Chapter 12

Query Processing

12.1 Query Interpretation

1. Why do we need to optimize?
   - A high-level relational query is generally non-procedural in nature.
   - It says "what", rather than "how" to find it.
   - When a query is presented to the system, it is useful to find an efficient method of finding the answer, using the existing database structure.
   - Usually worthwhile for the system to spend some time on strategy selection.
   - Typically can be done using information in main memory, with little or no disk access.
   - Execution of the query will require disk accesses.
   - Transfer of data from disk is slow, relative to the speed of main memory and the CPU
   - It is advantageous to spend a considerable amount of processing to save disk accesses.
2. Do we really optimize?
   - Optimizing means finding the best of all possible methods.
   - The term "optimization" is a bit of a misnomer here.
   - Usually the system does not calculate the cost of all possible strategies.
   - Perhaps "query improvement" is a better term.
3. Two main approaches:
   (a) Rewriting the query in a more effective manner.
   (b) Estimating the cost of various execution strategies for the query.

   Usually both strategies are combined.
   - The difference in execution time between a good strategy and a bad one may be huge.
   - Thus this is an important issue in any DB system.
   - In network and hierarchical systems, optimization is left for the most part to the application programmer.
   - Since the DML language statements are embedded in the host language, it is not easy to transform a hierarchical or network query to another one, unless one has knowledge about the entire application program.

- As a relational query can be expressed entirely in a relational query language without the use of a host language, it is possible to optimize queries automatically.
- SQL is suitable for human use, but internally a query should be represented in a more useful form, like the relational algebra.

4. So, first the system must translate the query into its internal form. Then optimization begins:
   - Find an equivalent expression that is more efficient to execute.
   - Select a detailed strategy for processing the query. (Choose specific indices to use, and order in which tuples are to be processed, etc.)
5. Final choice of a strategy is based primarily on the number of disk accesses required.

12.2 Equivalence of Expressions

1. The first step in selecting a query-processing strategy is to find a relational algebra expression that is equivalent to the given query and is efficient to execute.
2. We'll use the following relations as examples:

   Customer-scheme = (cname, street, ccity)
   Deposit-scheme = (bname, account#, name, balance)
   Branch-scheme = (bname, assets, bcity)

   We will use instances *customer, deposit* and *branch* of these schemes.

12.2.1 Selection Operation

1. Consider the query to find the assets and branch-names of all banks who have depositors living in Port Chester. In relational algebra, this is

   $$\Pi_{bname,assets}(\sigma_{ccity=\text{“}Port\ Chester\text{”}}\ (customer \bowtie deposit \bowtie branch))$$

   - This expression constructs a huge relation,

     $customer \bowtie deposit \bowtie branch$

     of which we are only interested in a few tuples.
   - We also are only interested in two attributes of this relation.
   - We can see that we only want tuples for which *ccity = "Port Chester"*.
   - Thus we can rewrite our query as:

     $$\Pi_{bname,assets}((\sigma_{ccity=\text{“}Port\ Chester\text{”}}(customer)) \bowtie deposit \bowtie branch)$$

   - This should considerably reduce the size of the intermediate relation.
2. **Suggested Rule for Optimization:**
   - Perform select operations as early as possible.
   - If our original query was restricted further to customers with a balance over $1000, the selection cannot be done directly to the customer relation above.
   - The new relational algebra query is

     $$\Pi_{bname,assets}\ (\sigma_{ccity=\text{“}PortChester\text{”}\ \wedge\ balance>1000}\ (customer \bowtie deposit \bowtie branch))$$

   - The selection cannot be applied to *customer*, as *balance* is an attribute of *deposit*.
   - We can still rewrite as

     $$\Pi_{bname,assets}\ ((\sigma_{ccity=\text{“}Port\ Chester\text{”}\ \wedge\ balance>1000}\ (customer \bowtie deposit)) \bowtie branch)$$

- If we look further at the subquery (middle two lines above), we can split the selection predicate in two:

  $\sigma_{ccity=\text{“}Port\ Chester\text{”}}(\sigma_{balance>1000}$
  $(customer \bowtie deposit))$

- This rewriting gives us a chance to use our “perform selections early” rule again.
- We can now rewrite our subquery as:

  $\sigma_{ccity=\text{“}Port\ Chester\text{”}}(customer) \bowtie \sigma_{balance>1000}(deposit)$

3. **Second Transformational Rule:**

   - Replace expressions of the form $\sigma_{P_1 \wedge P_2}(e)$ by $\sigma_{P_1}(\sigma_{P_2}(e))$ where P_1 and P_2 are predicates and e is a relational algebra expression.
   - Generally,
     $\sigma_{P_1}(\sigma_{P_2}(e)) = \sigma_{P_2}(\sigma_{P_1}(e)) = \sigma_{P_1 \wedge P_2}(e)$

12.2.2 Projection Operation

1. Like selection, projection reduces the size of relations.

   It is advantageous to **apply projections early**. Consider this form of our example query:

   $\Pi_{bname,assets}\ (((\sigma_{ccity=\text{“}Port\ Chester\text{”}}(customer)) \bowtie deposit) \bowtie branch)$

2. When we compute the subexpression

   $((\sigma_{ccity=\text{“}Port\ Chester\text{”}}(customer)) \bowtie deposit)$

   we obtain a relation whose scheme is
   (*cname, ccity, bname, account#, balance*)

3. We can eliminate several attributes from this scheme. The only ones we need to retain are those that

   - appear in the result of the query **or**
   - are needed to process subsequent operations.

4. By eliminating unneeded attributes, we reduce the number of columns of the intermediate result, and thus its size.

5. In our example, the only attribute we need is *bname* (to join with *branch*). So we can rewrite our expression as:

   $\Pi_{bname,assets}\ ((\Pi_{bname}((\sigma_{ccity=\text{“}Port\ Chester\text{”}}(customer)) \bowtie deposit)) \bowtie branch)$

6. Note that there is no advantage in doing an early project on a relation before it is needed for some other operation:

   - We would access every block for the relation to remove attributes.
   - Then we access every block of the reduced-size relation when it is actually needed.
   - We do more work in total, rather than less!

12.2.3 Natural Join Operation

1. Another way to reduce the size of temporary results is to choose an optimal ordering of the join operations.
2. Natural join is associative:

   $(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$

3. Although these expressions are equivalent, the costs of computing them may differ.

- Look again at our expression

  $\Pi_{bname,assets}\ ((\sigma_{ccity=\text{“}Port\ Chester\text{”}}(customer)) \bowtie deposit \bowtie branch)$

- we see that we can compute $deposit \bowtie branch$ first and then join with the first part.
- However, $deposit \bowtie branch$ is likely to be a large relation as it contains one tuple for every account.
- The other part,

  $\sigma_{ccity=\text{“}Port\ Chester\text{”}}(customer)$

  is probably a small relation (comparatively).
- So, if we compute

  $\sigma_{ccity=\text{“}Port\ Chester\text{”}}(customer) \bowtie deposit$

  first, we get a reasonably small relation.
- It has one tuple for each account held by a resident of Port Chester.
- This temporary relation is much smaller than $deposit \bowtie branch$.

4. Natural join is commutative:

   $r_1 \bowtie r_2 = r_2 \bowtie r_1$

   - Thus we could rewrite our relational algebra expression as:

     $\Pi_{bname,assets}\ (((\sigma_{ccity=\text{“}Port\ Chester\text{”}}(customer)) \bowtie branch) \bowtie deposit)$

   - But there are no common attributes between *customer* and *branch*, so this is a **Cartesian product**.
   - Lots of tuples!
   - If a user entered this expression, we would want to use the associativity and commutativity of natural join to transform this into the more efficient expression we have derived earlier (join with *deposit* first, then with *branch*).

12.2.4 Other Operations

1. Some other equivalences for union and set difference:

   $\sigma_P(r_1 \cup r_2) = \sigma_P(r_1) \cup \sigma_P(r_2)$
   $\sigma_P(r_1 - r_2) = \sigma_P(r_1) - r_2 = \sigma_P(r_1) - \sigma_P(r_2)$
   $(r_1 \cup r_2) \cup r_3 = r_1 \cup (r_2 \cup r_3)$
   $r_1 \cup r_2 = r_2 \cup r_1$

2. **Further Optimization**

   Our text covers only some of the possible operations. (For a more complete list, see Elmasri & Navathe, page 518.)

   Also, it makes sense to combine the implementation of various sets of operations in order to reduce the size of intermediate relations:

   - Combine projects and selects with a Cartesian product or natural join.
   - The idea is to do the selection and/or projection while computing the join.
   - This saves computing a large intermediate relation that is going to be subsequently reduced by the select or project anyway.

12.3 Estimation of Query-Processing Cost

1. To choose a strategy based on reliable information, the database system may store statistics for each relation r:
   - n_r - the number of tuples in r.
   - s_r - the size in bytes of a tuple of r (for fixed-length records).
   - $V(A, r)$ - the number of distinct values that appear in relation r for attribute A.
2. The first two quantities allow us to estimate accurately the size of a Cartesian product.
   - The Cartesian product $r \times s$ contains $n_r n_s$ tuples.
   - Each tuple of $r \times s$ occupies $s_r + s_s$ bytes.
   - The third statistic is used to estimate how many tuples satisfy a selection predicate of the form
     $<$*attribute-name*$>$ = $<$*value*$>$
   - We need to know how often each value appears in a column.
   - If we assume each value appears with equal probability, then
     $\sigma_{A=a}(r)$
     is estimated to have
     $$\frac{n_r}{V(A, r)}$$
     tuples.
   - This may not be the case, but it is a good approximation of reality in many relations.
   - We assume such a uniform distribution for the rest of this chapter.
   - Estimation of the size of a natural join is more difficult.
   - Let $r_1(R_1)$ and $r_2(R_2)$ be relations on schemes R_1 and R_2.
   - If $R_1 \cap R_2 = \emptyset$ (no common attributes), then $r_1 \bowtie r_2$ is the same as $r_1 \times r_2$ and we can estimate the size of this accurately.
   - If $R_1 \cap R_2$ is a key for R_1, then we know that a tuple of r_2 will join with exactly one tuple of r_1.
   - Thus the number of tuples in $r_1 \bowtie r_2$ will be no greater than n_{r_2}.
   - If $R_1 \cap R_2$ is **not** a key for R_1 or R_2, things are more difficult.
   - We use the third statistic and the assumption of uniform distribution.
   - Assume $R_1 \cap R_2 = \{A\}$.
   - We assume there are
     $$\frac{n_{r_2}}{V(A, r_2)}$$
     tuples in r_2 with an A value of $t[A]$ for tuple t in r_1.
   - So tuple t of r_1 produces
     $$\frac{n_{r_2}}{V(A, r_2)}$$
     tuples in $r_1 \bowtie r_2$
3. Considering **all** the tuples in r_1, we estimate that there are
   $$\frac{n_{r_1} n_{r_2}}{V(A, r_2)}$$
   tuples in total in $r_1 \bowtie r_2$

4. If we reverse the roles of r_1 and r_2 in this equation, we get a different estimate

$$\frac{n_{r_1} n_{r_2}}{V(A, r_1)}$$

if $V(A, r_1) \neq V(A, r_2)$.

- If this occurs, there are likely to be some dangling tuples that do not participate in the join.
- Thus the lower estimate is probably the better one.
- This estimate may still be high if the $V(A, r_1)$ values in r_1 have few values in common with the $V(A, r_2)$ values in r_2.
- However, it is unlikely that the estimate is far off, as dangling tuples are likely to be a small fraction of the tuples in a real world relation.

5. To maintain accurate statistics, it is necessary to update the statistics whenever a relation is modified.

   This can be substantial, so most systems do this updating during periods of light load on the system.

12.4 Estimation of Access Costs Using Indices

1. So far, we haven't considered the effects of indices and hash functions on the cost of evaluating an expression.

   - Indices and hash functions allow fast access to records containing a specific value on the index key.
   - Indices (but not most hash functions) also allow the records of a file to be read in sorted order.
   - It is efficient to read records of a file in an order corresponding closely to physical order.
   - If an index allows this, we call the index a **clustering index**.
   - Such indices allow us to take advantage of the physical clustering of records into blocks.
   - Text doesn't distinguish clearly between a **clustering index** and a **primary index**. [ELNA89] define a primary index as one on the primary key where the file is sorted on that key, and a clustering index as one on non-primary key attribute(s) that the file is sorted on.
   - With this definition, for a primary index, there is only one tuple per search key value, while for a clustering index there may be many tuples.
   - In both cases, only one pointer is needed per search key value. (Why?)

2. Detailed strategy for processing a query is called the **access plan**. This includes not only the relational operations to be performed, but also the indices to be used and the order in which tuples are to be accessed and the order in which operations are to be performed.

3. The use of indices imposes some overhead (access to blocks containing the index.) We also must take this into account in computing cost of a strategy.

4. We'll look at the query

   select *account#*
   from *deposit*
   where *bname* = "Perryridge"
   and *cname* = "Williams"
   and *balance* > 1000

5. We assume the following statistical information is kept about the *deposit* relation:

   - 20 tuples of *deposit* fit in one block.
   - V(*bname*, *deposit*) = 50.
   - V(*cname*, *deposit*) = 200.

- V(*balance*, *deposit*) = 5000.
- $n_{deposit} = 10,000$ (number of tuples).

6. We also assume the following indices exist on deposit:
   - A clustering B^+-tree index for *bname*.
   - A nonclustering B^+-tree index for *cname*.
7. We also still assume values are distributed uniformly.
   - As V(*bname*, *deposit*) = 50, we expect 10,000/50 = 200 tuples of the *deposit* relation apply to Perryridge branch.
   - If we use the index on *bname*, we will need to read these 200 tuples and check each one for satisfaction of the rest of the **where** clause.
   - Since the index is a clustering index,
     200/20 = 10 block reads are required.
   - Also several index blocks must be read.
   - Assume the B^+-tree stores 20 pointers per node.
   - Then the B^+-tree must have between 3 and 5 leaf nodes (to store the 50 different values of bname).
   - So the entire tree has a depth of 2, and at most 2 index blocks must be read.

   So the above strategy requires 12 block reads.
8. **Note:** another way of calculating the number of levels in a B^+-tree is to remember that the height is no greater than
   $$1 + \lfloor \log_{\lceil n/2 \rceil}(K/2) \rfloor$$
   where there are K search key values in the relation, and n is the number of pointers in a node.
9. You can use the **change of base formula** to calculate this value using a log function of base x with your calculator:
   $$\log_b K = \frac{\log_x K}{\log_x b}$$
10. If we use the index for *cname*, we estimate the number of block accesses as follows:
    - Since V(*cname*, *deposit*)=200, we expect that 10,000/200 = 50 tuples pertain to Williams.
    - However, as the index on *cname* is nonclustering, we can expect that 50 block reads will be required, plus some for the index (as before).
    - Assume that 20 pointers fit into one node of a B^+-tree index.
    - As there are 200 customer names, the tree has between 11 and 20 leaf nodes.
    - So the index has a depth of 2 (says the text), and 2 block accesses are required to read the index blocks. (Actually, depth could be 3 — can you see how?)
    - This strategy requires a total of 52 block accesses.
    - So we conclude that it is better to use the index on *bname*.
11. If both indices were non-clustering, which one would we choose?
    - We only expect 50 tuples for *cname* = "Williams", versus 200 tuples with *bname* ="Perryridge".
    - So without clustering, we would choose the *cname* index as it would require reading and inspecting fewer tuples.
12. Another interesting method is to look at pointers first:

- Use the index for *cname* to retrieve **pointers** to records with *cname* = "Williams", rather than the records themselves.
- Let P_1 denote this set of pointers.
- Similarly, use the index on *bname* to obtain P_2, the set of pointers to records with *bname* = "Perryridge".
- Then $P_1 \cap P_2$ is the set of pointers to records with *bname* = "Perryridge" and *cname* = "Williams".
- Only these records need to be retrieved and tested to see if *balance* > 1000.
- Cost is 4 blocks for both indices to be read, plus blocks for records whose pointers are in $P_1 \cap P_2$.
- This last quantity can be estimated from our statistics.
- As V(*bname*, *deposit*) = 50 and V(*cname*, *deposit*) = 200, we can expect one tuple in 50 * 200, or 1 in 10,000 to have both values we are looking for.
- This means that $P_1 \cap P_2$ is estimated to have only one pointer.
- So we only need to read 1 block, and total cost is 5 block reads.

13. We didn't use the *balance* attribute as a starting point because there is no index for *balance* and also the predicate involves a "greater than" comparison (> 1000).

14. Generally, equality predicates are more selective than "greater than" predicates, as they return fewer tuples.

15. Estimation of access cost using indices allows us to estimate the **complete** cost, in terms of block accesses, of a plan. It is often worthwhile for a large number of strategies to be evaluated down to the access plan level before a choice is made.

12.5 Join Strategies

1. We've seen how to estimate the **size** of a join. Now we look at estimating the **cost** of processing a join.

2. Several factors influence the selection of an optimal strategy:
   - Physical order of tuples in a relation.
   - Presence of indices and type of index (clustering or not).
   - Cost of computing a temporary index for the sole purpose of processing one query.

3. We'll look at computing the expression $deposit \bowtie customer$ assuming no indices exist. We also let
   $n_{deposit} = 10,000$ (number of *deposit* tuples)
   $n_{customer} = 200$ (number of *customer* tuples)

12.5.1 Simple Iteration

1. If we don't create an index, we must examine every pair of tuples t_1 in *deposit* and t_2 in *customer*. This means examining 10,000 * 200 = 2,000,000 pairs!

2. If we execute this query cleverly, we can cut down the number of block accesses. We use the following method:

```
for each tuple d ∈ deposit do
  begin
    for each tuple c ∈ customer do
      begin
        examine pair (d, c) to see if a
        tuple should be added to the result
      end
  end
```

3. - We read each tuple of *deposit* once.

- This could require 10,000 block accesses.
- The total number of block access, if the tuples are not stored together physically, would be 10,000 + 10,000 * 200 = 2,010,000.
- If we put customer in the outer loop, we get 2,000,200 accesses.
- If the tuples of *deposit* are stored together physically, fewer accesses are required (at 20 per block, 10,000/20 = 500 block accesses).
- We read each tuple of *customer* once for each tuple of *deposit*.
- This suggests we read each tuple of *customer* 10,000 times, giving as many as 2,000,000 accesses to read *customer* tuples!
- This would give a total of 2,000,500 accesses.
- We can reduce accesses significantly if we store *customer* tuples together physically.
- At 20 tuples per block, only 10 accesses are required to read the entire relation (as opposed to 200).
- Then we only need 10 * 10,000 = 100,000 block accesses for *customer*.
- This gives a total of 100,500.

4. Text says further savings are possible if we use *customer* in the outer loop.
   - Now we reference each tuple of *deposit* once for each tuple of *customer*.
   - If *deposit* tuples are stored together physically, then since 20 tuples fit on one block, $n_{deposit}/20 = 500$ accesses are needed to read the entire relation.
   - Since *customer* has 200 tuples, we read the *deposit* relation 200 times.
   - Earlier printings of the text say at most 200 * 500 = 10,000 block accesses to *deposit* are required.
   - **WRONG!** 200 * 500 is 100,000.
   - Total cost is then 100,000 for inner loop plus 10 accesses to read the *customer* relation once for a total of 100,010.
   - Compared to previous estimate of 100,500, the savings are small (490).
5. **Note that we are considering worst-case number of block reads, where every time a block is needed it is not in the buffer.**

   Good buffer management can reduce this considerably.

12.5.2 Block-Oriented Iteration

1. If we process tuples on a per-block basis we can save many accesses.

   The idea is that, if both relations have tuples stored together physically, we can examine all the tuple pairs for a block of each relation at one time. We still need to read all the tuples of one relation for a block of the other relation.

   The block method algorithm is:

```
for each block Bd of deposit do
 begin
  for each block Bc of customer do
   begin
    for each tuple d in Bd do
     begin
      for each tuple c in Bc do
       begin
        test pair (d, c) to see if a tuple
        should be added to the result
       end
     end
   end
 end
```

- Instead of reading the entire *customer* relation for each **tuple** of deposit, we read the entire *customer* relation once for each **block** of *deposit*.
- Since there are 500 blocks of *deposit* tuples and 10 blocks of *customer* tuples, reading *customer* once for each block of *deposit* requires 10 * 500 = 5000 accesses to *customer* blocks.
- Total cost is then 5000 + 500 (for accesses to *deposit* blocks) = 5500.
- This is obviously a significant improvement over the non-block method, which required roughly 100,000 or 2,000,000 accesses.
- Choice of *customer* for the inner loop is arbitrary, but does provide a potential advantage.
- Being the smaller relation, it may be possible to keep it all in main memory.
- If this was the case, we would only require 500 blocks to read *deposit* plus 10 blocks to read *customer* into main memory, for a total of 510 block accesses.

12.5.3 Merge-Join

1. Suppose neither relation fits in main memory, and both are stored in sorted order on the join attributes. (E.g. both *deposit* and *customer* sorted by *cname*.)
2. We can then perform a **merge-join**, computed like this:
   - Associate one pointer with each relation.
   - Initially these pointers point to the first record in each relation.
   - As algorithm proceeds, pointers move through the relation.
   - A group of tuples in one relation with the same value on the join attributes is read.
   - Then the corresponding tuples (if any) of the other relation are read.
   - Since the relations are in sorted order, tuples with same value on the join attributes are in consecutive order. This allows us to read each tuple only once.
3. In the case where tuples of the relations are stored together physically in their sorted order, this algorithm allows us to compute the join by reading each block exactly once.
4. - For $deposit \bowtie customer$, this is a total of 510 block accesses.
   - This is as good as the block-oriented method with the inner loop relation fitting into main memory.
   - The disadvantage is that both relations must be sorted physically.
   - It may be worthwhile to do this sort to allow a merge-join.

12.5.4 Use of an Index

1. Frequently the join attributes form a search key for an index on one of the relations being joined.
   - In such cases we can consider a join strategy that makes use of such an index.
   - Our first join algorithm is more efficient if an index exists on *customer* (inner loop relation) for *cname*.
   - Then for a tuple d in *deposit*, we no longer have to read the entire *customer* relation.
   - Instead, we use the index to find tuples in *customer* with the same value on *cname* as tuple d of *deposit*.
   - We saw that without an index, we could take as many as 2,010,000 block accesses.
   - Using the index, and making no assumptions about physical storage, the join can be performed more efficiently.
   - We still need 10,000 accesses to read *deposit* (outer loop relation).
   - However, for each tuple of *deposit,* we only need an index lookup on the *customer* relation.

- As $n_{customers} = 200$ and we assumed that 20 pointers fit in one block, this lookup requires at most 2 index block accesses (depth of B^+-tree) plus block access for the *customer* tuple itself.
- This means 3 block accesses instead of 200, for each of the 10,000 deposit tuples.
- This gives 40,000 total, which appears high, but is still better than 2,010,000.
- More efficient strategies required tuples to be stored physically together.
- If tuples are not stored physically together, this strategy is highly desirable.
- The savings (1,970,000 accesses) is enough to justify creation of the index, even if it is used only once.

12.5.5 Hash Join

1. Sometimes it may be useful to construct a "use once only" hash structure to assist in the computation of a single join.
   - We use a hash function h to hash tuples of both relations on the basis of join attributes.
   - The resulting buckets, pointing to tuples in the relations, limit the number of pairs of tuples that must be compared.
   - If d is a tuple in *deposit* and c is a tuple in *customer*, then d and c must be compared only if $h(d) = h(c)$.
   - The comparison must still be done as it is possible that d and c have different customer names that hash to the same value.
   - As before, we need our hash function to hash randomly and uniformly.
2. We will now estimate the cost of a *hash-join*.
   - Assume that h is a hash function mapping *cname* values to $\{0, 1, \ldots, max\}$.
   - $H_{c_0}, H_{c_1}, \ldots, H_{c_{max}}$ denote buckets of pointers to *customer* tuples, each initially empty.
   - $H_{d_0}, H_{d_1}, \ldots, H_{d_{max}}$ denote buckets of pointers to *deposit* tuples, each initially empty.
   - The hash-join algorithm is shown in figure 9.4.
   - The first two loops assign pointers to the hash buckets, requiring a complete reading of both relations.
   - This requires 510 block accesses if we assume that both relations are stored together physically (i.e. not clustered with other relations).
   - As buckets contain only pointers, we assume they fit in main memory, so no disk accesses are required to read the buckets.
   - Final loop of the algorithm iterates over the range of hash function h.
   - Assume i is a particular value in the range of h.
   - The tuples rd and rc are assembled, from the pointers, where rd is the set of *deposit* tuples that hash to bucket i, and rc is the set of *customer* tuples that hash to bucket i.
   - Then $rd \bowtie rc$ is calculated.
   - This join is done using simple iteration, since we expect rd and rc to be small enough to fit in main memory.
   - Since a tuple hashes to exactly one bucket, each tuple is read only once by the final outer loop.
   - Earlier printings of the text say this requires another 510 block accesses, for a total of 1020. **This is wrong!**
   - If there are 10,000 *deposit* tuples and 200 *customer* tuples, then reading every tuple once could take 10,200 accesses, worst-case.
   - Total is then 510 block accesses to create the buckets, plus 10,200 to assemble the buckets' records, giving 10,710.
   - Exercise: calculate the cost by other methods. What have we gained, and at what cost?

3. If the query optimizer chooses to do a hash-join, the hash function must be chosen so that
   - The range is large enough to ensure that buckets have a small number of pointers, so that *rd* and *rc* fit in main memory.
   - The range is not so large that many buckets are empty and the algorithm processes many empty buckets.

12.5.6 Three-Way Join

1. We now consider the strategies for computing a 3-way join:

   $branch \bowtie deposit \bowtie customer$

2. We'll assume that
   - $n_{deposit} = 10,000$
   - $n_{customer} = 200$
   - $n_{branch} = 50$
3. **Strategy 1:**
   - Compute $deposit \bowtie customer$ using one of the previous methods.
   - As *cname* is a key for *customer,* the result of this join has at most 10,000 tuples (the number in *deposit*).
   - If we then build an index on *branch* for *bname*, we can compute

     $branch \bowtie (deposit \bowtie customer)$

     by considering each tuple t of

     $(deposit \bowtie customer)$

     and looking up tuples in *branch* with the same *bname* value using the index.
   - (How many accesses, roughly?)
4. **Strategy 2:**
   - Compute the join without any indices.
   - This requires checking 50 * 10,000 * 200 possibilities = 100,000,000.
   - Not too good of an idea...
5. **Strategy 3:**
   - Perform the pair of joins at **once**.
   - Construct two indices:
     - One on *branch* for *bname*.
     - One on *customer* for *cname*.
   - Then consider each tuple t in *deposit*.
   - For each t, look up corresponding tuples in *customer* and in *branch* with equal values on respective join attributes.
   - Thus we examine each tuple of *deposit* exactly once.

   Using strategy 3 it is often possible to perform a three-way join more efficiently than by using two two-way joins.
6. It is hard to calculate exact costs for 3-way joins. Costs depend on how the relations are stored, distribution of values and presence of indices.

12.6 Join Strategies for Parallel Processors

1. In many cases, multiple processors may be available for parallel computation of the join.
2. There are many architectures, including database machines.
3. We consider only a simple architecture:
   - all processors have access to all disks, and
   - all processors share main memory. (Nonshared, i.e., distributed database systems which will be covered in Chapter 15).

12.6.1 Parallel-Join

1. Parallel-join: split the pairs to be tested over several processors. Each processor computes part of the join, and then the results are assembled (merged).
2. Ideally, the overall work of computing join is partitioned evenly over all processors. If such a split is achieved without any overhead, a parallel join using N processors will take $1/N$ times as long as the same join would take on a single processor.
3. In practice, the speedup is less dramatic because
   (a) Overhead is incurred in partitioning the work among the processors.
   (b) Overhead is incurred in collecting the results computed by each processor.
   (c) If the split is not even, the final result cannot be obtained until the last processor has finished.
   (d) The processors may compete for shared system resources, e.g., for $A \bowtie B$ (e.g., $deposit \bowtie customer$), if each processor uses its own partition of A, and the main memory cannot hold the entire B, the processors need to synchronize the access of B so as to reduce the number of times that each block of B must be read in from disk.
4. A parallel hash algorithm to reduce memory contention.

   Choose a hash function whose range is $\{1, \ldots, N\}$ which allows us to assign each of the N processors to exactly one hash bucket. Since the final outer for-loop of the hash-join algorithm iterates over buckets, each processor can process the iteration that corresponds to its assigned bucket. Since no tuple is assigned to more than one bucket, so there is no contention for B tuples. Since each processor considers one pair of tuples at a time, the total main memory requirements of the parallel hash join algorithm are sufficiently low that contention for space in main memory is unlikely.

12.6.2 Pipelined Multiway Join

1. Computing several joins in parallel.
2. Example. $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ can be computed by first computing "$t_1 \leftarrow r_1 \bowtie r_2$" and "$t_2 \leftarrow r_3 \bowtie r_4$", and then "$t_1 \bowtie t_2$".
3. Moreover, it can be computed in pipelined way: $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$. Processor P_1 is assigned to process $(r_1 \bowtie r_2)$, P_2 to $(r_3 \bowtie r_4)$, and P_3 to process the join of the tuples being generated by P_1 and P_2.

12.6.3 Physical Organization

1. In order to reduce contention for disk access, the database can be partitioned over several disks, allowing several disk accesses to be serviced in parallel.
2. In order to exploit the potential for parallel disk access, we must choose a good distribution of data among the disks.

3. For the parallel 2-way join, it is useful to distribute tuples of individual relations among several disks (disk stripping). For example, assign tuples to disks based on the hash function value of the hash-join algorithm. All groups of tuples that share a bucket are assigned to the same disk. Each group is assigned to the same disk, if possible, or the groups are distributed uniformly among the available disks. This allows the parallel 2-way hash-join to exploit parallel disk access.

4. For the pipeline-join, it is desirable that each relation be kept on one disk and the distinct relations be assigned to separate disks to the degree possible.

   For example, for computing $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$, if each relation is on a different disk, contention is eliminated between processors P_1 and P_2.

5. The optimal physical organization differs for different queries. The DBA must choose a physical organization that is believed to be good for the expected mix of database queries.

6. The query optimizer must choose from the various parallel and sequential techniques by estimating the cost of each technique on the given physical organization.

12.7 Structure of the Query Optimizer

1. These are only some of the many query-processing strategies used in database systems.

2. Most systems only implement a few strategies.

3. Some systems make a heuristic guess of a good strategy, in order to minimize the number of strategies to be considered.

4. Then the optimizer considers every possible strategy, but quits as soon as it determines that the cost is greater than the best previously considered strategy.

5. To simplify the strategy selection task, a query may be split into several sub-queries.

6. This simplifies strategy selection and permits recognition of common sub-queries (no need to compute them twice).

7. Examination of a query for common subqueries and the estimation of the cost of a large number of strategies impose a substantial overhead on query processing.

8. However, this is usually more than offset by savings at query execution time.

9. Therefore, most commercial systems include relatively sophisticated optimizers.