UNIT 13 STRUCTURED QUERY FORMULATION

Structure

13.0 OBJECTIVES

After reading this Unit you will be able to:

- know the concept of Structured Query Formulation(SUF);
- understand the methods of creating queries in a structured manner; and
- acquaint yourself with the different methods and techniques of query optimization.

13.1 INTRODUCTION

Structured Query Formulation (SQF) is the method of creating queries in a structured manner in order to get the results accurately and efficiently. Every database query statement works on a database, and requires reading and writing of database records

or data blocks. It is necessary to understand how these queries work, and how many read and write is involved with each query, etc. It is possible to get the same result by reducing the read/wrote operation, by changing the query a little differently sometimes or by formulating the query in a more structured manner. The procedure of analyzing a query and converting it into another query statement of the same meaning is known as query optimization. Structured query formulation is possible by understanding the different methods and techniques of query optimization. A structured query is a result of a query optimization procedure on a non-structured query. This unit will help you understand query optimization and hence to write more efficient and structured queries.

Database queries have been written by using any query language. The flexibility, user friendliness and easiness of this language that one should only know what is the result expected after executing the query. And one need not know how the data is arranged, how the query is executed in terms of reads and writes of database blocks, etc. As a result the way the query is written, though it gives the result, it need not be efficient and fast in executing the query. It is very important to know how the query works and how a more optimized and efficient query of the same meaning can be written. Query optimization means to convert a query Q to its semantically equivalent query Q', which evaluates faster than Q. The newer Database Management Systems (DBMSs) provide/support declarative query languages, using such a query language. Users express their queries in terms of "what" they want rather "how" to obtain it; thus, the extra burden of finding a good access path and an efficient evaluation strategy is left to the DBMS.

Query optimization is concerned with the techniques used by a DBMS to reduce the execution time of a query. All of the major database vendors now include optimising SQL compilers which analyses the SQL query sent to it, rewrites the query if necessary, and finally produces an optimal access plan to retrieve the data from the database. This module of SQL compiler is called Query Optimizer. This optimization is done based on the different optimization rules formulated on the basis of the cost of each operation on the query. However, regardless of the improvements made in query compiler technology, the implementer will have a much better knowledge of how the database is constructed and how the applications interact with this database. In addition, optimisers can and often do make the wrong decision.

While SQL optimisation is a very important part of the process of finely tuning a database application, it should not be considered in isolation since there are a number of other areas which also require careful attention from the implementer. Some examples of areas which should also be scrutinised are: overall database design and structure, indexing, table space structures and types, database configuration parameters, hardware issues like memory, processor speed, and disk types and so on.

This unit deals with the cost implication of each query operation and how to optimize the total cost of a query. It also gives some optimization strategies to be followed. To understand these optimization methods, it is required to know the relation statistics.

13.2 RELATION STATISTICS

When we try to determine if one query will be evaluated faster than another, it depends mainly on the number of disk block reads and writes. A block is the smallest

amount of data that the disk hardware can read. A block, in general, is 1024 bytes. So a single block usually contains quite a few tuples of a database. Generally the disk I/O is slower than memory I/O. Relations are normally large and cannot reside entirely in memory; they must be read in and write out of memory during a query. Also, a database query will usually perform a very simple in-memory computation, So in general we can completely ignore the in-memory cost of a query.

It is necessary to know some important statistical parameters of a query for analyzing the same. For a relation, R, the following statistics are required to optimise the query.

card *(R)*	the cardinality or number of tuples in *R*
degree*(R)*	the number of attributes in *R*
blocks*(R)*	the number of blocks that R occupies on disk
values*(R, A)*	the number of different values for attribute *A* in relation *R*. We will assume that the *A* values are uniformly distributed in *R*, that is every different value appears in the same number of tuples
clusters*(R, A)*	if R is *clustered*, the size of a cluster, for a single value for attribute *A*, is the same as blocks*(R)*/values*(R, A)* (assuming that the values are uniformly distributed)

Consider a student relation with name and IDNo. And suppose the information is stored in blocks as below.

Block Number	1	2	3
Tuples	Rishi Gupta, AIS02001, Sunil Satpal, AIS02008, Deepika Chaturvedi,AIS01004	Sunil Satpal, AIS03008, Deepika Chaturvedi,AIS02004 Rishi Gupta, AIS01001	Deepika Chaturvedi, AIS03004 Rishi Gupta, AIS03001 Sunil Satpal, AIS03008,

This relation has the following statistics
card(*student)* = 9
degree(student) = 2
blocks(student) = 3
values(student, name) = 3
clusters(student, name) = 1, since the relation is un-clustered and un-indexed on name.

When the relation is indexed on name, the tuples in the relation with the same *name* attribute are stored in the same or adjacent blocks. In other words, the same values are clustered together in the same area on disk.

Block Number	1	2	3
Tuples	Deepika Chaturvedi,AIS03004 Deepika Chaturvedi,AIS02004 Deepika Chaturvedi,AIS01004	Rishi Gupta, AIS02001, Rishi Gupta, AIS01001 Rishi Gupta, AIS03001	Sunil Satpal, AIS02008, Sunil Satpal, AIS03008, Sunil Satpal, AIS03008,

R is *indexed* on attribute *Name*. An index is a special data structure that permits rapid search for tuples on a specific attribute (Name). Without an index, on average, we would have to read half of the relation (block by block) to find a tuple with a particular attribute value. But with an index we can go directly to the block or blocks that contains the desired tuple. It is also easier to find all the tuples with the same

attribute since all of them are clustered in the same location. Indexing also involve some initial cost. But it can be ignored compared to the cost involved in searching.

There are two types of indexes—clustered index, and non-clustered index. In clustered index the data is physically sorted, while a non-clustered index is a separate index structure independent of the physical sort order of the data in the table.

13.3 NEED FOR STRUCTURED QUERY FORMULATION

Consider the following query: "*Get names of suppliers who supply part P2*". A possible SQL formulation for this query is:

```
SELECT DISTINCT sname
FROM supplier, supply
WHERE supplier.sno = supply.sno AND supply.pno = 'P2'
```

Suppose also that card(supplier) = 100, and card(Supply) = 10000, where card(R) denotes the cardinality of relation R, i.e. the number of rows in R. Suppose also that 50 rows in supply are for the part P2. If we follow a top-down execution of this query, then the following sequence of steps have to be performed:

1) First we must perform a cartesian product of supplier and supply. This means that we must perform 100*10,000=1,000,000 read operations from the secondary storage. Furthermore, as the resulting relation has cardinality of 1,000,000 it is too large to be kept in the main memory and thus it should be written back to the secondary memory.

   Note that the most expensive operations in a computer are I/O operations i.e. to read data from secondary storage to main memory or to write data from the main memory to secondary memory. Thus, a good optimizer should try to minimize the number of I/Os to main and from main memory

2) Read 1,000,000 tuples from secondary storage back to main memory and apply the section condition. This results in a relation of cardinality 50, which can be kept in the main memory.

3) Project the resulting relation of step 2 on the attribute *sname*. The described execution sequence results in 3,000,000 I/Os.

Now consider the following query:

```
SELECT DISTINCT sname
FROM supplier, (SELECT sno AS sup_sno
                FROM supply
                WHERE supply.pno = 'P2')
WHERE supplier.sno = sup_sno
```

It is an equivalent SQL expression to the one given above; however, the execution sequence of this SQL expression is much more efficient than the former one. To understand why, consider its execution sequence:

1) Select from the supply relation all the tuples for part P2. This involves reading 10,000 tuples and produces an intermediate result that is of cardinality 50 which can be easily held in the main memory.

2) Join the result of Step 1 to relation supplier. This involves the retrieval of 100 tuples which in the worst case means (if the main memory is not that large) that they have to be read 50 times one for each tuples of the result of Step 1, i.e. 5000 read operations. The cardinality of the resulting relation will be 50 again and it can be held in the main memory.

3) Project the resulting relation of Step 2 on the attribute *sname*

Thus, even in the worst case the second execution sequence will require in order to produce the result at most 15,000 I/O operations, which means that it is approximately 200 times more efficient than the first execution sequence. Moreover, an index defined on the pno attribute of the supply table can improve the execution of the query ever further.

13.4 PHYSICAL VS. LOGICAL OPTIMIZATION FOR SQF

The term *physical optimization* refers to the "physical"access methods used to navigate in a database. It involves techniques and strategies such as indexes, sorts, hash tables, etc. to find the shortest path to the required data. The term logical optimization is concerned with replacing a query placed by a user with an equivalent optimal query.

13.5 GENERAL STRATEGIES FOR OPTIMIZATION

a) ***Set the optimization level:*** Query optimization can be a relatively expensive operation especially if the query itself is very simple and returns few result rows. In fact the optimization process can occasionally be longer than it would have taken to execute the non-optimized query in the first place. Some database implementations allow the optimization level to be configured on per query or per connection basis others only allow this to be done on a database-wide level if at all.

b) ***Update the database statistics:*** Before we start optimizing queries, ensure that the relevant database statistics are up to date in the test or development environment. If these statistics are not up to date then efforts to optimise a query are futile since the database will not be operating under normal conditions and the query will produce misleading results.

c) ***Test environment should be similar to production environment:*** We should also consider very carefully the differences between the production or customer environments and the test environment. Typically the production environment will contain much more data than a test environment. Make sure that the test environment reflects the production environment as closely as possible. If this can't be done in terms of quantities of data, it should ensure that the distribution of data is roughly the same as a production database.

d) ***Know the data :*** It is important to know and understand the data which is stored in the database. This can help in choosing the most efficient selection criteria.

e) ***Reduce the communication overhead:*** The communication between the database and application is a critical part of query execution. This is especially true where the data must travel over any form of network which in today's computing environments means the vast majority of cases. Cutting down on the amount of data which must be sent over the network will reduce total query times.

One commonly occurring situation is where the application itself performs filtering or other types of processing designed to reduce the amount of information presented to the final user. This wastes database resources because extra rows must be returned. It wastes server resources because of the extra processing required. Finally it wastes communication resources because of the extra data which must be sent between the database and the application. Always try to extract only the data which is actually required by the application to perform the current task. Even if this increases the database resources required, it will almost certainly be more efficient than increasing the communication costs by sending unnecessary data.

Another example of wasting communication bandwidth is by using SELECT* to return all of the columns for each row selected. In terms of communication costs it will always be more efficient to request only those columns actually required by the application.

Sometimes a chain of queries are used to achieve a final result (the result of each query is used to provide a set of parameters for the next query). This is wasteful of communication bandwidth because each query must be submitted to the DBMS and the results sent back to the application. Such groups of queries can be grouped together in the database as a stored procedure with one set of input parameters and one result set. In some database implementations stored procedures have the added advantage that the individual queries are precompiled giving an additional performance boost.

f) ***Avoid unnecessary sorting:*** Many SQL constructs force the database to perform one or more sorts before returning the final result set. Sorting is a very CPU and memory intensive task which is best avoided wherever possible.

If the ORDER BY clause must be used to sort the rows returned by the query, limit the number of columns to be ordered to reduce the amount of work the database must perform to fulfil the query.

If an ORDER BY must be used in a join, try to ensure that the columns specified in the ORDER BY clause are in the same table. This can help to ensure that the database does not perform a sort before returning the results.

Avoid using DISTINCT if duplicates are not a problem. Using the DISTINCT clause will normally cause a sort to be performed to enable duplicate rows to be removed from the result set. The DISTINCT clause is often used when unnecessary in cases where the query could never produce duplicate rows (for example when the result set of a Select on a single table includes the primary key of the table).

The UNION operator always uses a sort because it has to eliminate duplicate rows. However UNION ALL does not remove duplicates therefore no sort is performed. This is useful especially in cases where having duplicate rows is not a problem (assuming that the number of rows returned is not significantly greater) or it is known that duplicate rows cannot be returned.

g) ***Views:*** Views provide a window into one or more base tables. They are a very useful feature of SQL and there are often compelling reasons to use them, for example, to reduce the complexity of the queries which need to be written and hence enhance their maintainability. Views may also provide the only possible window into tables which store sensitive data in environments where views are used to protect data from unauthorised access.

Views are also frequently the source of SQL performance problems. This is because a view is defined in terms of an SQL query executed against the base tables. Therefore any SQL statement which makes use of a view must actually execute two queries - the first to extract data from the base tables and the second to extract data from the results of the first query. Therefore if you can possibly avoid using views by performing a query directly against the base tables do so - it will always perform better than executing the query against a view. While it may seem obvious do not forget that views may be optimised in the same way as normal queries by analysing the SQL which defines the view.

h) ***Calculations:*** Avoid performing calculations within the SQL itself if possible. This is a big resource waster but unfortunately is frequently unavoidable. The problem is that the calculation must be performed against each row in the table. If the calculation cannot be avoided then try to make sure that it performs the calculation on the minimum number of rows possible. This is especially true if the calculation is in the WHERE clause of the query. You should know that performing a calculation on an indexed column will in most cases disable the index for the current query.

Note that adding values to host variables and especially adding host variables together within the SQL can almost always be avoided:

```
SELECT u.app_user_name
FROM app_user AS u
WHERE u.last_login < (:todays_date - 30)
```

The following SQL will perform better than the above:

```
SELECT u.app_user_name
FROM app_user AS u
WHERE u.last_login < (:last_month)
```

i) ***Null values:*** Avoid the use of IS UNIL or IS NOT NULL against indexed columns since many databases are unable to perform index searches using these predicates.

j) ***Cursors:*** cursor is a work area that is used to store the results of a query. Declaration, processing and closing of cursor are some of overheads involved in using the cursor. In general a cursor should never be used for queries which are known in advance to return just one (or no) rows. The overhead for initialising, executing, processing and closing a cursor are quite high in comparison to executing a SELECT directly. The exception to this rule is when the row must be subsequently UPDATE as a result of the SELECT**.**

When a SELECT statement is used only for retrieval it runs faster i.e. if we use the FOR READ ONLY clause in the definition of the cursor. This will normally prevent exclusive row locking and hence improve data concurrency.

k) ***Comparisons:*** The comparisons in the WHERE clause of an SQL statement probably present the greatest opportunities for improving the performance of a query. They also present many pitfalls some of which are described here.

The first obvious tip is to try to reduce the number of comparisons wherever possible. Each comparison is applied to each row of the base or temporary tables. However be careful if multi-column indexes are used as removing an "unnecessary" comparison may actually decrease query performance.

Always place the comparisons which eliminate the most number of rows at the start of the WHERE clause. This means that subsequent comparisons will operate on fewer rows hence increasing the performance of the query.

Compare the data types which are equivalent in the WHERE clause. This prevents forcing the database to perform data type conversions and hence slowing down the query. In some databases indexes may not be used if data type conversion is required. The same principle applies to integers and floating point numbers - ensure that the precision and scale are the same on both sides of the comparison. A similar principle is to avoid the use of scalar functions in a query.

Try to avoid using OR to link conditions in the WHERE clause especially for comparisons on indexed columns. This is because using an OR will often force a full table scan. It is usually more efficient to split the query into two using a UNION to link the two parts.

Sometimes it is more efficient to do a full scan of the table rather than use an index. This is especially true when you are selecting most of the rows in the table. Most optimisers will select the correct access plan (i.e., full table scan or index search) if the table statistics are up to date.

13.6 COST OF OPERATIONS

The query optimisers use the rules to generate several alternative forms for a query, and then use the statistics to help determine the cost of each form. A cost-based optimization approach is possible only if the statistics are available. The overall cost of a query can be broken down into the cost of each operation in that query. So let's consider how to evaluate each operation and try to measure how much each is going to cost.

13.6.1 Cost of Selection

SELECT is a relational operator, denoted by s , that generates a new relation/ table by choosing rows from an existing relation/table that satisfy some condition. The new table contains all of the columns of the original table. Consider the following student table, S.

Student-ID	Name	Main-subject
A0301	Arvind Mathur	Medical information system
A0303	Nitish Kilash	IPR
A0305	Sidharth Kasturia	Business information system
A0307	Dinesh Kumar	IPR

The selection $s_{main\_subject}$ = IPR (S) will result another table, a horizontal subset of the original table; as follows

Student-ID	Name	Main-subject
A0303	Nitish Kilash	IPR
A0307	Dinesh Kumar	IPR

In general, a selection is done on a tuple-by-tuple basis, so the cost is the cost of reading the entire relation, that is

$$\text{cost}(s_q(R)) = \text{blocks}(R)$$

This is the minimum cost for the query. In determining this cost, we are ignoring the cost of writing the output; we will look at strategies for estimating the size of the output below.

For certain q's, and if R is indexed and clustered, then we can use the disk organization to lower the cost. For instance, assume that R is indexed on A. Then the following selection will have a lower cost.

$$\text{cost}(s_{A = \text{'a'}}(R)) = \min(\text{blocks}(R), \text{card}(R)/\text{values}(R,A))$$

The idea is that we can use the index to pinpoint in exactly which blocks each tuple with an 'a' value resides. In the worst case there is one in every block. Since there are at most

$$\text{card}(R)/\text{values}(R,A)$$

tuples with an 'a' value in all. In the worst case, we have to read that many blocks. If R is not only indexed, but also clustered on A, then

$$\text{cost}(s_{A = \text{'a'}}(R)) = \text{clusters}(R,A)$$

Since we can use the index to pinpoint the 'a' cluster, and we only need to read that cluster and not the entire relation.

13.6.2 Cost of a Projection

PROJECT is a relational operator denoted by π produces a new relation from an existing one by taking a vertical subset (i.e. columns) of the designated attributes from all of the tuples in the original table to form a new table. From the given student table, projection of name on student table, S π_{name}(S) gives

Name
Arvind Mathur
Nitish Kilash
Sidharth Kasturia
Dinesh Kumar

13.6.3 Cost of a Join

JOIN is a relational operator denoted by ◊◊ and produces a new relation by concatenating the rows of two existing tables. The original two tables must have at

least one attribute in common. A join condition may be specified that must be satisfied by the values of the common attributes in the original tables, such as equal, greater than, less than.

Consider the student table S given above and table A as follows

Student-ID	City
A0301	Kolkota
A0303	Chandigarh
A0305	Delhi
A0307	Pune

JOIN S and A, S◊◊A, will result as follows

Student-ID	Name	Main-subject	City
A0301	Arvind Mathur	Medical information system	Kolkota
A0303	Nitish Kilash	IPR	Chandigarh
A0305	Sidharth Kasturia	Business information system	Delhi
A0307	Dinesh Kumar	IPR	Pune

Equijoin is the most common relation JOIN operator. The condition for the join is the equality of values in common attribute. A natural join is an equijoin in which one of the duplicated attribute columns are eliminated.

Student-ID	Name	Main-subject
A0301	Arvind Mathur	Medical information system
A0309	Pawan Mishra	Technical writing
A0303	Nitish Kilash	IPR
A0305	Sidharth Kasturia	Business information system

Student-ID	City
A0301	Kolkota
A0303	Chandigarh
A0311	Trivandrum
A0305	Delhi
A0307	Pune

Equi-join S and A will be as follows:

Student-ID	Name	Main-subject	City
A0301	Arvind Mathur	Medical information system	Kolkota
A0303	Nitish Kilash	IPR	Chandigarh
A0305	Sidharth Kasturia	Business information system	Delhi

Joins, especially natural joins, are a common operation because normalised relations are often reconstructed during a query. Join is the most important operation for optimization because it is also one the most expensive queries.

Let's assume we have the following schema: *R(A,B), S(B,C)*. There are two bounds on the size of the result of a join.

if $\pi_B(R) \cap \pi_B(S) = \phi$ then $R \lozenge\lozenge S = \phi$.

if $\pi_B(R) = \pi_B(S)$ then $R \lozenge\lozenge S = R X S$.

Otherwise the size of the result is somewhere in between.

13.7 RULES FOR STRUCTURED QUERY FORMULATION

The technique known as algebraic manipulation can be used for query optimization. The technique converts a query into a relational algebraic expression using the relational operators, project (π_X ( )), selection (σ_F ( )), cartesian product (×). That algebraic expression is transformed into an optimal one using the algebraic laws of expression equivalence.

13.7.1 The Algebraic Laws of Equivalence

a) Commutative laws for join and cartesian product

- $E_1 \times E_2 = E_2 \times E_1$
- $E_1 \lozenge\lozenge E_2 = E_2 \lozenge\lozenge E_1$

b) Associative laws for join and cartesian product

- $(E_1 \times E_2) \times E_3 = E_1 \times (E_2 \times E_3)$
- $(E_1 \lozenge\lozenge E_2) \lozenge\lozenge E_3 = E_1 \lozenge\lozenge (E_2 \lozenge\lozenge E_3)$

c) Cascade of projections

$\pi_{A1,\cdots An}(\pi_{B1,\cdots,Bk}(E( )) = \pi_{A1,\ldots,An}(E( ))$= if *A1,..., An* , is a subset of *B1,... Bk* ,

d) Cascade of selections

$\sigma_{F1 \text{ and } F2}(E) = \sigma_{F1}(\sigma_{F2}(E)$ 0= = $\sigma_{F2}(\sigma_{F1}(E))$=

e) Commuting selections and projections

- $\pi_{A1,\ldots,An}(\sigma_F(E)) = \sigma_F(\pi_{A1,\ldots,An}(E))$ if *F involves* only attributes from *A1,...,An*.

- $\pi_{A1,\ldots,An}(\sigma_F(E)) = \pi_{A1,\ldots,An}(\sigma_F(\pi_{A1,\ldots,An,\, B1,\ldots,Bk}(E)))$ = if *F* also involves the attributes *B1, ..., Bk* ,that are not among the *A1, ..., An*.

6) Commuting selections with cartesian product

- $\sigma_F(E_1 \times E_2) = \sigma_F(E_1) \times (E_2)$ if *F* involves only attributes of E_1

• $\sigma_F(E_1 \times E_2) = \sigma_{F1}(E_1) \times \sigma_{F2}(E_2)$ where F_1 involves only attributes of E_1, and F_2 involves only attributes of E_2, and F does not involve any conditions with attributes from both operands.

• $\sigma_F(E_1 \times E_2) = \sigma_{F3}(\sigma_{F1}(E_1) \times \sigma_{F2}(E_2))$ where F_1 involves only attributes of E_1, and F_2 involves only attributes of E_2, and F_3 involves any conditions with attributes from both operands.

7) Commuting projections with cartesian product

$\pi_{A1, ..., An}(E1 \times E2) = \pi_{B1, ..., Bk}(E1) \times \pi_{C1, ..., Cm}(E2)$, where $\{A_1, ..., A_n\} = \{B_1, ..., B_k\} \cup \{C_1, ..., C_m\}$ and $B_1, ..., B_k$ are attributes that appear only in E_1 where as $C_1, ..., C_m$ are attributes that appear only in E_2

13.7.2 Pushing Selections

The rule states that we can *push* a selection on the result of a join before the join, that is, do the selection, then the join.

This means it is advised to do the selections first since it is cheaper to work with smaller relations especially in joins. By doing the selection first, the size of the relations involved in the join will usually be less.

Assume θ_R is a condition that only involves attributes in relation *R*.

$\sigma\theta_R(R \lozenge\lozenge S) = (\sigma\theta_R(R)) \lozenge\lozenge S$

It means that first to select only the desired tuples from relation R, which will result in a subset of R, then perform the join with S, so that it requires only lesser number of reads for the subset join operation than the original.

A natural join costs fewer disk reads if smaller relations are used. By doing the selection first, the relation become smaller than the original relation. Hence the cost also gets reduced.

13.7.3 Pushing Conjunctive Selections

The rule states that we can *push* a conjunctive selection condition into a join by splitting that condition.

Assume θ_R is a condition that only involves attributes in relation *R* and θ_S is a condition that only involves attributes in relation S.

$\sigma\theta_R \text{ AND } \theta_S(R \lozenge\lozenge S) = (\sigma\theta_R(R)) \lozenge\lozenge (\sigma\theta_S(S))$

Here $sq_R(R)$ is a subset of R and $sq_S(S)$ is a subset of S, which are smaller than R & S resp. Hence the new subset join requires lesser number of reads than the original RS.

13.7.4 Reordering and Generalising Selections

The combined condition is cheapest since each select requires a total scan through the relation.

$\sigma\theta_1(\sigma\theta_2(R)) = \sigma\theta_2(\sigma\theta_1(R)) = \sigma\theta_1 \text{ AND } \theta_2(R)$

Assume that the relation R has 1000 tuples, $sq_1(R) = 450$ and $sq_2(R) = 750$.

To execute $sq_2(R)$, the whole relation R is to be read, this may result in a subset of lesser i.e. 750. Then to execute $sq_1(sq_2(R))$, 750 tuples need to be read. This may result in 300, i.e., the total read may be 1000+700=1700.

But by changing the order of selection $sq_2(sq_1(R))$, the first selection will result in 400 tuples, which is to read for the second selection. That is, the total read may be 1000+400=1400.

But by combining the selection as sq_1 AND $q_2(R)$, you need to read the relation only once, i.e., 1000 tuples.

Selections applied in succession can be reordered, and turned into a conjunctive condition, so the cost of more reads can be reduced to lesser reads.

13.7.5 Pushing Projections

Creating smaller temporary relations can save disk space, and consequently disk I/O. Assume *A* is an attribute of *R* and *B* is the lone join attribute of *R* and *S*, then

$\pi_A(R \lozenge\lozenge S) = \pi_A(\pi_{A,B}(R) \lozenge\lozenge S)$

13.7.6 Reordering Joins

Joins are to be reordered to join the smallest relation first, so that the smaller relation joins the larger relation.

$(R \lozenge\lozenge S) \lozenge\lozenge T = R \lozenge\lozenge (S \lozenge\lozenge T)$

Suppose R and S each have 1,000,000 tuples, but T only has 10. Then doing the S join[] T first may result in a small relation which we can then join to R.

13.8 EVALUATION STRATEGIES

13.8.1 Selection

Assume relation *R* is unclustered, unsorted, and not indexed. Then, to evaluate a selection or *R* we have to iterate over every tuple in *R*. The cost of doing the query is the cost of reading *R*, that is, it is blocks(*R*).

If *R* is clustered, we may be able to save some in-memory costs, but we still have to read the entire relation.

The big savings come from using an index. If R is indexed then we can use the index to quickly find a value. So for a selection like s (A = 'Sunil Satpal') we can use the index on A to quickly find 'Sunil Satpal' in O(1) block reads. Similarly if *R* is sorted on *A* then O(log2 n) block reads are necessary to process the selection using a binary search technique on the sorted order.

13.8.2 Join

In determining the best join strategy we have to take account of several factors.

- physical order of tuples
- indexing/clustering
- cost of computing a *temporary* index

For the following join strategies we will consider the join $R \Diamond\Diamond S$ where

card(R) = 10000, blocks(R) = 500

card(S) = 200, blocks(S) = 10

and that there is a single join attribute, A.

The n the cost of $S \Diamond\Diamond R$ is

blocks(R) +card(R) × blocks(S) = 500 + 10000 × 10 = 100,500

13.8.2.1 Nested loop join/block iteration

Executing an operation several times is known as looping. The number of times a particular statement or operation is to be executed depends on another statement or operation. Both theses statements need to be represented in nested loop. This can be expanded to any number of such statements. In this type of nesting, the inner loop statement executes for each iteration of its outer loop statement and so on.

For nested loop join, the first strategy is we make no assumption about how the relations are stored. This strategy just iterates over all the tuples in each relation. Algorithm is the following.

for each tuple r in R do

for each tuple s in S do

if r.A = s.A then add to result

Here we read each block of R, then for each tuple in that block, we read the whole relation S. So the cost is

Blocks (R) + card(R) × blocks(S)

For the query example the cost is

10 + 200 × 500 = 100,010

13.8.2.2 Merge join

Merge join is a strategy which says that If both R and S are sorted on attribute A then we can merge the relations by going through each in sorted order and joining when necessary.

The cost of this join is blocks(R) + blocks(S). = 500+10 =510

13.8.2.3 Index join

If one of the relations is indexed, in the case of iteration the second read or the inner loop of the iteration is done only on the index. Index join takes advantage of the fact that one of the relations is indexed on the join attribute

The cost of this strategy depends on how widely the tuples in S are distributed. If the tuples in S are uniformly distributed, then the cost of the join is

blocks(R) + card(R) × blocks(S) × (values{S, A)/ card(S))

or in other words, we have to read R, and for each tuple in R we have to read enough blocks in S to get all the tuples containing the joined value.

If S is clustered as well as indexed however, the cost is lowered and it approaches the best-case cost since all the joining tuples appear in the same or adjacent blocks.

13.8.2.4 Hash join

Hash join does not assume that either relation is clustered, indexed or sorted. The idea behind hash join is to create a hash table where the key is the join attribute and the values are the tuples in R. Then S is read and for each tuple in S, the hash table is probed, and any found tuples are joined.

In the best case, hash join approaches the cost of just reading R and S, that is, blocks(R) + blocks(S), if we factor out the cost of building and probing the hash table.

Self Check Exercise

a) Consider the relations Books (title, author, pname, accno); Publisher (pname, paddr, pcity); borrowers (name, addr, city, cardno); loans (cardno, accno, date). Write a query to find all the book titles for books borrowed before the 4th July 1997. Optimise the query by applying the rules for optimization.

Note: i) Write your answer in the space given below.

ii) Check your answer with the answers given at the end of this unit.

.....

.....

.....

.....

.....

13.9 SUMMARY

This unit covered the cost implication of the query operations such as selection, join and projection and mentioned different optimization strategies which can be followed to write faster and less costly queries. Even though most of the DBMS compilers provide query optimizers, it will be helpful and worth learning these optimization parameters and solutions to moderate the queries.

13.10 ANSWERS TO SELF CHECK EXERCISES

SELECT title FROM books

(SELECT title, author, pname, accno, name, addr, city, cardno, date FROM loans, borrowers, books WHERE borrowers.cardno = loans.cardno AND books.accno = loans.accno)

WHERE date<04/07/1997

The algebraic expression for the above query is the following

p_{title} (s_{date}<4/7/1997 (p_S (s_F (loans × borrowers × books))

where S denotes the following attribute list *'title, author, pname, lcno, name, addr, city, cardno, date'*, and F denotes the condition *'borrowers.cardno = loans.cardno AND books.accno=loans.cardno*

step1. Simplify the complex expression

π_{title} (σ_{date}<4/7/1997 (π_S (σ*borrowers.cardno = loans.cardno AND books.accno=loans.cardno (loans × borrowers × books))*

as

π_{title} (σ_{date}<4/7/1997 (π_S (σ *borrowers.cardno = loans.cardno (loans × borrowers × books) AND s (books.accno=loans.cardno (loans × borrowers × books))*

step2. Push the selection down as far as possible. Apply rule 4,5,& 6 froms laws of algebraic equivalence.

π_{title} (σ *(books.accno=loans.cardno(books)* (σ *borrowers.cardno = loans.cardno (borrowers)* (σ_{date}<4/7/1997(loans)))

step 3. Push the projection down as fas as possible. Aplly rule 3,5 and 7

π_{title} (σ *(books.accno = loans.accno* ($\pi_{loans.accno}$ (σ *borrowers.cardno = loans.cardno* (($\pi_{titile,\ books.accno}$ *(books)* (($\pi_{borrowers.cardno}$ *(borrowers)* ($\pi_{loans.accno,\ loans.cardno,\ date}$ (σ_{date}< 4/7/1997(loans))))

13.11 KEYWORDS

Hashing : Hash - addressing, a storage and access method in which data records are stored at an address that can be computed according to some mathematical hashing function, generally using the value of the primary key. Hash-addressing requires less storage than other indexing methods but some additional computation is needed.

Index : Index is an auxiliary data structure used to speed up access to a data set on the basis of the key value for each record. The index may actually contain the key values, and the pointers and the key may be used to generate the address of the pointer in the index. The index can be used to provide an order to the data records and to provide the direct access to the records in the data set.

Relational Algebra : The definition of operators that can be used on relations. The algebra combines the set operators union, intersection, difference and Cartesian product, plus the special relational operators SELECT, PROJECT and JOIN. The result of every operation on a relation is also a relation.

13.12 REFERENCES AND FURTHER READING

CAROLYN WATTERS(1992). Dictionary of Information Science and Technology: SanDiego: Academic Press.

Mcfadyen, Ron(1991). Introduction to Structured Query Language. Dubuque :W M C Brown publishers.

Parag Diwan,R.K Suri and Sanjay Kaushik(2001). IT Encyclopedia.com, fundamentals of Information Technology. New Delhi: Pentagon Press.

Sanin, Leo.(1998).Client /server programming with access and SQL Server. New Delhi: Galgotia Publications.

Walker, Stephen (1990). Improving subject retrieval in online catalogue: relevance feedback and query expansion. London: British Library.