Preparation cours 2019-06-19

# Step 4.2 (chapitre 18)

Objective: To determine an efficient file organization for each base relation

One of the main objectives of physical database design is to store and acess data in an efficient way. For example, if we want to retrieve staff tuple in alphabetical order of name, sorting the file by staff name is a good file organisation. However, if we want to retrieve all staff whose salary is in certain range, searching a file ordered by staff name would not be particularly efficient. To complicate matters, some file organisation are efficient for bulk loading data into the database but inefficient after that. In others, we may want to use an efficient storage structure to set up the database and then change it for normal opertional use.

The objective of this step therefore is to choose an optimal file organisation for each relation, if the target DBMS allows this, In many case, a reltional DBMS may give little or no chocie for choosing file organizations, although some may be established as indexes more fully, we provide guidelines in Appendix F.7 for selecting a file organization based on the following types of file:

Heap

Hash

Indexed Sequential Access Method (ISAM)

B­+ - tree

Clusters

If the target DBMS does not allow the choice of file organisations, this step can be omitted.

Document choice of file organizations

The choice of file organizations should be fully documented, along with the reasons for the choice. In particular, document the reasons for slecting one apparoach where many alternatives exist.

Step 4.3 Choose indexs

Objective: To determine whether adding indexes will improve the performance of the system

One approach to selecting an appropriate file organiszation for a relation is to keep the tuples unordered and create as many secondary indexes as necessary.

Another approch is to order the tuples in the relation by specifying a primary or clustering index (see Appendix F.5). In this case, choose the attribute for ordering or clustering the tuples as:

The attribute that is used most often for join operations, as this makes the join operation more efficient, or

The attribute that is used most often to acess the tuples in a relation in order of that attribute.

If the ordering attribute chosen is a key of the relation, the index will be a primary index;

If the ordering attribute is not a key, the index will be a clustering index. Remember that each relation can have only either a primary index or a clustering index.

Specyfing indexes

We saw in Section 7.3.5 that an index can usually be created in SQL using the CREATE INDEX statement. For example, to create a primary index on the PropertyForRent relation based on the propertyNO attribute, we might use the following SQL statement:

CREATE UNIQUE INDEX PropertyNoInd ON PropertyForRent(propertyNo);

To create a clustering index on the PropertyForRent relation based on the staffNo attribute, we might use the following SQL statement:

CREATE INDEX StaffNoInd ON PropertyForRent(staffNo) CLUSTER;

As we have already mentionned, in some systems the file organization is fixed. For example, until recently Oracle has supported only B+-trees, but has now added support for clusters. On the other hand, INGRES offers a wide set of different index structures that can be chosen using the following optional clause in the CREATE INDEX statement:

[STRUCTURE = BTREE | ISAM | HASH | HEAP]

Choosing secondary indexes

Secondary indexes provide a mechanism for specifying an additional key for a base relation that can be used to retrieve data more eifficently. For example, the PropertyForRent relation may be hashed on the property number, propertyNo, the primary index. Howerver, there may be frequent acess to this relation based on the rent attribute. In this case we may decide to add rent as a secondary index.

There is an overhead involved in the maintenance and use secondary indexes that has to be balanced against the performance improvement gained when retrieving data. This overhead includes:

Adding an index record to every secondary index whenever a tuple is inserted into the relation

Updating a secondary index when the corresponding tuple in the relation is updated;

The increase in disk space needed to store the secondary index;

Possible performance degradation during query optimization, as the query optimizer may consider all secondary indexes before selecting an optimal execution strategy.

Guideline for choosing a “wish-list” of indexes

One approach to determining which seconday indexes are needed is to procudre a “wish-list” of attributes that we consider to be candidates for indexing, and then to examine the impact of maintaining each of these indexes. We provide the following guidelines to help procudre such a wish-list:

1. Do not index small relation. It may be more efficient to search the relation in memoery than to store an additional index structure
2. In general, index the primary key of a relation if it is not a key of the file organization. Although the SQL standard provides a clause for the specifation of primary keys, as discussed in Section 7.2.3, it should be noted that this does not guarantee that the primary key will be indexed.
3. Add a secondary index to a foreign key, if it is frequently accessed. For example, we may frequently join the PropertyForRent relation and the PrivateOwner Business Owner relations on the attribute ownerNo, the owner number. Therefore, it may be more efficent to add a secondary index to the PropertyForRent relation based on the attribute ownerNo. Note that some DBMS may automatically index foreign keys.
4. Add a secondary index to any attribute that is heavily used as a secondary key (for example, add secondary index to the PropertyForRent relation based on the attribute rent, as discessed previously).
5. Add a secondary index on attributes that are frequently involved in:
   1. Selection or join criteria;
   2. ORDER BY;
   3. GROUP BY
   4. Other operations involgin sorting (such as UNION or DISTINCT).
6. Add a secondary index on attributes involved in built-in aggregate functions, along with any attributes used for the built-in functions. For example: to find the average staff salary at each branch, we could use the following SQL query:

SELECT branchNo, AVG(salary)

FROM staff GROUP BY branchNo;

From the previous guideline, we could consider adding an index to the branchNo attribute by virtue of the GROUP BY clause. However, it may be more efficient to consider an index on both the branchNo attribute and the slary attribute. This may allow the DBMS to perform the entire query from data in the index alone, without having to acess the data file. This is sometime called an index-only plan, as the required response can be produced using only data in the index.

1. As more general case of the previous guideline, add a secondary index on attributes that could result in an index-only plan.
2. Avoid indexing an attribute or relation thati s frequently updated.
3. Avoid indexing an attribute if the query will retrive a significant propoertion (fro example 25%) of the tuples in the relation. In this case, it may be more efficient to search the entire relation than to search using an index.
4. Avoid indexing attributes that consist of long character strings.

If the search crtiera involve more than one predicate, and one of the terms contains an OR clause, and the term has no index/sort order, then adding indexes for the other attributes is not going to help improve the speed of the query, because a linear search of the relation will still be required. Fro example, assume that only the type and rent attributes of the PropertyForRent relation are indexed, and we need to use the following query:

SELECT \*

FROM PropertyForRent

WHERE (type = ‘Flat’ OR rent > 500 OR rooms > 5);

Although the two indexes could be used to find the tuples where (type = ‘Flat’ or rent > 500), the fact that the rooms attribute is not indexed will mean the these indexes cannot be used for the full WHERE clause. Thus, unless therer are other queries that would benefit from having the type and rent attributes indexed, there would be no benefit gained form indexing them for this query.

On the other hand, if the predicates in the WEHRE, clause we AND’ed together, the two indexes on the type and rent attributes could be used to optimize the query.

Removing indexes from the wish-list

Having drawn up the wish-list of potential indexes, we should now consider the impact of each of these on update transactions. If the maintenance of the index is likely to slow down important update transactions. Then consider dropping the index from the list. Note, however, that a particular index may also make update operations more efficeient. For example, if we want to update a member of staff’s salary, given the member’s staff number, staffNo, and we have an index on staffNo, then the tuple to be updated can be found more quickly.

It is good idea to experiment when possible to determine whether an index is improving performance, providing very little improvements, or adversely impacting performance. In the last case, clearly we should remove this index from the wish-list. If there is little observed improvement with the addition of the index, further examination may be necessary to determine under what circumstances the index will be useful, and whether these circumstances are sufficiently important to warrant the implementation of the index.

Some systems allow users to inspect the optimizer’s strategy for executing a particular query or update, sometimes called the Query Execution Plan (QEP). Fro example, Microsoft Office Access has performance Analyzer, Oracle has an EXPLAIN PLAN diagnostic utility (see Section 23.6.3), DB2 has an EXPLAIN utility, and INGRES has an online QEP-viewing facility. When a query runs slower than expected, it is worth using such a facility to determine the reason for the slowness and to find an alternative strategy that may improve the performance of the query.

If a large number of tuples are being inserted into a relation with one or more indexes, it may be more efficient to drop the indexes first, perform the inserts, and then recreate the indexes afterwards. As a rule of thumb, if the insert will increase the size of the relation by at least 10%, drop the indexes temporarily.

Updating the database statistics

The qery optimizer relies on database statitics held in the system catalog to select the optimal strategy. Whenerver we create an index, the DBMS automatically adds the presence of the index to the system catalog. However, we may find that the DBMS requires a utlity to be run to update the statistics in the system catalog relating to the relation and the index.

Document choice of indexes

The choice of indexes shoulb be fully documented, along with the reasons for the choice. In particular. If there are performance reasons why some attributes should not be indexed, these should also be documented.

File organizations and indexes for DreamHome with Microsoft Office Acess

Like most, if not all. PC DBMS, Microsoft Office Access uses a fixed file organisation, so if the target DBMS, is Micorsoft Office Access, Step 4.2 can be omitted. Microsoft Office Acess Does, however, support indexes as we will now briefly discuss. In this section we use the terminology of Office Access, which refers to a relation as a table with fiels and records.

Guidelines for indexes In Office Access, the primary key of a table is automatically indexed, but a filed whose data type is Memo, Hyperlink, or OLE Object cannot be indexed. For other fileds, Microsoft advises indexing a filed if all the following apply:

The field’s data type is Text, Number, Currency, or Date/Time;

The user anticipates searching for values stored in the field;

The user anticipates sorting values in the field;

The user anticipates storing many different values in the field. If many of the values in the field are the same, the index may not significantly speed up queries

In addition, Microsoft advises:

Indexing field on both sides of a join or creating a relationship between these field, in which case Office Access will automatically create an index on the foreign key field, if one does not exist already;

When grouping records by the balues in a joined field, specifying GROUP BY for the field that is in the same table as the field the aggregate is being calculated on.

Microsoft Office Access can optimize simple and complex predicates (which are called expressions in Office Access). For certain types of complex expressions, greater level of optimization. A complex expression is formed by combining two simple epxression with the And or OR operator, such as:

branchNo = ‘BOO1’ AND rooms > 5

type = ‘Flat’ OR rent > 300

In Office Acess, a complex expression is fully or partially optimizable depending on whether one or both simple expressions are optimizable, and which operator was used to combine them. A complex expression is Rushmore-optimizable if all three of the following conditions are true:

The expression uses AND or OR to join two conditions

Both conditions are made up fo simple optimizable expressions;

Both expression contain indexed fields. The fileds can be indexed individually or they can be part of a multiple-filed index.

Indexes for DreamHome

Before creating the wish-list, we ignore small tables from further consideration ,as small tables can usally be processed in memoery without requiring additional indexes. For DreamHome we ignore the Branch, Telephone, Manager, and Newspaper tables from fruther consideration. Based on the guidelines provided earlier:

1. Create the primary key for each table, which will cause Office Acess to auto-matically index this field
2. Ensure all relationships are created in the Relationships window, which will cause Office Acess to automacally index the foreign key fields.

As an illustration of which other indexes to crfeate, we consider the query transctions listed in Appendix A for the StaffClient user views of DreamHome. We can producre a summary of interactions between the base tables and these transactions shown in Table 18.2 This figure shows for each table: the transaction(s) that operate on the table, the type of acess ( a search based on a predicate, a join together with the join field, any ordering field, and any grouping field, and the frequency with which the transcation runs.

|  |  |  |  |
| --- | --- | --- | --- |
| TABLE | TRANSACTION | FIELD | FREQUENCY PER DAY |
| Staff | (a), (d) | Predicate fName, lName | 20 |
|  | (a) | Join: Staff on supervisorStaffNo | 20 |
|  | (b) | Ordering: fNAMe, Iname | 20 |
|  | (b) | Predicate: position | 20 |
| Client | (e) | Join: Staff on staff No | 1000-000 |
|  | (j) | Predicate: fName, Iname | 1000 |
| PropertyForRent | (c) | Predicate: rentFinish | 5000-10,000 |
|  | (k), (l) | Predicate: rentFinish | 100 |
|  | c) | Join: PrivateOwner/BusinessOwner on ownerNo | 5000-10,000 |
|  | d) | Join: Staff on staff No | 20 |
|  | f) | Predicate: city | 50 |
|  | f) | Predicate: rent | 50 |
|  | g) | Join: Client on clientNo | 100 |
| Viewing | i) | Join: Client on clientNo | 100 |
| Lease | c) | Join: PropertyForRent on propertyNo | 6000-10,000 |
|  | l) | Join: PropertyForRent on propertyNo | 100 |
|  | j) | Join: Client on clientNo | 1000 |

Based on this information, we choose to create the addional indexes shown in Table 18.3 We leave it as an exercice for the reader to choose additional indexes to create in Microsoft Office Access for the transactions listed in Appendix A for the Branch biew of DreaHome (see Exercice 18.5)

File organization and indexes for DreamHome with Oracle

In this section we repeat the previous exercise of determining appropritate file organizations and indexes for the StaffClient user views of DreamHome. Once again.

Table 18.3 Additional indexes to bre created in Microsof Office Acess based on the qeury transcations for the StaffClient user views of DreamHome

|  |  |
| --- | --- |
| Table | Index |
| Staff | fName, Iname |
|  | Position |
| Client | fNAME, INAME |
| PropertyForREnt | rentFinish |
|  | City |
|  | rent |

We use the terminology of the DBMS-Oracle refers to a relation as a table with columns and rows.

Oracle automatically adds an index for each primary key. In addition, Oracle recommends that UNIQUE indexes not be explicity defined on tables but instead UNIQUE integrity constraints be defined on the desired columns. Oracle enforces unique key. Exceptions to this recommendation are usally performance-related. For example, using a cREATE TABLE … AS SELECT with a UNIQUE constraint is slower than creating the table wihtout the constrant and then manually creating a UNIQUE index.

Assume that the tables are created with identified primary, alternate, and foreign keys specified. We now dientify whether any clusters are required and whether any additionnal indexes are required. To keep the design simple, we will assume that clusters are not appropriate. Again, considering just the query transactions listed in Appendix A for the StaffClient user views of DreamHome, there may be performance benefits in adding the indexes shown in Table 18.4 Again, we leave it as an exercice for the reader to choose additional indexes to create in Oracle for the transctions listed in Appendix A for the Branch view of DreamHome (see Exercice 18.6)

# Chapitre 7.3.5 et 7.3.6 Creating an Index (CREATE INDEX)

An index is astructure that provides accelereated access to the rows of a table based on the values of one or more columsn (see Appendix F for discussion of indexes and how they may be used to improve the efficenty of data retrivels). The presence of an index can significanlty improve the performance of a query. However, as indexes may be updated by the system every time the underlying tables are updated, additional overhead may be incurred. Indexes are usally created to satisfy particular search criteria after the table has been in use for some time and has grown in size. The creation of indexes is not stand SQL., However, most dialects support at least the following capabilities:

CREATE [UNIQUE] INDEX indexName

ON TableName (columnName [ASC | DESC] [, . . . ])

The specified columns constitute the index key and should be listed in major to minor order. Indexes can be created only on base tables not on views. If the UNIQUE clause is used, uniqueness of the indexed column or combination of columns will be enforced by the DBMS. This is certainly required for the primary key and possibly for other columns as well ( for example, for alternate keys). Although unique index on a table with records in it, because the values stored for the indexed column(s) may already contain duplicates. Therefore, it is good pratice to create unique indexes, at least for primary key columns, when the base table is created and the DBMS does not automatically enforce primary key uniqueness

For the Staff and PropertyForRent tables, we may want to create at least the following indexes:

CREATE UNIQUE INDEX StaffNoInd ON Staff (staffNo);

CREATE UNIQUE INDEX PropertyNOind ON PropertyForREnt (propertyNo);<

For each columns, we may specifty that the order is ascending (ASC) or descending (DESC), with ASC being the default setting. For example, if we create an index on the PropertyForRent table as:

CREATE INDEX RentInd On PpropertyForREnt (city, rent);

Then an index called RentInd is created ofr the PropertyForRent table, Entries will be in alphabetical order by city and then by rent within each city.

# 7.3.6 Removing an Index (DROP INDEX)

If we create an index for a base table and later decide that is is no longer needed we can use the DROP INDEX statement to remove the index from the database dROP INDEX has the following format:

DROP INDEX IndexName

The following statement will remove the index created in the previous example:

DROP INDEX RentINd;

# Chapitre 19: Methodology – Monitoring and Tuning the Operational System

Chapter Objectives

In this chapter you will learn:

The meaning of denormalization

When to denormalize to improve performance

The importance of monitoring and tuing the operational system.

How to measure efficiency

How system ressources affect performance

In the previous chapter we presented the first five steps of the physical database design methodology for relational databases. In this cahpter we describe and illustrate by example the final two steps of the physical database design methodology.

We provide guidelines for dtermining when to denormalize the logical data model and introduce redundancy, and then discess the importance of monitoring the opreational system and constinuing to tune it. In place, we show physical implementation details to clarify the discussion.

19.1 Denormalizing and introducing controlled redundancy

Step 7: Consider the introduction of controlled redundancy

Objective: To determine whether introducing redundancy in a contrlled manner by realxing the normalization rules will improve the performance of the system

As we disceussed in Cahpter 14 and 15, normalization is a technique for deciding which attributes belong together in a relation. One of the basic aim of relational database design is to group attributes together in a relation because there is a functional dependency between them. The result of normalization is a logical database design that is structurally consistent and has minimal redundancy. However, it is sometimes argued that a normalized database design does not provide maximum processing efficiency. Consequently, there may be circumstances in which it may be necessary to accept the loss of some of the benefits of a fually normalised design in favor of performance. This should be considered only when it is estimated that the system will not be able to meet its performance requirements. We are not ad vocating that normalization should be omitted from logical database design: normalization force us to understand completely each attribute that has to be represented in the database. This mary be the most important factor that contribute to the overall success of the system. In addition, the following factors have to be considered:

Denormalization makes implementation more complex

Denormalization often sacrifics flexibility;

Denromalization may speed up retrivals but slowes down updates

Formally, the term denormalization refers to a refinement ot the relational schema such that the degree of normalization for a modified relation is less than the degree of at least one of the orignal relations. We also use ther term more lossely to refer to sitations in which we combine two relations into on new realtion, and the new relations is till normalized bu contains more nulls than the original relations. Some authors refer to dernormalization as usage refinement

As a gneral rule of thumb, if performance is unsatisfcatory and a relation has a low update rate and very high query rate, denormalization may be a viable option.

The transaction/relation cross-reference matrix that may have been produced in Step 4.1 provides useful information for this step. The matrix summarizes in a visual way the acess patterns of the transactions that will run on the database. It can be used to highlight possible candidates for denormalisation and to assess the effects this would have on the rest of the model.

More specifically, in this step we consider duplicating certain attributes or joining relation together to reduce the number of joins required to perform a query.

Indirectly, we have encountered an implciit example of denormalization when dealing with address attributes. For example, consider the definition of the Branch realtion:

Branch (branchNo, street, city postcode, mgrStaffNo)

Strictly speaking, this relation is not in third normal form: postcode (the post or zip code) functionnaly determines city. In other words, we can determine the value of the city attribute given a value for the poscode attribute. Hence, the Branch relation is in 2NF. To normalize the relation to /NF, it would be ncessary to split the relation into two, as follows:

Branch (branchMNo, street, postcode, mgrStaffNo)

Postcode (postcode, city)

However, we would rarely wish to acess the branch address without the city attribute. This would meanr that we would have to perform a join whener we want a complete address ofr a branch. As a result, we sttle for 2NF and implement the original Branch relation.

Unfortunately, there are no fiexed rules for determining thwne to denormalize relations. In this step we discess some of the more common situations for considering denormalization. For additional information, the interested reader is referred to Rogers (1989) and Fleming and Von Halle (1989) In particular, we consider denormalization in the following sutations speifically to speed up frequent or critical transactions:

Step 7.1 Comgining one-to-one (1:1) relationships

Step 7.2 Duplicating non-key attributes in one-to-many (1:\*) relationship to reduce joins

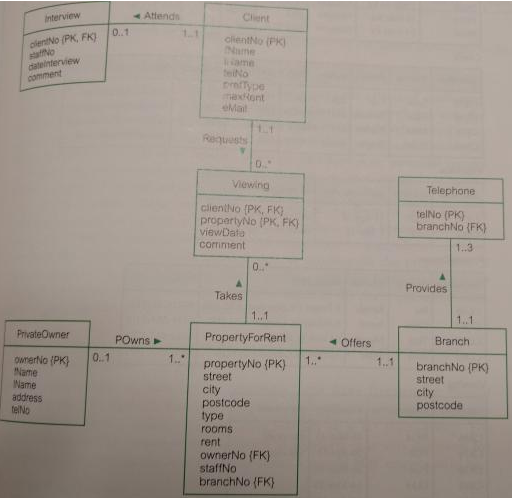
Step 7.3 Duplicating foreign key attributes ti none-to-many (1:\*) relationships to reduce joins

Step 7.4 Duplicating attributes in many-to-many (\*:\*) relationships to reduce joins

Step 7.6 Creating extract tables

Step 7.7 Partiioning relations

To illustrate these step we use the relation diagram shows in Figure 19.1 (a) and the sample data shown in Figure 19.1 (b)



Step 7.1: Combining one-to-one (1:1) relationships

Re-examine one-to-one (1:1) relationships to dtermine the effects of combining the relations into a single realtion. Combination should be considered only for relations that are frequently referenced together and infrequently referenced separetly. Consider. For example the 1:1 relationship between Client and Interview, as shown in Figure 19.1 The Client relation contains information on portential renters of property: The interview relations contains the date of the interview and comments made by a member of staff about a Client

We could combine these two relations together to form a new relation ClientInterview as shown in Figure 19.2 Because the relationship between Client nad interview is 1:1 and the participation is optional, there may be a significant number of nulls in the combined relation ClientInterview, depending on the propertion of tuple involed in the particiaption, as shouwn in Figure 19.2 (b) If the original Client relation is large and the proportion of tuple involved in the participation is small there will be a significant amount of wates space.

Step 7.2 Duplicating non0key attributes in one-to-many (l:\*) relationships to reduce joins

With the specific aim of reducing or removing joins form frequent or critical queries, consider the benefits that may result in duplicating one or more non-key attributes of the parent relation in the child relation in a 1:\* relationship. For example, whenever the PropertyForRent relations is acessed, it is very common for the owner’s name to be accessed at the same time. A typical SQL query would be:

SELECT p.\*, o.lName

FROM PropertyForRent p, PrivateOwner o

WHERE p.ownerNo = o.ownerNO AND branchNo = ‘B003’;

Based on the original relations diagram and smaple relations shoun in Figure 19.1 if we duplicate the Iname attribute in the PropertyForRent relations, we can remove the PrivateOwner relation from the query, which in SQL becomes:

SELECT p.\*

FROM PropertyForRent p

WHERE branchNo = ‘B003’;

Based on the revised relation shoun in Figure 19.3

The benefits that result from this change have to be balanced against the problems that ma yarise. For example, if the duplicated data is changed in the parent relation, it must be updated in the child relation. Further, for a 1:\* relationship there may be mutiple occurences of each data item in the child relation (for example, the name Farrel and Shaw both appear twice in the revised PropertyForRent relation), in which case it becomes ncessary to maintain consistency of multiple copies. If the update of the Iname attribute in the PrivateOwner and PropertyForREnt relation cannot be automated, the potential for less of integiry is considerable. And associated problem with duplication is the additional time that is required to maintain consistency automatically every time a tuple is inserted, updated, or deleted. In our case ,it is ulikely that the name of the owner of a property will change, so the duplication may be warranted.

Another problem to consider is the increase in storage space resulting from the duplication. Again, with the relatively low cost of secondary storage nowadys, this may not be so much of a problem. Howerver, this is not justiciation for arbitrary duplication.

A special case of a one-to-many (1:\*) relationship is a lookup table, sometimes called a reference table or pick list. Typically, a lookup table contains a code and a description. Fro example, we may define a lookup (parent) table for property type and modify the PropertyForRent (chiild) table, as shoun in Figure 19.4 the advantages of using a lookup table are:

* Reduction in the size of the child relation: the type code occupies 1 byte as opposed ot 5 bytes for the type description.
* If the description can change ( which is not the case in this particular example\_. It is easier changing it once it’s in the lookup table as opposed to changing it many times in the child relations;
* The lookup table can be used to validate user input.

If the lookup table is used in frequent or critical querie, and the description is unlikely ot change, consideration should be given to duplicating the description attribute in the child relation, as shound in Figure 19.5 The original lookup table is ot redundant- it can still be used to validate user input. However, by duplicating the description in the cild relation, we have elimnated the need to join the child relation to the lookup table.

Step 7.3: Duplicating foreign key attributes in one-to-many (1:\*) relationships to reduces joins

Again, with the specific aim of reducing or removing njoins from frequent or cirtical queries, consider the benefits that may result in duplicating one or more of the foreign key attributes in a relationship. For example, a frequent query for DreamHome is to list all the private property owners at a branch using an SQL query of the following form:

SELECT o.Iname

FROM PropertyForRent p, PrivateOwner o

WHERE p.ownerNO = o.ownerNo AND branchNo = ‘BOO3’;<

Based on the orignal data shound in Figure 19.1. In other words, because there is no direct relationship between PrivateOwner and Branch, then to get the list of owners we have to use the PropertyForRent relation to gain acess to the bra nch number, branchNo. We can remove the need for this join by duplciating the foreign key bra nchNo in the PrivateOwner relation; that is, we introduce direct relationship between the Branch and PrivateOwner relations. In this case, we can simplify the SQL query to:

SELECT o.Iname

FROM PrivateOwner o

WHERE branchNo = ‘BOO3’;

Based on the revised relation diagram and PrivateOwner relation shound in Figure 19.6 if this change is made, it will be ncessary to introduce additional foreign key constraints, as discussed in Step 2.2

If an owner could rent properties through many branches, the privouis change would not work. In this case, it would be necessary to model as many-to-m,any (\*:\*) relationship between Branch and PrivateOwner. Note also that the PropertyForRent relation has the bra nchNo attribute because it is possible for a property not th ave a member of staff allocated to it, particularly at the start when the property is first taken on by the agency. If the PropertyForRent relation did not have the branch number, it would be necessary to join the PropertyForRent relation to the Staff relation based on the staffNo attribute to get the required branch number. The original SQL query would then become:

SELECT o.lNAME

FROM Staff s, PropertyForRent p, PrivateOwner o

WHERE s.staffNo = p.staffNo AND p.ownerNo = o.ownerNo AND s.branchNO = ‘BOO3’

Removing two joins from the query may provide greater justification for creating a direct relationship between PrivateOwner and Branch and thereby duplicating the foreign key branchNo in the PrivateOwner relation

Step 7.4 Duplciating attributes in many-to-many (\*:\*) relationships to reduces joi ns

During logical database design, we mapped each \*:\* relationship into three relations: the two relations derived from the orignal entities and a new relation representing the relationship between the two eneties. Now, if we wish to produce information from the \*:\* relationship, we have to join these three relations. In some circumstances, it may be possible to reduce the number of relations to be joined by duplcating attributes from one of the original enties in the intermediate relation. For example, the \*:\* relationship between Client and PropertyForRent has been decomposed by introducing the intermediate Viewing relation. Consider the requirement that the DreamHome slaes staff should contact clients who have still to make a comment on the properties they have viewed. Howerver, the sales staff need only the street attribute of the property when talking to the client. The required SQL query is:

SELECT p.street, c.\*, v.viewDate

FROM Client c, Viewing v, ProeprtyForRent p

WHERE v.propertyNo = p.propertyNo AND c.clientNO = v.clientNo AND comment IS NULL;

Based on the reltion model and sample data shoun in Figure 19.1 If we duplicate the street attribute in the intermediate Viewing relation, we cane remove the PropertyForRent relation from the query giving the SQL query:

SELECT c.\*, v.street, v.viewDate

FROM Client c, Viewing v

WHERE c.clientNo = v.clientNO AND comment IS NULL;

Based on the revised Viewing relation shound in Figure 19.7

Step 7.5 introducing repeating groups

Repeating goups were eliminated from the logical data model as a result of the requirement that all entities be in first normal form. Repeatin groups were seperated out into a new relation, forming a 1:\* relationship with the original (parent) relation. Occasionally. Reintroducing repeating groups is an effective way to improve system performance. For example, each DreamHome branch office has a maximum of three telephone number, although all offices do not necessarly have the same number of lines. In the logical data model, we created a Telephone entity with a three-to-one (3:1) relationship with Branch, resulting in two relations, as shound in Figure 19.1

If acess to this information is important or frequent, it may be more efficent to combiner the relations and store the telphone detail in the original Branch relation with one attribute for each telephone ,as shound in Figure 19.8.

In general, this type of denormalization should be considred only in the following circumstances:

The absolute number of items in the repeating group is known (in this example there is a maximum of three telephone numbers);

The number is static and will not change over time ( the maximum number of telephone lines is fixed and is not expected to change);

The number is not very large, typically not greater than 10, although this is not as important as the first two conditions.

Sometimes it may be only the most recent or current alue in a repaeting group, or just the fact that there is a repeating group, ths is needed most frequently. In the previous example we may choose to store one telephone number in the Branch relation and leave the remaining numbers for the Telephone relation. This would remove the presecne of nulls form the Branch relation, as each branch must have at least one telephone number.

Step 7.6 Creating extract tables

There may be situations where reports have to be run at peak times during the day.

These reports acess derived data and perform multirelation joins on the same set of base relations. Howerver, the data the report is based on may be relatively static or, in some ase, may not have to current (that is, if the data is a few hous old, the report would be perfectly acceptable), In this case, it may be possible to create a single, highly denormalized extract table based on the relations required by the reports, and allow the users to acess the e xtract direactly instead othe te base relations. The most common technique for producing extract tables is to crfeate and populate the table in an overnight batch run when the system is lightly loaded.

Step 7.7 Partitioning relations

Rather than combining relations together, an alternative approach that adresses the key problem with supporting very large relations (and indexes) is to decompose them into a number of smaller and more manageable pieces called partitions. As illustrated in Figure 19.9, there are two main types of partiioning: horizontal partitioning and vertical partiioning

Horizontal partitionning: Distribute the tuples of a rleation across a number of (smaller) relations

Vertical Partitionning: Distribute the attributes of a relation across a number of (smaller) relations ( the primary key is duplicated to allow the original relation to be reconstructed).

CREATE TABLE archivedPropertyForREntPArition(

propertyNo VARCHAR2(6) NOT NULL’

street VARCHAR2(25) NOT NULL’

city VARCHAR2(15) Not NULL,

postcode VARCHAR2(8).

Type CHAR NOT NULL

Rooms SMALLINT NOT NULL,

Rent NUMBER(6,2) NOT NULL

OwnerNO VARCHAR2(5) NOT NULL,

staffNO VARCHAR2(5)

branchNo CHAR(4) NOT NULL

PRIMARY KEY (propertyNO),

FOREIGN KEY (ownerNO) REFERENCES PrivateOWNer(ownerNO),

FOREIGN KEY (staffNo) REFERENCES Staff(staffNO),

FOREIGN KEY (branchNO) REFERENCES Branch(branchNO))

PARTITION BY HASH (branchNo)

(PARTITION b1 TABLESPACE TB01,

PARTITION b2 TABLESPACE TB02,

PARTITION b3 TABLESPACE TB03,

PARTITION b4 TABLESPACE TB04);

Partition are particulary useful in applications that store and analyse large amounts of data. For example, DreahHome maintains an ArchivedPropertyForRent relation with several hundres of thousands of tuples that are held indefinitely, for analys purposes. Searching for a particular tuple at a b ranch could be quite time consuming; howerver, we could reduce this time by horizontally partiioning the relation, with one partiion for each bran ch. We can create a (hash) partition for this scenario in Oracle using the SQL statement shound in Figure 19.10.

As well as hash partiioning, other common type of poartioning are range (each partition is defined by a range of values for one or more attributes) and list (each partition is defined by a laist of values for an attribute). There are also composite partitions, such as ranage-hash and list-hash (each partition is defined by a range or a list of values and then each partition is further subdivied based on a hash function),

There may also be circumstances in which we frequently examine particular attributes of a very large relation and it may be appropraite to vertically partition the relation into those attributes that are frequently accessed together and another vertical partition for the reamining attributes ( with the primary key replicated in each partition to allow the original relation to be reconstructed using a join).

Partition has a number of advantages:

Improved load balancing: Partitions can be alocated to different areas of secondary storage thereby permitting parallel access whiile at the same time minimizing the contention for acess to the same storage area if the relation was not partionned.

Improved performance. By limiting the amount of data to be examined or processed, and by enabling parallel execution, performance can be enhanced.

Increas availability. If partiions are allocated to different storage area and one storage area were to become unavailable, the other partitions would still be available.

Improved recovery, Smaller Partiions can be recovered more efficiently (equally well, the DBA may find backing up smaller partitions easier than backing up very large relations),

Security, Data in a partiion can be restricted to those users wo require access to it, with different partitions having different access restrictions.

Paritioning can also have a number of disadvantages:

Complexity, Partitioning is not usually transparent to end-users and queries that utilize more than one partition become more complex to write

Reduced performance, Queries that combine data from more than one partition may be slower than nonpartiionned approach

Deuplication, Vertical partitioning involves duplication of the primary key. This leads not only to increased storage requirements but also to potential inconsistencies arising.

Consider implications of denormalization

Consider the implications of denormalization on the previous steps in the methodology. For example, it may be necessary to reconsider the choice of index on the relations that have been denormalized to establich whether existing indexes should be removed or additional indexes added. In addition it will be necessary to consider how data integrity will be maintained. Common solutions are:

Triggers: Triggers can be used to auotmate the updating of derived or duplciated data ( see Section 8.3)

Transactions. Build into each application transactions that make the updates to denormalized data as single (atomic) action.

Batch reconcilation. Run batch programs at appropriate times to make the denormalized data consistent.

In terms of maintaining integrity, triggers provide the best solution, although they can cause performance problems. The advantage and disadvantage of denormalization are summarized in Table 19.1

|  |  |
| --- | --- |
| ADVANTAGES | DISADVANTAGES |
| Can improve performance by: | May speed up retriavals but can slow down updates |
| Precomputing derived data; | Always application-specific and needs to be re-evaluated if the applications changes |
| Minimizing the need for joins | Can increase the size of relations |
| Reducing the number of foreign keys in relations | May simplfiy implementation in some cases but may make it more complex in others |
| Reducing the number indexes (thereby saving storage space) | Sacrifices flexibilty |
| Reducing the number of relations |  |

Document introduction of redundancy

The introduction of redundancy should be fully documented, along with the reasons for introducing it. In particular, document the reasons for selecting one apparoach where many alternatives exists. Update the logical data model to reflect any changes made as a result of denormalization.

# 19.2 Monitoring the System to Improve Performance

Step 8: Monitor and Tune the Opertional System

Objective: To monitor the operational system and improve the performance of the system to correct inappropriate design decisions or rflect changing requirements

For this activity, we should remember that one of the main objectives of physical database design is to store and acess data in an efficeint way (see Appendix F)

There are a number of factors that we may use to measure efficientcy:

Transaction throughput. This is the nubmer of transactions that can be processed in a given time interval. In some systems, such as airline reservations, high transaction throughput is critical to the overall success of the system

Reponses time, This is the elapsed time for the completion of a single trsaction. From a user’s point of view, we want to minimize responses time as much as possible. However, there are some factors that influence respones time that the designer may have no control over, such as system loading or communication times. Respones time can be shortened by:

Reducing contention and wait times, particularly disk I/O wait times;

Reducing the amount of time for which ressource are required;

Using faster components.

Disk storage This is the maont of disk space required to store the database files The signer may wish to minimize the amount of disk storage used.

Howerver, there is no one factor that’s is always correct. Typically, the designer must trade one factor off against another to achieve a reasonable balance. For exemple, increasing the amount of data stored may decrease the response time or transaction throughput. The initial physical database design has been implemented. It will be necessary to monitor the system and tune it as a result of observed performance and changing requirements (see Step 8). Many DBMS provide the DBA with utilities to monitor and tune the opreation of the system.

There are many benefits to be gained from tuning the database.

Tuning can avoid the procurement of additional hardware.

It may be possible to downsize the hardware configuration. This results in les and cheaper hardware and consequently less expensive maintenance

A well-tuned system produces faster respones times and better throughput, which in turn makes the users – and hence the organisation – more productive.

Improved respones times can improve staff morale.

Improved respones times can increase customer satisfcation.

There last two benefits are less tangible than the others. However, we can certainly state that slow reponses times may demoralize staff and potentially lose customers.

To tune an operationnal system, the physical database designer must be aware of how the various hardware components interact nad affect database performance, as we now discuss.

Understanding system ressources

Main memory Main memoery acesses are signifanctly faster than secondary storage accesses --- sometimes tens or even hudnres of thousands of times faster. In general, the more main memoery avaialble to the DBMS and the database applications., the faster the applications will run. However, it is sensible always to have a minimum of 5% of main memory available. It is also advisable not to have any more than 10% avaliable; otherwise, main memoery is not being used optimally. When there is insuficient memoery to accommodate all processes, the operating system trnasfers pages of porcesses to disk to free up memoery. When one of these pages is next required the operating system has to trnasfer it back from disk. Sometimes it is necessary to swarp entire processes from memory to disk and back again to free up memory. Problems occur with main memory when pading or swapping becomes excessive.

To ensure efficent usage of main memory, it is necessary to understand how the garget DBMS uses main memory, what buffers it keeps in main memory, what paramters exist to allow the isze of the buffers to be ajusted, and so on For example, Oracle keeps a data dictionary cache in main memory that ideally should be large enough to handle 90% of data dictionary acceses without having to retrive the information from disk. It is also necessary to understand the acess patterns of users: an increase in the number of conccurent users acessing the database will result in an increase in the amount of memory being utilized.

CPU The CPU controls the tasks of the other system ressources and executes user processes, and is the most costly ressource in the system, so it needs to be correctly utilized. The main objective ofr this component is to prevent cPU contetion in which processes are waiting ofr the CPU. CPU bottlenecks occur when either the operating system or user processes make too many demands on the CPU. This is often a result of excessive paging.

It is necessary to understand the typical workload through a 24-hours period and ensure that sufficient ressources are avaiable for not only the normal workload but also the peak workload (ofr example, if the system has 90% CPU utilization and 10% idle during the normal workload then there may not be sufficent scope to handle the peak workload) One options is to ensure that during peak load no uncessary jobs are being run and that such jobs are instead run in off hours. Another option may be to oncisder multiple CPUs, which allows processing to be distributed and operations to be performed in parallel.

CPU MIPS (millions of instructions per second) can be used as a guide in comparing platforms and determining their ability ot meet the enterprise’s throughput requirements

Disk I/O With any large DBMS, there is a significant amount of disk I/O involved in storing and retrieving data. Disks usually have a recommended I/O rate and when this rate is exceeded, I/O bottlenecks occur. Although CPU clock speeds have increaseed dramatically in recent years, I/O speeds have not increased proportionately. The way in which data is organised on disk can have a major impact on the overall disk performance. One problem that can arise is disk contention. This occurs when multiple processes try to acess the same disk simultaneously. Most disk have limtis on both the number of acesses and the amont of data they can transfer per second and when these limtis are reached, processes may have to wait to acess the disk. To avoid this, it is recommended that storage should be evenly distributed across available drives to reduce the likelihood of performance problems occuring. Figure 19.11 illustrates the basic prinicples of distributing the data across disks:

The opreating system files should be separted from the database files;

The main database files should be separated from the index files;

The recovery log file ( see Sections 22.3.3) should be separated from the rest of the database

If a disk still appears to be overloaded, one or more of its heavily acessed files can be moved to a less active disk (this is known as distributing I/O) Load balancing can be achived by applying this principle to each of the disks until they all have approximately the same amount of I/O. Once again, the physical database designer needs to understand how the DBMS operates, the cahracteristics of the hardware and the acess patterns of the uses.

Disk I/O has been revolutionized with the introduction of RAID (Redundant Array of Indepedendent Disks) technology. RAID Works on having a large disk array comprising and arrangement of several independent disks that are organised to increase performance and at the same time imprive reliability. We discuss RAID in Section 20.2.7

Network When the maount of traffic on the network is too great, or when the nubmer of network collisions is large, netowkr bottlenecks occur.

Each of the previous ressources may affect other system ressources. Additionnaly an improvement in one resource may effect an improvement in other system resources. For example:

Procuring more main memory should result in less paging, which should help avoid CPU bottlenecks

More effective use of main memoery may result in less disk I/O

Summary: Tuning is an activity that is never complete. Throughout the life of the system, it will be necessary to monitor performance, particularly to acccount for changes in the environment and user requirements. However, making a change to one area of an operational system to improve performance may have an adverse effect on another area. For example, adding an index to a relation may improve the performance of one transaction, but it may adversely affect another, perhaps more important, transaction. Therefore, care must be taken when making changes to an operational system. If possible, test the changes either on a test database, or alternatively, when the system is not being fully used (such as outside of wokring hours).

Document tuning activity

The mechanis, used to tune the system should be fully documented, along with the reasons for tuning it in the closen way. In particular, document the reasons for sleecting one approach where many alternatives exists.

New requirement for DreamHome

As well as tunign the system to maintain optimal performance, it may also be necessary to handle changing requirements. For example, suppose that after some months as a faully operational database, several users of the DreamHome system raise two new requirements:

1. Ability to hold picture of the properties for rent, together with commetns that describe the main features of the property.

In microsoft Office Acess we are able to accommodate this request using OLE (Object Linking and Embedding) fields, which are used to store data such as Microsoft Word or Microsoft Excel documents, pictures, sound, and other types of binary data created in other programs. OLE objects can be linked to or embedded in a field in a Microsoft Office Acess table and then displayed in a form or report.

To implement this new requirement, we restructure the PropertyForRent table to include:

1. A field called picture specified as an OLE data type; this field holds graphical image of properties, created by scanning photographs of the properties for rent and saving the images as BMP (bitmapped\_ graphic files;
2. a field called comments specified as a Memo data type, capable of storing lengthy text.

A form based on some fields of the PropertyForRent table, including the nw fields, is shown in Fgiure 19.12 The main problem associated with the storage of graphic images ithe large amount of disk space required to store the image files. We would therefore need to contiinue to monitor the perfromance of the DreamHome database to ensure that satisfying this new requirement does not compromise the system’s performance.

1. Ability to publish a report describing properties avaialble for rent on the Web

This requirement can be accommodated in both Microsoft Office Acess and Oracle, as both DBMS provide many feactures for developing a Web application and publishing on the Internet. However, to use these features, we require a Web browser, such as Microsoft Internet Explorer or Mozilla Firefox, and a modren or other network connection to access the Internet. In Cahpter 29, we describe in detail the technologies used in the intergration of database and the Web.

1. Enable property owners to acess details of their properties and commetns made by prospective client on the Web.

This requirement is similar to the previous requirement in term of technology, but also requires that the dtabase is restructured to allow for storage of a property owner’s login detail, which include email and password. Once logged in, a property owner will be able to view only the detail relating to their properties. (The client relation restructured to store a property owner’s login details is shoun in Figure 4.3)

Chapter Summary

Formally, the term denormalization refers to a refinement to the realtion schema such that the degrees of normalization for a modified relation is less than the degree of at least one of the original relations. The term is also used more loosely to refer to situations in which two relations are combined into one new relations and the new relations is still normalized but contains more nulls than the oringal relations.

Step 7 of physical database design includes a consideration of whether to denormalize the relational schema to improve performance. There may be circumstance in which it may be necessary to accept the loss of some of the benefits of a fully normalized design in favor of performance. This options should be considered only when it is estimated that the system will not be able to meet its performance requirements. As a rule of thumb if performance is unsatisfacty and a relation has low update rate and a very high query rate, denormalization may be a viable option.

The final step (Step 8) of physical database design is the ongoing process of monitoring and tuning the operational system to achive maximum performance

One of the main ojbectives of physical database design is to store and acess data in an efficeint way. There are a number of facots that can be used to mesure efficiency, including throughput, respones time, and disk storage.

To imprive performance it is necessary to be ware of how the following four basic hardware components interact and affect system performance main memory, CPU, disk I/O and network.

Review Questions

19.1 Describe that purpose of the main steps in the physical design methodology presented in this chpaer

19.2 Under what circumstance would you want to denormalzie a logical data model& Use examples to illustrate your answer

19.3 What factors can be used to measure efficiency?

19.4 Discuss how the four basic har dware components interact and affect system performance

19.5 How should you distribute data across disks?

Exercices

19.6 Investiage whether your DBMS can accommodate the three new requirements for the DreamHome case study given in Step 8 of this cahpter. If feasible produce a design for the tree requirements and implement them in your target DBMS.