variations, and the IO section while in a different order has identical numbers. The research plans are also very similar. Strange enough though all the processes in these research plans are the same, rates are different. Unfortunately I don't know enough about research plans to really tell why. Query 1 has a CPU time nearly doubled the other two querys and a past time of 4-5 times greater. Well we hope that comes from the original question. The Client Table has 123 logic reads for all three requests. However if you compare it the logic for the table to whom you will see 59 each for Queries 2 & 3 and 3816 for Query 1! That's a lot of it extra! So now let's compare the search plans If you look good, you'll notice that plans are very similar. The only difference (other than these) is the Clustered Index Scan for Individuals. Person on Query 1 and scan the index on Queries 2 and 3. Scan of Clustered Index makes perfect sense to me. The etched index contains all the data for the table in the leaf nodes and all the data that they returned. So it makes sense to scan across the whole table, returning everything. It took me a moment to figure out why Queries 2 and 3 are doing an Index scan on AK_Person_rowguid. (And honestly if anyone would like to confirm this I would greatly appreciate it.) This particular index is on a column named Rowgid that is a unique identifier and does not reference anywhere in the query. So why use it? My understanding is that this index in particular is being used for these two reasons. First this is the smallest index on the table, the other two being the clustered index and an index on the first, last and middle names. The second (and the real key here) is that the embedded index key has to be in the sheet nudes of each non-inclusion index. Suppose a etched index index exists of course. In this case in particular the clustered index is on BusinessEntityID which is the value we need for our search. Thus, the optimizer decides that it would speed up it in the entire non-clusters index, pulling the BusinessEntityID from the leaf clouds, rather than using the index nose in the index that is seized. Now I'm not sure the situation from the post I've seen is the same. It relies on the indexes that are available, and I don't have that information. But I think that's a pretty good indication of what happened. If nothing else matters, for me at least, it was a good exercise in reading the research plans and trying to figure out exactly what was going on. Join is faster than subquery. grant made for busy access, thinking of hard disk read-write needle (top?) that goes back and forth when accessing: User, SearchExpression, PageSize, DrilldownPageSize, User, SearchExpression, PageSize, DrilldownPageSize, User... and so on. Joining tasks does not focus the operation on the result of the first two tables, any subsequent join would focus joining in on the In-Memory (or cache of that cache) result in the tables first joining, and so on. Less read-write needle movement, so Faster Source: Here's one of the challenges of writing SQL queries will choose whether to use a grant or a JOIN. There are many situations in which a JOIN is the best solution, and there are others where a subquery is the best. Consider this topic in detail. Subqueries are used in complex SQL queries. Generally, there is a primary search outside and one or more nest subqueries in the outdoor search. Subqueries can be simple or corrected. Simple subqueries don't rely on the columns in the outside query, whereas correcting subqueries refers to data from the outside query. You can learn more about subqueries in the SQL Subqueries article by Maria Alcaraz. The JOIN clause does not include additional requests. It connects two or more tables and selects data from them in a single result range. It is most commonly used to join tables with primary and foreign keys. You can read more about JOINs in the article How to Practice SQL Joins by Emil Drkušić. Subqueries and JOINs can be both used in a complex query to select data from multiple tables, but they do so in different ways. Sometimes you have a choice of either, but there may be in which a subquery is the only real option. We will describe the various scenarios below. Consider two simple tables, products and retail, which we will use in our examples. Here is the product table. idnamecostyearcity 1chair245.002017Chicago 2armchair500.002018Chicago 3desk900.002019Los Angeles 4mp85.002017Cleveland5bench2000.002018Seattle 6stool2500.002020Austin 7tv table2000.00202020Austin This table contains the following column: The identifier: the identifier of the product.name: the product name. price: The price of the product. year: the product the year was done. city: the city in which the product was made. And the other table, sold: idproduct_idpriceyearcity 12200.002020Chicago 22590.00202020New York 32790.002020Cleveland 53800.00201 64100.00202020Detroit 752300.002019Seattle 872000.00202020New York these columns: Id: The identifier of the sale. product_id: The identifier of the retail product. price: the sale price. year: the year in which the product was sold. city: the city where the product was sold. We will use these two tables to write complex requests and subqueries and JOINs. When recovering Subqueries and JoiNs SQL beginners often use subqueries when the same results can be achieved with JoiNs. While subqueries may be easier to understand and use for many SQL users, JOINs are often more efficient. JOINs are also easier to read as demand becomes more complex. So we're focused first on when you can a grant with a Join for better efficiency and readability. Scalar Subquery First this case is the stair subquery. A staircase subquery returns a single value (one column and one row) to be used by the external query. Here is an example. Supposing we need the names and costs of the products that have been sold for $2,000. Let's look at the code with a subquery: SELECT NAME, Price from Product WHERE id = (SELECT product_id FROM Sale WHERE PRICE = 2000 AND product_id = pwodwi.id); and the result: nonkost armchair500.00 TV table2000.00 The outdoor query selects the names (names) and the cost (price) of the products. Since we don't want all the products, we use a WHERE clause to filter the rows in the product ID back by the subquery. Now let's look at the grant. The retail table contains sales records of products. The first subquery filters the records only those with the sale price equal to $2,000 (price=2000). It then uses the product ID (product_id) of the selected sales to identify the records from the product table (product_id = pwodwi.id). This is a correlation subquery, since the second condition of the subquery references a column in the outside query. Only two products were sold at $2,000: the arm and the TV table. This research is not very effective. How should we edit it? We can build a JOIN structure and get the same result. Look at the query with a JOIN: SELECT p.name, p.cost from Pin product sold s ON p.id s.product_id = WHERE s.price = 2000; In this query, we connect the two product tables and sale with a JOIN operator. In the JOIN condition, the records from the product table are linked to the records from the Sale table of the product ID. In the end, rows are filtered by a WHERE clause to select the record when the sale price of the product equals $2,000. Subquery in the clause in Clause another subsidy that is easily replaced by a JOIN is one of the use of an operator IN. In this case, the subquery returns to the external query a list of values. Let's say we want to get the names and costs of the products sold in our example. SELECT NAME, PRICE FROM PRODUCT WHERE ID IN (SELECT product_id from Sale); The query outside selects the names and the costs of the products; it then filters through the files containing product ID that bear parts on the list returned by the subquery. The subquery selects the product ID from the sale table (select product_id from sale), so the only products sold are returned by this research in the final set, like this: nonkost armchair5 million lanp85.000 bench2000.00 Desk9000.00 Has more products in the product table, but only four of them have been sold. The query below returns the same result using a JOIN: Select distinct p.name, p.cost from product PIN Sold s ON s.product_id = p.id; It becomes a very simple search. It connects the two tables by product ID and selects the names and expenses of these products. It's an INNER join, so if a product doesn't have its ID in the sale table, it won't Note that we also use the DISTINCT keyword to remove duplicate records. This is often necessary if you transform subqueries with an IN OR AN IN INTO JOINs. Want to learn more about SQL Subqueries and the IN operator? Watch an episode of us learn SQL series on Youtube. Remember to subscribe to our channel. This is just like the previous situation, but here is the subquery used in a NOT IN operator. We want to choose the names and expenses of the products that haven't been sold. Below is an example with a subquery inside the OPERATOR NOT IN: SELECT NAME, Price from Product WHERE ID IS NOT IN (SELECT product_id from Sale); The results: Chairman of noncost chair245.00 stool25 million The Subquery returns the product ID from the retail table (products are sold) and compares them with the product ID of the external query. If a file in the outside query does not find its product ID in the list returned by the subquery, they return the file. How do you rewrite this grant with a JOIN? You can do it as this: Select Distinct p.name, p.cost from Retail PFT product S ON s.product_id = p.id WHERE S.PRODUCT_ID IS NULL; This query connects the two product tables and sold by the product ID. You should also use the DISTINCT keyword, like we did when we transformed the subquery before and an IN into a JOIN. Note that in the grant recruits to PA, we used a left merge with a WHERE. In this way, you start with all the products including those not sold, then select only the NULL records in the column product_id. the denote NULL that the product has not been sold. Correcting subqueries to EXIST AND IN DOES NOT EXIST THE subqueries OF AN EXISTS OR WITHIN AN EXISTS THEY ALSO EASILY REWRITE WITH Joins. The research below uses a grant to get the details on products that were not sold in 2020. SELECT NAME, PRICE, CITY FROM PRODUCT WHERE NOT EXISTS (SELECT ID FROM RETAIL WHERE YEAR = 2020 AND product_id = product.id); The result: noncostity president245.00Chicago desk900.00Los Angeles bench2000.00Seattle stool2500.00Austin Per product of the outward research, the subquery selecting the records that have yearly selling is 2020 (year=2020). If there is no record for a given product in the subquery, there does not exist the clause returns true. The result set contains the products and the year sold other than 2020 as well as the products without any records in the retail table. You can rewrite the same query using a JOIN: select p.name, p.cost, p.city from product p left PFT JOIN s on s.product_id = p.id WHERE s.year! = 2020 OR s.year IS NULL; Here, we link the product table and the sale table to a LEFT JOIN operator. This allows us to include the products that have never been sold in set of results. WHERE THE clause filters the records by selecting the products with no records in the sale table (s.year IS NULL) as well as the products and the sale year other than 2020 (s.year!=2020). When you can't replace a subquery with a Joins can be effective, but there are situations that yon sibvansyon epi yo pa yon Antre nan. Anba la a se kek nan sitiyasyon sa yo. Premye a sa yo se yon subquery nan yon ki soti nan kloz le li sevi avek yon GWOUP pa kalkile vale total. Let's look at the following example: SELECT city, sum_price FROM (SELECT city, SUM(price) AS sum_price FROM sale GROUP BY city) AS s WHERE sum_price < 2100; and the result: citysum_price Chicago2000.00 Detroit100.00 Cleveland1590.00 Here, the subquery selects the cities and calculates the sum of the sale prices by city. The sum of all sale prices in each city from the sale table is calculated by the aggregate function SUM(). Using the results of the subquery, the outer query selects only the cities whose total sale price is less than $2,100 (WHERE sum_price < 2100). You should remember from previous lessons how to use aliases for subqueries and how to select an aggregate value in an outer query. Subquery Returning an Aggregate Value in a WHERE Clause Another situation in which you cannot rewrite a subquery structure with a JOIN is an aggregate value being compared in a WHERE clause. Look at this example: SELECT name FROM product WHERE cost The result: name chair armchair desk lamp This query retrieves the names of the products whose costs are lower than the average sale price. The average sale price is calculated with the help of the aggregate function AVG() and is returned by the subquery. The cost of each product is compared to this value in the outer query. Subquery in an ALL Clause Yet another situation is a subquery with an ALL clause. SELECT name FROM product WHERE cost > ALL(SELECT price from sale); Subquery a retounen tout pri yo vann nan tablo a vann. Reket la deyo retounen non an nan pwodwi a ak pri a vann pi wo pase pri a. Rezilta a: Le yo sevi ak yon subquery vs. yon JOIN Nou te revize kek itilizasyon komen nan subqueries ak sitiyasyon yo nan ki kek subqueries ta ka rekrite ak JOINs olye. Yon JOIN se pi efikas nan pifo ka yo, men gen ka nan ki konstwi lot pase yon subquery se pa posib. Pandan ke subqueries ka plis lizib pou débutan, JOINs yo pi lizib pou kod SQL ki gen eksperyans kom requtes yo vin pi konpleks. Li se yon bon pratik pou fe pou evite plizye nivo nan subqueries nich, depi yo pa fasil lizib epi yo pa gen bon pefomans. An jeneral, li se pi bon yo ekri yon rechech ak JOINs olye ke ak subqueries si sa posib, espesyalman si subqueries yo korelasyon. Si ou enterese nan aprann plis oswa si ou vle pratike ladres ou, tcheke deyo seksyon yo subqueries nan la SQL Basics kou oswa nan SQL Pratike Mete kou. Kou.