Notes on DBMS Internals

Neil Conway ⟨neil.conway@gmail.com⟩

November 10, 2003

Preamble

These notes were originally written for my own use while taking CISC-432, a course in DBMS design and implementation at Queen's University. The textbook used by that class is *Database Management Systems*, 3rd Edition by Raghu Ramakrishnan and Johannes Gehkre; some of the material below may be specific to that text.

This document is provided in the hope that it is useful, but I can't provide any assurance that any information it contains is in any way accurate or complete. Corrections or additions are welcome.

Distribution Terms: This document is released into the public domain.

Query Evaluation

External Sorting

- A DBMS frequently needs to sort data (e.g. for a merge-join, `ORDER BY`, `GROUP BY`, etc.) that exceeds the amount of main memory available. In order to do this, an *external sort* algorithm is used.
- *2-Way External Merge Sort:*
  - In the first pass, each page of the input relation is read into memory, sorted, and written out to disk. This creates N runs of 1 page each.
  - In each successive pass, each run is read into memory and merged with another run, then written out to disk. Since the number of runs is halved with every pass, this requires $\lceil \log_2 N \rceil$ passes. Since an additional initial pass is required and each pass requires $2N$ I/Os, the total cost is: $2\mathsf{N}(\lceil \log_2 \mathsf{N} \rceil + 1)$
  - Thus, we can see that the number of passes we need to make is critical to the overall performance of the sort (since in each pass, we read and write the entire file). Furthermore, the number of runs that we start with also determines the number of passes that are required.
  - This algorithm only requires 3 buffers: 2 input buffers (for the merging passes) and 1 output buffer.
- *Generalized Merge Sort:*

- We usually have (far) more than 3 buffers available to use for sorting. If we have B buffers available, we can use them to reduce the I/O required for sorting.
- In the initial pass, rather than producing N runs of 1 page each, we can make larger runs by reading B pages into memory, sorting all of these, and writing them out as a single run. That means that we create $\lceil \frac{N}{B} \rceil$ sorted runs of B pages each.
- In the merge passes, rather than merging two runs at a time, we can merge $B-1$ runs simultaneously: by reading the first page of each sorted run into a buffer and then picking the smallest of these values, writing it to the output buffer, and then repeating the process. This means that merging takes $\log_{B-1}\lceil \frac{N}{B} \rceil$ passes.
- Since there is 1 additional initial pass and each pass requires $2N$ I/Os, the total cost of the merge sort is: $\mathsf{2N} * (\mathsf{1} + \log_{\mathsf{B-1}}\lceil \frac{\mathsf{N}}{\mathsf{B}} \rceil)$

- *Variations On Merge Sort:*

  replacement sort: If the initial pass can produce longer runs, this may be good, even if it results in using a sorting algorithm that is not efficient (i.e. we are willing to trade increased CPU time for fewer I/Os). A "replacement sort" achieves this: slower sort, longer initial runs (twice the length of the runs produced by a traditional sorting algorithm, on average).

  double buffering: Since the dominant factor that determines the performance of an external sort is the number of disk I/Os that need to be performed, we can trade some buffer space to improve performance by prefetching data into some buffers before it is actually required (and predicting what data is needed next is usually pretty easy). This means that the main sort process doesn't need to block waiting for I/O. The trade off is that there are fewer buffers available for doing the sort process itself.

  blocked I/O: Another way to reduce I/O costs is to increase the size of the sequential blocks that are read from disk during the sort. Therefore, rather than reading in only the first page of each run in the merge phase, we can read in the first k pages. The downside is that we have $\frac{1}{k}$ times fewer buffers to use for the merge process, so it may take more passes. But since a practical DBMS can devote enough buffers to an external sort that almost any reasonably sized file can be sorted in 2 passes, this is an acceptable trade off.

- *Sorting Using A B+-Tree Index:*
  - If a B+-tree index is available on the attribute we want to sort by, we can use a completely different algorithm to produce a sorted output relation: just read the tuples in order off the leaf-level of the B+-tree index.
  - If the index is clustered, this can be very fast: the heap is already basically sorted, so all we really need to do is read it in.
  - If the index is unclustered, this actually turns out to be a pretty bad strategy: we may end up doing up to 1 random I/O for each tuple in the result set (not including the time it takes to read the leaf pages themselves). Due to the high cost of random I/O, it may well be faster to do an external merge sort.

Selection Operator

- The basic choice here is between using a sequential scan of the heap relation, and making use of one or more indexes that may be present on the relation.
  - Naturally, an index is only worth considering if the index key includes some portion of the selection predicate.
  - The attractiveness of using an index increases as the size of the relation increases, and decreases as the size of the result set increases.
  - In other words, indexes are at their most useful when processing selections on large relations that return only a small set of matching tuples.
- When executing an index scan on an unclustered index, we can make use of an interesting refinement: scan through the index, keeping track of the RIDs of matching index entries. Then sort this list of RIDs in heap order. Then iterate through the list, fetching the matching tuples.
  - This ensures that any given page in the heap relation is touched either once or not at all. It still isn't as good as clustering (since the number of pages containing one or more matching tuples will still be higher than with clustering, and the matching pages will be distributed over the disk), but it is still an improvement.
  - If the number of matching tuples is very large (so that holding the list in memory as the index scan proceeds would be prohibitively expensive, and/or we would need to do an external sort), this method is inapplicable.
    - In this situation, we can apply this method iteratively: whenever we run out of buffer space for the RID list, we can go ahead and fetch the corresponding heap tuples and clear the list.
- There are two general approaches to evaluating a complex selection predicate:
  1. *Most Selective Access Path*
     - Once we have fetched a heap tuple from disk, it is cheap to apply the remaining selection predicates to it. So the primary thing we want to optimize is reducing the number of tuples we need to fetch from disk in the first place.
     - In order to do this, we should apply the *most selective* predicate first – the predicate that we predict will produce the smallest intermediate relation.
     - As we retrieve the tuples that match this restriction, we can apply the rest of the selection predicates on-the-fly.
  2. *Intersection of RIDs*
     - If there are multiple indexes on the relation that match (distinct) parts of the selection predicate, we can make use of all of them.
     - For each applicable index, compute the set of RIDs that match that particular part of the selection predicate.
     - We then take the *intersection* of all these sets.
     - We then fetch the resulting set of tuples and apply any remaining predicates.

Projection Operator

- The projection operator must do two things: remove unwanted fields, and remove duplicate tuples. Typically (in SQL systems), duplicate removal is only done if it is explicitly requested via `SELECT DISTINCT`.
- *Sort-Based Approach:*
  - We can do duplicate elimination and field elimination by making a few (trivial) changes to the external sort algorithm. These changes lead to better performance, as well!
  - When creating the initial sorted runs (Pass 0), we can immediately remove unwanted fields. This reduces the size of each individual tuple, allowing us to create longer runs.
  - When merging runs, we can trivially remove duplicates: they will appear next to one another. Once again, this makes the output of successive sort passes smaller and smaller as more duplicates are removed.
  - Cost: $\mathsf{P(R)} + 2 * \mathsf{P(T)}$, where $P(T)$ is the number of pages consumed by the sort runs (after the elimination of unwanted fields) and $P(R)$ is the number of pages in the relation.
    - The algorithm reads in all $P(R)$ pages of the relation. Then, it writes out and reads back in all the sorted runs, which we can process on-the-fly to produce the result relation. There are $P(T)$ pages of sorted runs.
- *Hash-Based Approach:*
  - First, we create $B-1$ buffers to hold the output partitions. We assume the existence of a hash function h that distributes tuples among the partitions. We read in each tuple of R (using the free page in the pool to hold the current input page), remove unwanted fields and hash the remaining fields using h to decide which partition to put the tuple into.
    - By definition, duplicate tuples will be placed in the same hash partition. So to find the set of duplicates for some tuple t, we need only consider the other tuples in the same partition as it.
  - We assume the existence of a hash function $h' \neq h$. For each partition p, we hash each tuple of p using h' and insert it into a hash table. If a duplicate for this tuple has already been inserted into the table, discard it.
    - If p does not fit into memory, we can apply this procedure recursively to detect duplicates within the partition, using a hash function h''.
  - Cost: $\mathsf{P(R)} + 2 * \mathsf{P(T)}$, where $P(T)$ is the number of pages of hash partitions we need to create.
    - $P(T) \leq P(R)$ because T does not contain any unwanted fields (although it *does* contain duplicate tuples).
    - We arrive at this formula because this algorithm reads in the relation, writes out $P(T)$ pages of partitions, and then reads each of those partitions back into memory in turn.
- Unless there is a clear reason not to use it, the sort-based approach is the standard technique: it is less sensitive to data skew, and produces a sorted result relation.

- We can use indexes to assist in the evaluation of a projection operator in two ways:
  1. If an index contains all the desired attributes of the relation, we can apply one of the projection algorithms to the index rather than the heap relation. This leads to better performance, because index entries are smaller than heap tuples (and an index is almost always smaller than the heap relation).
  2. If the index is an ordered-tree (e.g. a B+-tree) and it contains all the desired attributes as a *prefix* of the index key, we can optimize this further: we can walk through the index in order, discarding unwanted fields, and removing duplicates by comparing a tuple with its adjacent tuples in the index.
- Duplicate removal is also required to implement `UNION`, so similar algorithms are used.

Join Operator

- In theory, of course, we can evaluate any join query by materializing the cartesian product of the two relations, and then applying the join condition. However, this is typically extremely expensive to do for even small relations, so a practical system avoids doing this whenever possible.
- For the sake of simplicity we will assume that the join query that needs to be evaluated is an equality join on a single column between two relations. Other kinds of join queries with more complex join conditions are possible, of course, and can restrict the choice of join algorithm.
- Let R be the outer join relation and let S be the inner join relation. Let $P(N)$ and $F(N)$ give the number of pages and the number of tuples per page, respectively, for some relation N.

1 Simple Nested Loops Join

- For each tuple in the outer relation R, we scan the *entire* inner relation S:

```
foreach tuple r in R do
  foreach tuple s in S do
    if r.id == s.id then add <r,s> to the result
```

- Cost: $\mathsf{P(R)} + [\mathsf{F(R)} * \mathsf{P(R)} * \mathsf{P(S)}]$

2 Page Oriented Nested Loops Join

- For each page in the outer relation R, we scan the entire inner relation S, and add any tuples that match the join condition to the output relation.
  - This is somewhat more CPU-intensive than simple nested loops (we might need to construct a hash table for the page of the outer relation), but it requires much less disk I/O.
- Cost: $\mathsf{P(R)} + [\mathsf{P(R)} * \mathsf{P(S)}]$

- As we can see, this is an incremental improvement over simple nested loops: for every possible R and S, page-oriented nested loops is more efficient.
- For both simple nested loops and page-oriented nested loops, we *always* want to make the smaller join relation the outer join operand. From the cost formulae for these algorithms, we can easily see that this will always result in better performance.

3 Index Nested Loops Join

- For each tuple in the outer relation R, we probe the index on S's join column and add any matching tuples to the output relation.
- Cost: $\mathsf{P(R)} + [\mathsf{F(R)} * \mathsf{P(R)} * \mathsf{k}]$, where k is the cost of probing the index.
  - The average cost of probing a hash index to find the first matching index entry is 1.2. The average cost for a B+-tree index is 2–4.
  - If the index entry doesn't include the full tuple (which is the usual case), we also need to fetch the heap tuple. This requires an additional I/O.
  - If the index is clustered, we usually don't need to do any more I/Os; if the index is unclustered, we need to perform up to 1 additional I/O for each matching S tuple.

4 Block Nested Loops Join

- This algorithm takes the general approach used by page-oriented nested loops (scan as much of the outer relation as possible, so as to reduce the number of times the inner relation needs to be scanned), and extends it one step further.
- We divide the buffer pool into 3 areas:
  1. One page to hold the current page of the inner join relation.
  2. One page to hold the current page of the output relation.
  3. The remaining pages in the pool, which are used to hold a "block" of the outer relation.
- If there are B pages in the buffer pool, that means that we can use $B - 2$ pages to hold the current "block" of the outer relation.
- The algorithm is simply: for each block of $B - 2$ pages in the outer relation, scan through the inner relation, and add each tuple that matches the join condition to the output relation.
  - This assumes that it is cheap to tell whether a tuple from the inner relation matches *any* of the tuples in the current block of the outer relation. This seems a reasonable assumption: we can use a hash table to do this search with constant-time complexity. Note that we don't account for the memory consumed by this hash table in the buffer space calculations above.
- Cost: $\mathsf{P(R)} + \mathsf{P(S)} * \lceil \frac{\mathsf{P(R)}}{\mathsf{B-2}} \rceil$, where B is the number of pages in the buffer pool.

5 Sort-Merge Join

- If the input relations to the join are sorted on the join attribute, then performing the join is much easier: for any given tuple of R, we have a much reduced set of S that we need to scan through to produce the resulting join tuples.
- So in the sort-merge join algorithm, we first sort the two input relations (if necessary), and then "merge" the two sorted streams.
  - Let r represent the current outer tuple and s the current inner tuple.
    1. If $r > s$, then advance the scan of the inner relation.
    2. If $r < s$, then advance the scan of the outer relation.
    3. Else if $r == s$, then we say that we have found a "partition" of R and S. For each tuple in the R partition (i.e. for each R tuple equal to r), for each tuple in the S partition, we add a tuple to the result relation containing (r, s).
       * We assume that we have enough buffer space to hold the current partitions of both R and S at the same time. If the join produces a large number of tuples, this assumption may not hold. When one of the relations has no duplicates in the join column (e.g. in a foreign-key join), we can guarantee that this does not happen.
- As a refinement, we can skip the last phase of the external sort, and effectively merge together the runs of R and S to form the join result. This requires more buffer pages, however: we need one buffer page for every run of R and S. If L is the size of the largest relation, this method is applicable when $B \geq 2\sqrt{L}$
- Cost: $3 \cdot [\mathsf{P(R)} + \mathsf{P(S)}]$
  - This assumes that neither input relation is sorted, and that we are making use of the sort-merge refinement mentioned above. In that case, the algorithm effectively does the following:
    1. Read in the outer join relation and write it out as a set of sorted runs. This requires $2P(R)$ I/Os.
    2. Read in the inner join relation and write it out as a set of sorted runs. This requires $2P(S)$ I/Os.
    3. Read in both sets of sorted runs (one page at a time), and produce the result relation. This requires $P(R) + P(S)$ I/Os, not including the cost for producing the output (which we don't need to consider).
    4. Therefore, the total cost is $2P(R) + 2P(S) + P(R) + P(S) = 3 \cdot [P(R) + P(S)]$
  - The merge phase requires $P(R) + P(S)$ I/Os, provided that we always have enough buffer space to hold both partitions in memory at the same time. If the join produces a large result set (a given tuple in one relation joins with many tuples in the other relation), this assumption may not hold.
  - We don't have to pay the cost to sort one of the input relations if it is sent to us already in sorted order (which might happen if the relation is clustered, or if an operator lower in the query tree has already sorted it). So a query optimizer should be careful to notice this possibility when it is present.

    - When we don't need to sort one of the input relations, the cost of a merge join is $3P(R_1) + P'(R_2)$, where R_1 is the unsorted relation, R_2 is the sorted relation, and $P'(N)$ gives the number of I/Os required to read the "sorted" version of the relation N. For example, if we obtain the sorted input relation via an in-order traversal of a B+-tree, the cost to obtain the sorted input would be the number of I/Os required to traverse the leaf-level of the tree.

- Merge-join is less sensitive to data skew than hash join, which is discussed below.

6 Hash Join

- *Partition Phase*: Create $B-1$ output buffers, where B is the number of pages in the buffer pool. We assume the existence of a hash function h that distributes tuples over the $B-1$ buffers, based upon the value of the tuple's join attribute. Read in both relations (separately), hash them, and write out the resulting hash partitions. Note that the tuples for the different join relations are hashed and partitioned separately.
  - Since we used the same hash function h for both relations, we know that a tuple t in outer join partition i can only join with the tuples in inner join partition i.
  - If h does not uniformly distribute the input tuples over the hash partitions, we will see poor performance: as a result, one of the partitions may be so large that it will not fit in memory. So picking a good hash function h is important.
  - Since we want to reduce the size of the partitions as much as possible, we therefore want to increase the number of partitions as far as possible. That is why we use as many pages as possible $(B-1)$ to hold the output buffers for the partitions.
  - Cost: $2 \cdot [\mathsf{P(R)} + \mathsf{P(S)}]$
    - We need to read in each relation and write it out again.
- *Probe Phase*: We assume the existence of a hash function $h' \neq h$. For each outer partition produced by the previous phase, read in the partition and hash each tuple with h'. Then, for each tuple in the corresponding inner partition, read the tuple in, hash it with h' and add the resulting join tuples to the output relation.
  - This phase is highly memory sensitive: if one of the outer relation partitions cannot fit into memory, the probing phase will be very expensive since it will require swapping pages into and out of main memory.
    - If a partition r of R is too large too fit into memory, we can apply the hash join algorithm recursively to join r with S.
    - Note that the algorithm is far less sensitive to the sizes of the inner relation partitions.
  - How much memory is required?
    - *Assume:* Each relation is divided into $B-1$ partitions; each partition is of uniform size (i.e. h is perfect); the largest partition we can hold in memory has $B-2$ pages; the hash table we use for the tuples in the partition of R does not require any additional memory.
    - Therefore, if $\frac{P(R)}{B-1} < B-2$ then we have enough memory. Therefore, $B > \sqrt{P(R)}$.

  - Cost: $\mathsf{P(R)} + \mathsf{P(S)}$

- Cost: $3 \cdot [\mathsf{P(R)} + \mathsf{P(S)}]$
- One interesting property of the hash join algorithm is how easy it is to parallelize: once each join relation has been divided into k partitions, we can divide the probe phase among k different machines or processors without requiring any additional inter-machine communication (other than to produce the final result relation).

Query Optimization

- The task of the query optimizer is to accept as input a tree of relational algebra operators (or, equivalently, the parse tree for a query language statement), and to produce a query execution plan. This plan specifies exactly which operations should be performed, in which order.
  - There are a large number of query execution plans that are equivalent (in terms of their result set) to a typical relational algebra expression. So one of the tasks of the query optimizer is to search among this solution space of equivalent query plans to choose the "best" one.
    - The solution space is typically so large that an exhaustive search is impossible, so various heuristic techniques need to be utilized. In practise, the goal of a query optimizer is not to find the globally optimal query plan, but merely to avoid an extremely bad plan.
  - Given a particular query plan, the second task of the query optimizer is to estimate the expense of executing that query plan (largely in terms of disk I/O operations).
    - This is difficult to do precisely: in practise, much of this process comes down to a series of educated guesses and estimates.
  - IBM's System-R query optimizer, described below, introduced many foundational ideas to this field when it was first introduced in the 1970s. As a result, it is still used as the model for query optimizers today. System-R's query optimizer uses a dynamic programming algorithm along with some heuristics to reduce the size of the search space, and works well for queries with fewer than approximately 10 join operators.
  - Strictly speaking, a query optimizer does not work on the entire query language statement submitted by the user; instead, the query is broken down into "query blocks" that are independently optimized. A single query block consists of selections, projections, joins, and aggregation (`GROUP BY`, `HAVING`, etc.); any subqueries that the query references are treated separately, as described below.

- We can classify query plans according to their use of pipelining:

  left-deep: A query plan in which each join operator uses a database relation as the inner join relation, and, where possible, uses an intermediate result relation as the outer join relation.

  right-deep: A query plan in which each join operator uses a database relation as the outer join relation, and, where possible, uses an intermediate result relation as the inner join relation.

bushy: A query plan in which one or more join operators use two intermediate result relations for both join relations.

- Generally speaking, left-deep query plans tend to be more efficient than right-deep or bushy plans.
  - This is because a left-deep query plan can make effective use of pipelining. Join algorithms treat their input relations differently: in some algorithms (such as nested loops), the outer relation is scanned once sequentially, whereas the inner relation is repeatedly scanned. This means that we can make much more progress given a single tuple of the outer relation than the inner relation.
  - When a node in a query plan uses pipelining, it means that it produces its output one relation at a time, which is then immediately consumed by its parent in the query tree. As a result, there is no need to materialize the intermediate result relation (which is an expensive operation).
    - Note that not all query plan operators can effectively take advantage of pipelining (for example, hash join).

Relational Algebra Equivalences

- Selections are commutative: they can be applied in any order, so we are free to apply the most highly-selective predicate first. Also, selections allow for cascading: we can rewrite a predicate of n terms of the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ into n distinct predicates, one for each term.
- Projections can also be cascaded: if we apply n projection operations after one another, the result is equivalent to applying only the last projection.
- Joins are commutative: the order of their operands can be changed without affecting the result. Joins are also associative: given three relations, joining them in any order will always produce the same final result relation.
  - In combination, these two equivalences can produce a huge number of equivalent relational algebra expressions for a query involving a significant number of joins.
- There are some other equivalences involving multiple operators, such as the ability to produce a join from a cross-product and a selection, or to "push down" a selection, but these are all fairly logical.

Cost Estimation

- There are two components to estimating the cost of evaluating a particular operator in a query plan: the cost function of the operator itself, and the number of tuples to which the operator is applied.
  - We know the number of tuples in each base relation. Therefore, we can estimate the sizes of the input and output relations for each node in the query plan in a "bottom up" fashion, starting at the leaf nodes of the tree.

  - Given these input and output sizes, along with other simple factors like whether one of the input relations is already sorted in a particular order, it is usually trivial to apply the operator's cost formula to derive a total cost, in terms of some metric such as the number of disk I/O operations — so we will not focus on this part of the problem any more.

- The primary factor that affects the number of tuples being manipulated are `WHERE` clause predicates (both regular selections and join conditions). For each predicate, we want to estimate a **reduction factor**, which is the portion of the input for which the selection predicate holds.
  - We estimate the size of the result relation as the size of the input relations multiplied by the product of the reduction factors of the selection predicates. *This assumes that the predicates in the query are independent.* While this assumption is of course not true, it is usually made for the sake of simplification.
- For the predicate `col = value`, the reduction factor is $RF = \frac{1}{NKeys(I)}$, where I is an index on this attribute.
- For the predicate `col > value`, the reduction factor is $RF = \frac{Max(I)-value}{Max(I)-Min(I)}$, where I is an index on this attribute.
- Any practical DBMS will make use of statistics (such as histograms) to estimate reduction factors more accurately.

The System-R Algorithm

- A fundamental design decision made by System-R is to *only* consider left-deep query plans. This is a heuristic technique to reduce the size of the solution space, however: the globally optimal plan may not be left-deep.
- Query optimization for a query block that doesn't include any joins is fairly simple:
  - Regardless of which method is used to access the actual data, we generally want to apply the most selective predicate (the one with the highest reduction factor first).
  - If an index is available, the optimizer considers using it. In addition to a simple index scan for a predicate involved in the query block, we can also use index-only scans, multiple index scans followed by an intersection of RIDs, and fetching a sorted stream of tuples from a tree index.
  - In addition, there is always the option of using a simple sequential scan.
  - Typically, aggregate operations are performed last. Where possible, multiple operations will be done simultaneously (for example, we can do projection and on-the-fly selection at the same time).
  - The query optimizer enumerates each of these possibilities, estimates their cost, and chooses the one with the least cost.
- For query blocks involving joins, System-R uses a dynamic programming algorithm to search through the space of left-deep plans in a series of passes:

1. Enumerate all the single relation query plans for each relation in the query. This involves examining the available indexes, and selecting the elements of the selection predicate that only apply to this relation. For each such query plan, we note the plan's total cost, the number of result relations it produces, and whether the result relation is produced in sorted order. We prune the uninteresting plans from this set (see below), and continue.
2. For each single-relation plan P produced by Pass 1, we iterate through every other single-relation plan P', and consider the plan in which P is the outer join relation and P' is the inner join relation. For each pair (P, P'), we consider the best way to access P'. This involves considering any predicates that apply only to P' (so they can be applied before the join), predicates that apply to both joined relations (so that they can be used as the join condition), and predicates that involve other attributes and can be applied after the join.
3. We now consider three-relation plans. For each interesting two-relation query plan produced by the previous pass, we consider that query plan as the outer join relation, and each of the single-relation query plans produced by Pass 1 as the inner relation. Naturally, we don't consider the relations already joined as candidates for being the inner join relation. We prune uninteresting query plans and continue to the next pass.
4. ...

This process continues until we have constructed a query plan with a sufficient number of joins. Then we pick the cheapest query plan as the plan to be used for the entire query.

- At the end of each pass, we only retain a subset of the plans we generated. We keep the cheapest plan without an interesting sort order for each set of relations. We also keep the cheapest plan with a particular interesting sort order for each set of relations.
- Aggregate operations (including `GROUP BY`, `HAVING`, and aggregate functions) are handled at the very end of this process, once the join order has been determined.
- We handle nested queries by essentially treating them as subroutine invocations: for each tuple in the outer query, we need to reevaluate the nested query.
  - There are some situations in which the nested query need only be evaluated once. If the nested query does not refer to any construct defined by an outer join (in other words, if it is *uncorrelated*), we can merely compute the nested query's relation once.
  - Therefore, where possible, the query optimizer will effectively "push up" an uncorrelated nested query to become a join in the parent query.
    - Rather than consider the full range of join methods for this new temporary relation, practical optimizers will typically only use some variant of nested loops. This is to reduce the difficulty of implementation, since nested loops are also used for correlated nested queries as described below.
  - When the nested query references some component of the parent query, this simple technique is not possible: we need to reevaluate the nested query for each tuple in the parent query.

    - In effect, this is a simple nested loops algorithm: for each tuple in the outer relation (the parent query), we need to fetch the entire inner relation (which means, in this case, produce the nested query's result set). Obviously, this is inefficient in many circumstances.
    - Furthermore, the nature of the nested query imposes some additional constraints on the optimizer's ability to rewrite the query: the parent query is always the outer join relation, and the nested query is always the inner join relation. No other possibilities are considered.
  - Many nested queries, including correlated nested queries, can be expressed equivalently without requiring nesting. Unfortunately, this often cannot be done by the query optimizer automatically, so it needs to be done by an educated database user.
  - In general, the ad-hoc manner in which the System-R algorithm treats nested queries leads to poor performance for these types of queries.

Concurrency Control

Basic Definitions

- A database management system executes **transactions** that consist of database actions. A transaction can perform **reads** and **writes** of **database objects**. Furthermore, a transaction's final action must be either a **commit** (in which case the changes made by the transaction are permanently recorded), or an **abort** (in which case the transaction's changes should be undone). For performance reasons, we want to allow the system to execute multiple transactions concurrently.
  - The database system guarantees that every transaction will be **atomic**: either all of the changes made by the transaction will be stored permanently, or none of them will.
  - We will leave the definition of a "database object" deliberately vague for the moment: the granularity of the objects being locked is discussed later.
  - The possibility of a transaction aborting means that "cascading aborts" could occur: if an uncommitted transaction writes a value that is read by another transaction before it aborts, the second transaction also needs to be aborted, because it is operating upon invalid data. Any real-world system needs to devise some form of concurrency control that prevents cascading aborts.
- So we need to define the manner in which concurrent transactions are allowed to access shared resources. The kinds of concurrent behavior that we allow should have no direct effect upon the user, other than by influencing database performance.
  - In practise, some concurrency control schemes will need to occasionally abort and restart user transactions in order to avoid problematic situations.
- So that it is easy for users to understand the behavior of the system, we want to make the interleaving of concurrent transactions correspond to some serial schedule of the database actions.

Schedules

- When we want to enumerate the work performed by a set of concurrently-executing transactions, we can use a **schedule** to list the transactions and the order in which they perform operations. A **serial schedule** is a schedule in which no transactions are interleaved: one transaction runs from beginning to end, followed by another. Two schedules are **equivalent** if the execution of one schedule leaves the database in the same state as the execution of the other schedule. A schedule is **serializable** iff it has the same effect on the state of the database as some serial schedule.
  - Whenever possible, we want to restrict the interleaving of database actions so that only serializable schedules are possible. This is because it is easier for the user to understand the effect of serializable schedules.
  - Unfortunately, we need a more useful definition of "serializability": it is difficult to algorithmically assess whether a given schedule satisfies the above definition of serializability.
  - Therefore, we define the notion of **conflict equivalence**: two schedules are conflict equivalent iff they perform the same actions on the same database objects, and *every pair of conflicting operations is ordered in the same way.*
    - To elaborate, a "conflicting operation" occurs when two transactions write to the same object or one transaction writes an object and another transaction reads that object (either the read or the write can occur first). Conflict equivalence basically states that if T_i precedes T_j when performing a potentially conflicting action, T_i always precedes T_j for all potentially conflicting actions involving these two transactions.
  - We call a schedule **conflict serializable** iff it is conflict equivalent to some serial schedule.
  - For example, the following schedule is neither serializable nor conflict serializable:

T_1	T_2
R(A), W(A)	
	R(A), W(A)
	R(B), W(B)
R(B), W(B)	

    It is not serializable, because there is no serial schedule to which it is equivalent: not T_1, T_2, and not T_2, T_1. It is not conflict serializable because there is no serial schedule to which it is conflict equivalent: T_1 modifies A before T_2 does, but T_2 modifies B before T_2 does, so the order of these conflicting operations is not consistent.

- To determine if a given schedule is conflict serializable, we can use a **precedence graph**.
  - There is a node in the graph for every active transaction.
  - We add an edge to the graph from T_i to T_j whenever T_j performs an operation that conflicts with a previous operation performed by T_i. In other words, an edge from T_i to T_j indicates that T_i must precede T_j in order to preserve consistency.
  - A schedule is conflict serializable iff the schedule's precedence graph is acyclic.

- However, there are serializable schedules that are not conflict serializable: conflict serializability is a "sufficient but not necessary" condition for serializability. An alternative definition of serializability is called **view serializability**.

  - Two schedules S_1 and S_2 are called **view equivalent** iff:
    * If a transaction $T_i \in S_1$ reads the initial value of A, then $T_i \in S_2$ must also read the initial value of A.
    * If a transaction $T_i \in S_1$ reads the value of A written by $T_j \in S_1$, then $T_i \in S_2$ must also read the value of A written by $T_j \in S_2$.
    * If a transaction $T_i \in S_1$ writes the final value of A, then $T_i \in S_2$ must also write the final value of A.
  - A schedule S is called **view serializable** iff it is view equivalent to some serial schedule.
  - Any conflict serializable schedule is view serializable, but not all view serializable schedules are conflict serializable (the view serializability condition is weaker, therefore, than the conflict serializability condition).
  - For example, the following two schedules are *not* conflict equivalent, but they are view equivalent:

T_1	T_2	T_3
R(A)		
	W(A)	
W(A)		
		W(A)

T_1	T_2	T_3
R(A)		
W(A)		
	W(A)	
		W(A)

  - Furthermore, the first schedule is *not* conflict serializable, but it is view serializable.
  - In general, conflict serializability is unnecessarily restrictive about "blind writes" (writes made without a preceding read). View serializability handles blind writes more effectively.

- Every practical database management system uses conflict serializability, not view serializability, because it is significantly easier to implement.

- A schedule is **recoverable** if, for every transaction T in the schedule, T only commits once every transaction T has read data from have committed. The point here is that if T commits before a transaction that has written some data it has read has committed, we have a problem: if the writing transaction later aborts, T will need to be aborted as well because it has read data it should not have been able to read.
  - We say that a schedule **avoids cascading aborts** if the schedule consists entirely of transactions who only read values written by previously committed transactions.[1]

- We say that a schedule is **strict** if whenever any transaction T in the schedule writes a value, that value is not read or overwritten by another transaction until T has committed or aborted.

Two-Phase Locking

- To ensure that the system executes user transactions in a serializable fashion, we can limit the ways in which transactions can be interleaved. One such limitation is called **Strict Two Phase Locking** (Strict 2PL).

[1] As an exception, we should probably allow a transaction to read a value it has itself previously written by it has committed.

- In Strict Two Phase Locking, a transaction must obtain a **shared lock** on an object before reading it, and must obtain an **exclusive lock** on an object before modifying it. When a transaction acquires a lock, it is *never* released until the end of that transaction (at which point all locks are simultaneously dropped).
- This scheme *only* allows for serializable schedules. However, it also results in poor concurrency.
- As with any concurrency control system that utilizes locks, deadlocks are a possibility, and must be dealt with.
- To allow us to generalize this concurrency control scheme, we can divide it into two phases: the **Growing Phase**, in which a transaction is still acquiring new locks, and the **Shrinking Phase**, in which a transaction is releasing its locks. In Strict 2PL, the shrinking phase must come at commit- or abort-time.
  - If we loosen this restriction and allow the Shrinking Phase to begin at any point, the resulting scheme is called **Two Phase Locking** (2PL).
  - Note that in 2PL, although we allow the Shrinking Phase to start earlier, we still cannot allow the Shrinking and the Growing Phrase to overlap one another: once a transaction has begun releasing locks, it cannot acquire any new ones.
  - Like Strict 2PL, this scheme only allows serializable schedules.

Deadlocks

- A deadlock occurs when two transactions are waiting for each other to terminate in order to be able to acquire some shared resource. In other words, if we construct a graph in which the nodes are transactions and an edge from T_1 to T_2 indicates that T_1 is waiting for a resource held by T_2, then a deadlock has occurred iff there is a cycle in the graph.
- Deadlocks are possible in any database system that uses locks for concurrency control.
- There are two main approaches to dealing with deadlocks:

  1. **Deadlock Prevention:** the locking protocol disallows the possibility of a deadlock occurring, at the expense of aborting a few more transactions than strictly necessary.
     - The general idea is that we want to prevent potentially dangerous situations from arising. Furthermore, when we need to choose between two transactions, we always prefer the **older** one, on the theory that it has done more work, and is therefore more difficult to abort.
     - We assign priorities to each transaction. The older a transaction is, the higher priority it has.
     - When T_i attempts to acquire a lock held by T_j, two algorithms are possible:

       Wait-Die: If T_i has higher priority, T_i waits for T_j; otherwise, T_i aborts.
       - In this scheme, once a transaction has successfully acquired a lock, it knows that it has acquired the lock permanently: it will never be preempted or aborted by a higher priority transaction.

       Wound-Wait: If T_i has higher priority, T_j is aborted; otherwise, T_i waits for T_j.

        * This is a preemptive scheme: lower priority transactions are forcibly killed and aborted when they interfere with the progress of a higher priority transaction.
    - *When an aborted transaction is restarted, it has the same priority it had originally.* In other words, once a transaction has been aborted, we give it preference in the future.
    - Regardless of which deadlock prevention scheme is used, however, the end result is the same: some transactions are needlessly aborted. These algorithms are too conservative: if there is even a possibility of a deadlock occurring, a transaction is aborted.
2. **Deadlock Detection:** the locking protocol allows the occurrence of deadlocks, and then detects and resolves them when they happen.
    - This approach is more optimistic: the insight is that since deadlocks occur relatively rarely, it is a better idea to simply assume that they won't occur, and detect and handle them when they do.
    - To detect deadlocks, we can create a "waits for" graph, which is precisely the same as the graph described earlier: the nodes in the graph are transactions, the edges are the un-granted lock requests, and the algorithm merely walks through the graph, checking for cycles (which indicate that there is a deadlock).
        * Once we have detected a deadlock, how do we decide which transaction to abort? The standard method is to choose the "youngest" transaction by some metric (for example, the transaction with the newest timestamp, or the transaction that holds the least number of locks, etc.)
    - The downside to this approach is that it is relatively expensive to construct this graph and check for cycles. As a result, the deadlock check is only done periodically. Therefore, when a deadlock occurs, there will be a period of time before the deadlock is detected: in this window, the system is not doing very much useful work. As a result, if deadlocks occur frequently, this can result in a loss of throughput.

- In real database management systems, deadlock detection is almost always used: the view is that since deadlocks are a fairly rare event, so the "cure" provided by deadlock prevention is worse than the "disease". Provided the assumption that deadlocks are rare holds true, this is usually a pretty effective solution to the deadlock problem.

Locking Granularity

- We haven't considered how granular the locks we are acquiring are. Should we lock entire databases, relations, pages, tuples, or individual attributes within a tuple?
    - There is a trade-off here: the finer the level of granularity, the higher the degree of concurrency allowed (since two locks are less likely to conflict). However, it also means that there are more locks that need to be managed. Furthermore, since each lock acquisition and release has some finite overhead, this overhead increases as the granularity of the locks increase.
    - We also need to consider the "nested" nature of the containers in which we can store data: a page contains tuples, and a relation contains pages, and so on. For example,

what behavior do we expect to happen if a transaction tries to acquire an exclusive lock on a relation in which another transaction has already acquired a lock on an individual page or tuple?

- Rather than picking a single granularity level at which to acquire locks, we can instead use a system of "intentional" locks. This requires inventing some new locking modes: "intentional shared" and "intentional exclusive".
  - For the sake of convenience, we can also introduce a new lock mode called "shared-intentional exclusive" for the common situation in which a transaction is reading through the content of an object, intermittently updating some elements within it.
  - A shared-intentional exclusive lock is like a combination of a shared lock and an intentional exclusive lock: if conflicts with anything that would conflict with either of these locks.
  - The new logic for which locks conflict with one another is as follows:

	—	IS	IX	S	X
—	x	x	x	x	x
IS	x	x	x	x	
IX	x	x	x		
S	x	x		x	
X	x				

    where `x` indicates that the two lock modes do not conflict with one another.
  - When acquiring a lock on a particular object, we need to acquire intentional locks on all the parents of this object in the storage hierarchy. For example, before we can acquire an exclusive lock on a tuple t, we must first acquire intentional-exclusive locks on the database, relation, and page in which the tuple is located.
    - In order to be able to acquire a shared lock on a node, we must hold an intentional shared, intentional exclusive, or shared-intentional exclusive lock on the node's parent.
    - Similarly, we must hold an intentional exclusive or shared-intentional exclusive lock on the parent node before we can acquire an exclusive lock on the node.
  - We follow the opposite procedure when releasing a lock that we have acquired: we release the most specific lock first. We start at the bottom of the object hierarchy, releasing locks and working upward.

Locking for Dynamic Databases

- We have earlier stated that Strict 2PL guarantees that it will only allow serializable schedules. However, this claim does not hold if we allow new database objects to be created, as we will see; in this situation, additional steps must be taken to avoid undesirable behavior.
- This is because when a transaction attempts to lock a set of objects matching some predicate, only the objects that match the predicate at a given point in time will actually be matched. If a new object that happens to match the predicate is added to the database while the transaction still holds its lock, an inconsistent situation has arisen.

- A complete solution to this problem is **predicate locking**.
  - This is a locking scheme in which locks are acquired by specifying some arbitrary predicate.
  - When new locks are granted, the new predicate is checked with the predicates of all the outstanding locks on the system. If any of these predicates can match the same tuples, the lock acquisition is blocked until the conflicting lock is released.

  The problem with predicate locking is that it is both difficult to implement, and can have a disastrous impact on performance. So few practical systems implement it, if any.

- An acceptable practical solution to this problem is **index locking**. When a set of objects that match some predicate is locked, the database system also checks to see if there is an index whose key matches the predicate. If such an index exists, the structure of the index should allow us to easily lock all the pages in which new tuples that match the predicate appear – or *will appear* in the future.
  - If another transaction tries to insert a new object that matches the predicate, it will first attempt to acquire a lock on the page in the index where the new object would belong. This will block until the original transaction has released the lock. Therefore, no new objects matching the lock predicate can be inserted until the lock is released.
  - Therefore, index locking is an efficient way to implement some kinds of predicate locking.
  - The practical issue of *how* to locate the relevant pages in the index that need to be locked and what locks need to be acquired is discussed in the following section: only B+-tree indexes are explicitly discussed, however.

Locking for B+-Tree Indexes

- A B+-tree index, like any tree-based index, contains two types of nodes: interior nodes, which are used to "direct" searches to the portion of the tree in which the results can be found, and leaf nodes, which contain the index entries themselves. All B+-tree operations begin at the root node and work their way down through the tree.
  - Naively, we could apply the same locking ideas to B+-trees that we do to normal database operations. However, this does not take advantage of the nature of a B+-tree index: it would effectively serialize modifications to the index, because every transaction that needed to perform an update would acquire an exclusive lock on the root node (and due to Strict 2PL, this lock would not be released until the transaction committed).

- We can take advantage of two properties of B+-tree indexes to allow for better concurrent performance:
  1. Interior nodes do not contain any data: they merely "direct" searches.
  2. When modifying a leaf node, we only need to modify the leaf node itself, *unless* a split is required. Fortunately, splits occur relatively rarely.

- The basic idea behind all tree-locking algorithms is similar: once we have locked a node and fetched and locked the appropriate child node, we can determine if the child node is **safe**. If it is, we know that no modification we can make to the sub-tree rooted at the child node could

affect the parent node, so we can release any locks we have acquired on nodes higher up in the tree.

  - A node is safe for an insert operation if it is not full. A node is safe for a delete operation if it is at least half full.
  - Note that these locking algorithms *violate Strict 2PL*. However, the nature of the B+-tree data structure allow us to make these performance optimizations without effecting correctness: these locking algorithms still guarantee serializability.

- We studied several variations of B+-tree locking in class:

  Algorithm 1: Start at the root node and traverse through the tree as normal. If the operation is a search, we acquire shared locks on child nodes and then we can immediately drop the shared lock we have acquired on the parent node. If the operation is an insert or delete, we acquire exclusive locks on child nodes. If the child node we have just acquired an exclusive lock for is safe, we can release the locks we hold on all ancestor nodes.

  - This algorithm is very simple. Furthermore, even it is better than naively applying Strict 2PL techniques to tree indexes.
  - However, it does not allow for very good concurrency: if someone holds an exclusive lock on a node that is close to the root (or worse yet, the root itself), this blocks a lot of other operations. In other words, this algorithm typically results in a high degree of contention for the locks on the nodes of the upper levels of the tree.

  Algorithm 2: The algorithm works like Algorithm 1, with the exception that insert and delete operations acquire shared locks on interior nodes (rather than exclusive locks). When an exclusive lock is acquired on the appropriate leaf node, it is checked: if the leaf node is safe, we proceed with the insertion or deletion. Otherwise, we release all the locks on this index and start again, using Algorithm 1.

  - The improvement here is that we have "optimized for the common case": since we only rarely need to balance the tree, it is more efficient to optimistically assume that no re-balancing is needed (until we can determine for certain whether it is). If this assumption turns out to be correct, we have to start over, but for most usage scenarios this algorithm should offer significantly better concurrent performance than Algorithm 1.

  Algorithm 3: This algorithm works like Algorithm 1, with the exception that insert and delete operations acquire "intensional exclusive" (IX) locks on interior nodes (rather than exclusive locks). As before, when we have arrived at the correct leaf node, we immediately acquire an exclusive lock on it. If the leaf node is unsafe, we start from the locked node that is closest to the root and upgrade all the IX locks we have acquired to exclusive locks.

  - Note that we need only upgrade our locks from intentional exclusive to true exclusive locks for the nodes that we still have acquired locks on when the algorithm reaches the leaf node. In other words, we still release parent nodes as before; once a lock on a parent lock has been released, it never needs to be subsequently "upgraded" to an exclusive lock.

- Doing the lock upgrade operation starting from the top and working down the tree reduces the chance of a deadlock occurring.
  - (In this author's opinion, this algorithm still seems prone to deadlock!)

Algorithm 4 (Hybrid): This algorithm takes advantage of an additional observation: as we move from the root toward the leaf nodes, the chance of needing to acquire an exclusive lock on a particular node increases. However, the impact that acquiring an exclusive lock has on concurrent performance decreases: the deeper into the tree we are, the fewer other operations we interfere with when we hold an exclusive lock on a node.

Therefore, we divide the tree into three horizontal sections. In the first, the closest to the root, we use Algorithm 2's approach: we acquire shared locks on interior nodes, but if a split is required, we drop all our locks and restart the operation using Algorithm 1. In the second section, we use Algorithm 3's approach: we acquire intentional exclusive locks on interior nodes, but if a split is required, we do a top-down upgrade of those locks to exclusive locks. In the last section (at the leaf level, or perhaps slightly above it?), we acquire exclusive locks to begin with.

Buffer Management and Crash Recovery

- Before discussing crash recovery itself, it is worth noting that there are two properties of a buffer manager that significantly affect the task of the DBMS recovery manager:

  steal: A buffer manager is said to use a "steal" approach if it allows pages dirtied (and subsequently unlocked and unpinned) by an uncommitted transaction to be candidates for page replacement. In other words, if the buffer manager uses stealing, a change made by a transaction may reach disk before the transaction has committed.
  - If we disallow stealing, we need enough space in the buffer pool to hold all the pages modified by all the transactions that are active at any one time.
  - If we allow stealing, we need some mechanism to restore the old values of database objects that are modified and written to disk by a subsequently aborted transaction.

  force: A buffer manager is said to use a "force" approach if it forces all the changes made by a transaction to disk when that transaction aborts.
  - If we use a force approach, we suffer a significant performance penalty: if a page is modified 20 times by different transactions in short succession, it would need to be force-written to disk 20 times. Furthermore, it limits the ability of the storage subsystem (either the operating system or the disk subsystem itself) to reorder these writes to optimize performance, which is common practise.
  - If we do not use a force approach, we need to take additional measures to ensure that the changes made by a successfully committed transaction persist through system crashes.

- A buffer manager that uses no-steal and force is the easiest to implement. It also makes concurrency control and crash recovery less complex. Unfortunately, it leads to poor performance.

- For that reason, nearly all practical database systems use a steal, no-force approach to buffer management. We assume that this approach is used throughout the rest of these notes.

Crash Recovery

- Crash recovery implements two important properties of a relational database management system:

  atomicity: We need to ensure that the database only contains changes written by successfully committed transactions. This requires that when a transaction aborts, we have some mechanism of undoing the changes it has made.

  durability: It is possible for a system crash to occur at any time (for example, the power might fail). After the crash, the database must be restored to a consistent state: all the data written by committed transactions must be on-disk, and none of the data written by aborted transactions should be (transactions that were in-progress at the time of the crash are considered aborted).

- Since we assume that the buffer manager uses a steal, no-force approach, the **recovery manager** must be responsible for implementing these properties. The basis for the recovery manager we will be studying is called **Write-Ahead Logging** (WAL). WAL is used in most relational database systems. The particular WAL technique we are studying is called "ARIES", and was invented by IBM.

- The general idea behind "logging" for crash recovery is:

  - The log is a file containing a sequence of records. Each record represents a change in the state of the database system (for example, "T_1 changed object N from x to y"). In addition to the transaction ID and record type indicator (see the enumeration of record types below), each record also contains:

    Log Sequence Number: Called the LSN, this is a monotonically increasing integer that identifies this particular log record (and hence, its position within the sequence of log records).

    prevLSN: The LSN of the previous log record emitted for an action made by this transaction. These fields form a singly-linked list backward through the WAL, allowing all the changes made by a particular transaction to be efficiently retrieved.

  - If we want to "undo" the effect of a database action (for example, as part of aborting a transaction), we can use the log records describing the change to revert the modified database object to its previous state.

  - If we want to guarantee that an update made by a transaction is recorded in stable storage (for example, when that transaction commits), we can force-write the log record for the update in question. If the actual data change itself is lost (due to a system crash before the buffer manager writes the modified page out to disk, for example), the log record contains enough information that it can be used to "redo" the lost change.

  - Of course, we could accomplish the same effect by forcing all the changes made by a transaction to disk when it commits. The advantage to using a log is that it is much faster to force the log to disk than it is to write out all the changes made by the transaction:

    * The transaction may have made a lot of modifications to a lot of different pages. On the other hand, log records are small, so we can typically fit many of them into a

single page. Therefore, force-writing all the log records for almost any transaction only requires a single disk write, rather than one disk write for each modification.

  * Furthermore, the log file is written sequentially to disk, so it is cheaper to do this force-write than it is a variable number of forced-writes at random locations on the disk.
  * A single forced-write writes all records up to and including a particular record to disk. That means that if multiple transactions commit concurrently, we can ensure all their changes are recorded on permanent storage with only a single forced-write.

- The ARIES algorithm also adheres to the following principles:

  Write-Ahead Logging: The algorithm *requires* that the WAL record for a database action be recorded on disk *before* the changes made by the database action itself are recorded on disk.

  Repeating History During Redo: When recovering from a crash, ARIES first replays all the actions in the log to bring the database into the exact state it was in just prior to the crash, and then it aborts any active transactions.

  Logging Changes During Undo: When performing an "undo" operation, we need to change the database state by reverting changes made by aborted transactions. These database changes are themselves recorded in the WAL, so that if the system crashes during the recovery process we can recover from that as well.

- Each page contains an additional field: a `pageLSN`, which contains the LSN of the most recent WAL record that refers to a change made to this page.

- There are several types of WAL records:

  Update: When a transaction changes the state of a relation, an "update" record is written to the WAL. The LSN of this record is marked in the modified page's `pageLSN` field.

  Commit: When a transaction is ready to commit, a "commit" record is force-written to the WAL. As soon as this record has reached stable storage, the transaction is considered to have been successfully committed. After this record has been written, some additional actions need to be taken; once these have been completed, an "end" record is written to the log.

  Abort: A transaction can be aborted for several reasons: the user can explicitly abort the transaction, the system can abort a transaction as a result of a condition such as deadlock, or the crash recovery process can abort in-progress transactions when restoring the state of the database after a system crash. Whatever the cause of the abort, we write an "abort" record to the log to indicate that this transaction has been aborted.

  End: When the subsequent changes implied by a "commit" or "abort" record have been successfully completed, an "end" record is written to the WAL to indicate that the specified action has completed.

  Begin Checkpoint: This indicates that the system has begun to perform a checkpoint; see the section on checkpointing below.

End Checkpoint: This indicates that a checkpoint operation has completed. This record contains a copy of the transaction table and dirty page table that were accurate as of the instant we began to take this checkpoint.

Undo Update: Also known as a **Compensation Log Record** (CLR). When the system wants to abort a transaction, it needs to "undo" the changes made by that transaction to database objects. It does this by discovering the old value of the modified object (for example, by consulting the WAL record for the original modification), and updating the object to this value. Just before this update is made, a CLR record describing the change is written to the WAL. CLR records contain an additional field `undoNextLSN` which is the LSN of the next WAL record whose changes need to be undone. For a given update record U, the `undoNextLSN` of a CLR record C for U is the `prevLSN` of U.

- **Checkpointing** is utilized by the recovery manager to improve the speed of crash recovery and to bound the amount of disk space consumed by the WAL. A checkpoint is executed on a periodic basis by the DBMS.
  - When the checkpoint is begun, a "begin checkpoint" record is written to the WAL.
  - Next, a "end checkpoint" record is constructed in memory. This record contains a copy of the transaction table and the dirty page table as of the moment at which the "begin checkpoint" record was written.
  - Next, the "end checkpoint" record is written to the WAL. Note that additional WAL records may have been written to the log between the beginning and end of the checkpoint: since we only assume that the "end checkpoint" record accurately describes the state of the system at the moment of "begin checkpoint", this is acceptable.
  - Finally, a special "master" log record is updated containing the LSN of the new "begin checkpoint" record.
  - When the system begins crash recovery, it can begin reading the WAL at the "begin checkpoint" record indicated by the master log record. Further, the only log records prior to this "begin checkpoint" record that need to be accessed in the future are those after the earliest `recLSN` in the dirty page table.
    - Therefore, the more quickly dirty pages are flushed to disk, the less WAL information needs to be retained, and the faster crash recovery can be done.

- During normal operation, the state of the system is encoded in two data structures:

  Transaction Table: This table contains one entry for every active (currently executing) transaction. In addition to the transaction ID, we also record the `lastLSN` for each active transaction: this is the LSN of the *last* WAL record related to this transaction.

  Dirty Page Table: This table contains one row for every page in the buffer pool with the *dirty* bit set. In addition to the page ID, we also record the `recLSN` for each dirty page: this is the LSN of the *first* log record that caused this page to become dirty. As a result, this field gives the LSN of the earliest modification that might need to be re-applied during crash recovery.

These tables are incrementally maintained by the system during normal operation. An important step in crash recovery is using the information in the log to restore the state of these data structures to the point in time just before the crash occurred.

- When a crash occurs, ARIES employs the following 3-stage algorithm to restore the system to a consistent state:

  Analysis: The goal of this phase is to restore the state of the transaction table and the dirty page table to the point in time just before the crash occurred.

  1. Read the master record, and obtain the LSN of the "begin checkpoint" record belonging to the most recently completed checkpoint. Locate this record in the WAL.
  2. Initialize the dirty page table and transaction table using the stored copy in the "end checkpoint" record that should closely follow the "begin checkpoint" record.
  3. For each record r in the log after the beginning of the checkpoint:
     - If r is an "end" record for transaction T, remove T from the transaction table because it is no longer active.
     - If r is a non-ending record for a transaction T, add T to the transaction table unless it is already there. Also, update the transaction table to reflect the information in r: update the `lastLSN` field, and note in the transaction's status if it has started to commit.
     - If r is an update record or CLR that refers to a page P, add P to the dirty page table unless it already exists.
  4. We now have up-to-date copies of the transaction table and dirty page table; this is used by the next phase in the crash recovery process.

  Note that the dirty page table we construct during this phase may be more conservative than necessary: pages that were modified by WAL records after the last checkpoint may have been written out before the system crashed, so they may not actually be dirty. But since we can detect and avoid applying an update that has already been applied, this is not a problem.

  - Some systems write an additional WAL record to indicate that a particular dirty page has been flushed out to disk by the buffer manager. This is not done by ARIES: although it would improve the speed of crash recovery, it would slow down routine operation.

  Crash During Recovery: If the system crashes during the analysis phase, no additional steps need to be required: since this phase does not write any additional WAL records and does not modify the database, the system can once again restore the transaction table and dirty page table when the system is restarted after the second crash.

  Redo: In this phase, the system uses the information in the transaction table and the dirty page table, in combination with the WAL, to reapply the effects of *all* the changes performed by the system before the crash occurred.

  - Note that in this phase we also apply the updates made by transactions that were aborted before the crash, and transactions that were still in-progress when the crash occurred (which are considered to be aborted by the system). The changes made by the former are undone when the appropriate "undo update" is read from the WAL;

the changes made by the latter are undone during the subsequent undo phase of crash recovery.

This phase performs the following actions:

1. Find the smallest LSN that is the `recLSN` of an entry in the dirty page table (this identifies the WAL record for the oldest update that *may* not have been written to disk).
2. For each record r with an LSN greater to or equal to the LSN from the previous phase:
   - If r is not an update or CLR record, it can be ignored.
   - If r applies to a page that is not in the dirty page table, it need not be reapplied.
   - If r applies to a page in the dirty page table but the `recLSN` of that page's entry in the table is greater than the LSN of r, r need not be reapplied.
   - If r applies to a page whose `pageLSN` that is greater than or equal to the LSN of r, r need not be reapplied. (Note that this requires fetching the page in question from the heap.)
   - Otherwise, r should be reapplied. Therefore, the page that r applies to is updated as necessary (using the information in r), and the page's `pageLSN` is set to the LSN of r.
     * Note that no WAL record is written to record this action, at this point in time. We also do not force write the new state of the page back to disk.

   Crash During Recovery: If a crash occurs during the "redo" phase, no additional steps need to be taken. That is because we have not written any additional WAL records; therefore, the state of the WAL is identical to the state of the WAL when the crash recovery process was initially invoked. We may have modified the state of the database; however, this was done in accordance with the WAL protocol (WAL records describing the changes being reapplied are already in the log), so we know that any work we were doing that was lost as a result of the crash will be continued when the system is restarted (and the crash recovery process is started once again).

Undo: At this point in crash recovery, all the relevant records in the log have been read; the state of the system should be identical to its state just before the crash occurred. The final thing that must be done is to abort any transactions that were active at the time of the crash and undo their updates.

The following steps are taken as part of the undo phase:

- The transaction table contains an entry for each transaction that needs to be aborted: these are called the **loser transactions**. The `lastLSN` field of each entry identifies the last WAL record describing a change made by that transaction. We want to undo the changes made by loser transactions, in the *reverse order* in which the changes were made. Unlike the other phases of crash recovery, this phase reads the WAL backward.
- We construct a set `ToUndo` which initially contains the `lastLSN` values for all the loser transactions. Until `ToUndo` is empty:
  * Remove the largest LSN from `ToUndo` (in other words, the WAL record that was written most recently). Fetch the WAL record r with this LSN.

* If r is an update record, the change it specifies needs to be undone. A CLR record describing the change is written to the WAL and the object is modified to undo the original update. The `prevLSN` of r is added to `toUndo`.
* If r is a CLR record, *we know that the undo that it describes has already been applied*, because the "redo" phase has already been performed. Therefore we don't need to re-apply the undo in this phase. Rather, we can merely examine r's `undoNextLSN` field: if it is `NULL`, we have finished undoing this transaction, so an end record is written to the WAL and r is discarded. Otherwise, the `undoNextLSN` of r is added to `toUndo`.

Once `toUndo` is empty, crash recovery is complete and the database is in a consistent state.

Crash During Recovery: Some care needs to be taken to ensure that the "undo" phase is robust enough to allow for crashes to occur during crash recovery. Unlike the other phases in crash recovery, this phase *does* modify the WAL (by writing additional CLR records) as well as the on-disk state.

- We know that the state of the system is captured by the WAL and the on-disk storage.
- This phase modifies the on-disk storage by undoing updates. However, before an update is reversed, a CLR record describing the change is written to the WAL. If the system crashes before crash recovery has completed, this CLR record will be reapplied by the "redo" phase when the system restarts. Therefore, any updates made to pages is sure to persist through system crashes.
- This phase modifies the WAL by writing CLR records. However, we know that this does not prevent crash recovery. The change that is undone by the CLR record is described by a WAL record that precedes the CLR record in the WAL; therefore, by the time that the "redo" phase sees the CLR that reverses the change, the original change will have been redone, so that the undo can also be redone. When the CLR in question is processed by the undo phase, its change has already been redone — all we need to do is continue "where we left off" by adding the CLR record's `undoNextLSN` to the `toUndo` set.

- Note that aborting a single transaction during normal database operation can be performed as a simple special-case of the "Undo" phase described above. The considerations described above ensure the undo phase persists cleanly through crashes apply to normal transaction aborts as well.