Chapter 14: Query Optimization

Introduction

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation (Chapter 13)
- Cost difference between a good and a bad way of evaluating a query can be enormous
  - Example: performing a \( r \times s \) followed by a selection \( r.A = s.B \) is much slower than performing a join on the same condition
- Need to estimate the cost of operations
  - Depends critically on statistical information about relations which the database must maintain
  - E.g., number of tuples, number of distinct values for join attributes, etc.
- Need to estimate statistics for intermediate results to compute cost of complex expressions

Introduction (Cont.)

- Generation of query-evaluation plans for an expression involves several steps:
  1. Generating logically equivalent expressions
     - Use equivalence rules to transform an expression into an equivalent one.
  2. Annotating resultant expressions to get alternative query plans
  3. Choosing the cheapest plan based on estimated cost
- The overall process is called cost based optimization.

Overview of chapter

- Statistical information for cost estimation
- Equivalence rules
- Cost-based optimization algorithm
- Optimizing nested subqueries
- Materialized views and view maintenance

Statistical Information for Cost Estimation

- \( n_r \): number of tuples in a relation \( r \).
- \( b_r \): number of blocks containing tuples of \( r \).
- \( s_r \): size of a tuple of \( r \).
- \( f_r \): blocking factor of \( r \) — i.e., the number of tuples of \( r \) that fit into one block.
- \( V(A, r) \): number of distinct values that appear in \( r \) for attribute \( A \); same as the size of \( \Pi_A(r) \).
- \( SC(A, r) \): selection cardinality of attribute \( A \) of relation \( r \); average number of records that satisfy equality on \( A \).
- If tuples of \( r \) are stored together physically in a file, then:
  \[
  h_r = \frac{n_r}{b_r}
  \]

Catalog Information about Indices

- \( f_i \): average fan-out of internal nodes of index \( i \), for tree-structured indices such as B+-trees.
- \( HT_i \): number of levels in index \( i \) — i.e., the height of \( i \).
  - For a balanced tree index (such as B+-tree) on attribute \( A \) of relation \( r \), \( HT_i = \log_2(V(A, r)) \).
  - For a hash index, \( HT_i \) is 1.
- \( LB_i \): number of lowest-level index blocks in \( i \) — i.e., the number of blocks at the leaf level of the index.
Measures of Query Cost

- Recall that
  - Typically disk access is the predominant cost, and is also relatively easy to estimate.
  - The number of block transfers from disk is used as a measure of the actual cost of evaluation.
  - It is assumed that all transfers of blocks have the same cost.
  - Real life optimizers do not make this assumption, and distinguish between sequential and random disk access.

- We do not include cost to writing output to disk.
- We refer to the cost estimate of algorithm A as $E_A$.

Statistical Information for Examples

- $f_{\text{account}} = 20$ (20 tuples of account fit in one block)
- $V(\text{branch-name, account}) = 50$ (50 branches)
- $V(\text{balance, account}) = 500$ (500 different balance values)
- $n_{\text{account}} = 10000$ (account has 10,000 tuples)
- Assume the following indices exist on account:
  - A primary, B+-tree index for attribute branch-name
  - A secondary, B+-tree index for attribute balance

Implementation of Complex Selections

- The selectivity of a condition $\Theta_i$ is the probability that a tuple in the relation $r$ satisfies $\Theta_i$. If $s_i$ is the number of satisfying tuples in $r$, the selectivity of $\Theta_i$ is given by $s_i/n_r$. 
- Conjunction: $\sigma_{\Theta_1 \land \Theta_2 \ldots \land \Theta_n}(r)$. The estimate for number of tuples in the result is: $n_r = s_1 * s_2 * \ldots * s_n / n_r^*$
- Disjunction: $\sigma_{\Theta_1 \lor \Theta_2 \ldots \lor \Theta_n}(r)$. Estimated number of tuples: $n_i = 1 - (1 - s_1/n_r) * (1 - s_2/n_r) * \ldots * (1 - s_n/n_r)$
- Negation: $\sigma_{\neg \Theta}(r)$. Estimated number of tuples: $n_i = \text{size}(r(n_r))$

Selection Size Estimation

- Equality selection $\sigma_{\Theta_1, \Theta_2}(r)$
  - $|\sigma_{\Theta_1, \Theta_2}(r)|$ = number of records that will satisfy the selection
  - E.g. Binary search cost estimate becomes $E_{2} = \left| \log_{2}(b_{j}) \right| / f_{r}$

- Equality condition on a key attribute: $SC(A, r) = 1$

Selections Involving Comparisons

- Selections of the form $\sigma_{\Theta_1}(r)$ (case of $\sigma_{\Theta_1, \Theta_2}(r)$ is symmetric)
  - Let $c$ denote the estimated number of tuples satisfying the condition.
  - If $\min(A, r)$ and $\max(A, r)$ are available in catalog
    - $C = 0$ if $v < \min(A, r)$
    - $C = n_r - v - \min(A, r)$
    - $C = \frac{n_r - \max(A, r) - v}{\max(A, r) - \min(A, r)}$

- In absence of statistical information $c$ is assumed to be $n_r/2$.

Join Operation: Running Example

Running example:
- depositor $\land$ customer

Catalog information for join examples:
- $n_{\text{customer}} = 10000$.
- $f_{\text{customer}} = 25$, which implies that $n_{\text{customer}} = 10000/25 = 400$.
- $n_{\text{depositor}} = 5000$.
- $f_{\text{depositor}} = 50$, which implies that $n_{\text{depositor}} = 5000/50 = 100$.
- $V(\text{customer-name, depositor}) = 2500$, which implies that, on average, each customer has two accounts.

Also assume that customer-name in depositor is a foreign key on customer.

Estimation of the Size of Joins

- The Cartesian product $r \times s$ contains $n_r * n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.
- If $R \cap S = \emptyset$, then $r \times s$ is the same as $r \times s$.
- If $R \cap S$ is a key for $R$, then a tuple of $s$ will join with at most one tuple from $r$,
  - therefore, the number of tuples in $r \times s$ is no greater than the number of tuples in $s$.
- If $R \cap S$ is a foreign key in $S$ referencing $R$, then the number of tuples in $r \times s$ is exactly the same as the number of tuples in $s$.
  - The case for $R \cap S$ being a foreign key referencing $S$ is symmetric.

- In the example query depositor $\land$ customer, customer-name in depositor is a foreign key of customer,
  - hence, the result has exactly $n_{\text{depositor}}$ tuples, which is 5000.

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{\emptyset\}$ is not a key for $R$ or $S$,
  - If we assume that every tuple $t$ in $R$ produces tuples in $R \times S$, the number of tuples in $R \times S$ is estimated to be:
    - $n_r * n_s / V(\emptyset)$

  If the reverse is true, the estimate obtained will be:
  - $n_r * n_s / V(R, r)$

  The lower of these two estimates is probably the more accurate one.
Estimation of the Size of Joins (Cont.)
- Compute the size estimates for depositor\(\land\)customer without using information about foreign keys:
  - \(V(\text{customer-name, depositor}) = 2500\), and \(V(\text{customer-name, customer}) = 10000\)
  - The two estimates are 5000 * 10000/2500 - 20,000 and 5000 * 10000/10000 = 5000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation (Cont.)
- Outer join:
  - Estimated size of \(r \bowtie s\) = size of \(r \times s\) + size of \(r\)
  - Case of right outer join is symmetric
  - Estimated size of \(r \bowtie s\) = size of \(r \times s\) + size of \(r \times s\)

Estimation of Distinct Values (Cont.)
- Joins: \(r \bowtie s\)
  - If all attributes in \(A\) are from \(r\)
    - \(V(A, r \bowtie s) = \min(V(A), n_r \bowtie s)\)
  - If \(A\) contains attributes \(A1\) from \(r\) and \(A2\) from \(s\), then estimated
    - \(V(A, r \bowtie s) = \min(V(A1, r) \bowtie V(A2 - A1, s), V(A1 - A2, r) \bowtie V(A2, s), n_{r \bowtie s})\)
    - More accurate estimate can be got using probability theory, but this one works fine generally.

Transformation of Relational Expressions
- Two relational algebra expressions are said to be equivalent if on every legal database instance the two expressions generate the same set of tuples.
  - Note: order of tuples is irrelevant.
- In SQL, inputs and outputs are multisets of tuples.
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if on every legal database instance the two expressions generate the same multiset of tuples.
  - An equivalence rule says that expressions of two forms are equivalent.
    - Can replace expression of first form by second, or vice versa.

Equivalence Rules
1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.
   \(\sigma_{A}(E) = \sigma_{A_{1}}(E) \bowtie \cdots \bowtie \sigma_{A_{n}}(E)\)
2. Selection operations are commutative.
   \(\sigma_{A}(E) \bowtie \sigma_{B}(E) = \sigma_{B}(E) \bowtie \sigma_{A}(E)\)
3. Only the last in a sequence of projection operations is needed, the others can be omitted.
   \(\Pi_{A}(\Pi_{B}(...(\Pi_{D}(E))...)) = \Pi_{A}(E)\)
4. Selections can be combined with Cartesian products and theta joins.
   a. \(\sigma_{Q}(E \times E) = E_{1} \bowtie E_{2}\)
   b. \(\sigma_{Q}(E_{1} \bowtie E_{2}) = E_{1} \bowtie E_{2}\)
Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in $\theta_3$ involve only the attributes of one of the expressions ($E_i$) being joined.
   $\sigma_{\theta_3}(\Pi_{\theta}(E_i E_j)) = (\sigma_{\theta_3}(E_i)) \times (\sigma_{\theta_3}(E_j))$
   (b) When $\theta_3$ involves only the attributes of $E_i$ and $\theta_2$ involves only the attributes of $E_j$.
   $\sigma_{\theta_3}(\Pi_{\theta_2}(E_i E_j)) = (\sigma_{\theta_3}(E_i)) \times (\sigma_{\theta_2}(E_j))$

Equivalence Rules (Cont.)

8. The projections operation distributes over the theta join operation as follows:
   (a) If $\Pi$ involves only attributes from $L_1 \cup L_2$:
   $\Pi_{\theta_{L_1 \cup L_2}}(E_i \times E_j) = (\Pi_{\theta_{L_1}}(E_i)) \times (\Pi_{\theta_{L_2}}(E_j))$
   (b) Consider a join $E_1 \times E_2$.
      - Let $L_1$ and $L_2$ be sets of attributes from $E_1$ and $E_2$, respectively.
      - Let $L_1$ be attributes of $E_1$ that are involved in join condition $\theta$, but not in $L_2 \cup L_3$.
      - Let $L_2$ be attributes of $E_2$ that are involved in join condition $\theta$, but not in $L_1 \cup L_3$.
   $\Pi_{\theta_{L_1 \cup L_2}}(E_i \times E_j) = \Pi_{\theta_{L_1 \cup L_2}}(\Pi_{\theta_{L_1}}(E_i)) \times (\Pi_{\theta_{L_2 \cup L_2}}(E_j))$

Transformation Example

- Query: Find the names of all customers who have an account at some branch located in Brooklyn.
  $\Pi_{\text{customer-name}}(\sigma_{\text{branch-city} = \text{Brooklyn}} (\text{account} \times \text{depositor}))$
- Transformation using join associatively (Rule 6a):
  $\Pi_{\text{customer-name}}(\sigma_{\text{branch-city} = \text{Brooklyn}} (\text{branch} \times (\text{account} \times \text{depositor})))$
- Performing the selection as early as possible reduces the size of the relation to be joined.

Example with Multiple Transformations

- Query: Find the names of all customers with an account at a Brooklyn branch whose account balance is over $1000$.
  $\Pi_{\text{customer-name}}(\sigma_{\text{branch-city} = \text{Brooklyn}} \times \text{balance} > 1000 \times \text{branch} \times (\text{account} \times \text{depositor}))$
- Transformation using join associatively (Rule 6a):
  $\Pi_{\text{customer-name}}(\sigma_{\text{branch-city} = \text{Brooklyn}} \times \text{balance} > 1000 \times \text{branch} \times (\text{account} \times \text{depositor}))$
- Second form provides an opportunity to apply the “perform selections early” rule, resulting in the subexpression
  $\sigma_{\text{branch-city} = \text{Brooklyn}} (\text{branch}) \times \sigma_{\text{balance} > 1000} (\text{account})$
- Thus a sequence of transformations can be useful.
Join Ordering Example (Cont.)

Consider the expression

\[ \Pi_{customer-name}(\sigma_{branch-city = \text{"Brooklyn"}}(\text{branch}) \times \text{account} \times \text{depositor}) \]

Could compute account \( \times \) depositor first, and join result with

\[ \sigma_{branch-city = \text{"Brooklyn"}}(\text{branch}) \]

but account \( \times \) depositor is likely to be a large relation.

Since it is more likely that only a small fraction of the bank's customers have accounts in branches located in Brooklyn, it is better to compute

\[ \sigma_{branch-city = \text{"Brooklyn"}}(\text{branch}) \times \text{account} \]

first.

Evaluation Plan

An evaluation plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.

Cost-Based Optimization

Consider finding the best join-order for \( r_1 \times r_2 \times \ldots \times r_n \).

There are \( (2^n - 1)!! \) different join orders for above expression. With \( n = 7 \), the number is 665280, with \( n = 10 \), the number is greater than 176 billion!

No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of \( \{r_1, r_2, \ldots, r_k\} \) is computed only once and stored for future use.

Dynamic Programming in Optimization

To find best join tree for a set of \( n \) relations:

1. To find best plan for a set \( S \) of \( n \) relations, consider all possible plans of the form: \( S_1 \times (S - S_1) \) where \( S_1 \) is any non-empty subset of \( S \).
2. Recursively compute costs for joining subsets of \( S \) to find the cost of each plan. Choose the cheapest of the \( 2^n - 1 \) alternatives.
3. When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it.

More details shortly.
Join Order Optimization Algorithm

procedure findbestplan(S)
if (bestplan[S].cost ≠ ∞)
    return bestplan[S]
else if bestplan[S] has not been computed earlier, compute it now
for each non-empty subset S1 of S such that S1 ⊆ S
    P1 = findbestplan(S1)
    P2 = findbestplan(S - S1)
    A = best algorithm for joining results of P1 and P2
    cost = P1.cost + P2.cost + cost of A
    if cost < bestplan[S].cost
        bestplan[S].cost = cost
        bestplan[S].plan = “execute P1; plan; execute P2; plan;
             join results of P1 and P2 using A”
return bestplan[S]

Left Deep Join Trees

- In left-deep join trees, the right-hand-side input for each join is a relation, not the result of an intermediate join.

Interesting Orders in Cost-Based Optimization

- Consider the expression (r1 ×r2 ×r3) ×r4 ×r5
- An interesting sort order is a particular sort order of tuples that could be useful for a later operation.
- Generating the result of r1 ×r2 ×r3 ×r4 ×r5 sorted on the attributes common with r1 or r5 may be useful, but generating it sorted on the attributes common only r1 and r5 is not useful.
- Using merge-join to compute r1 ×r2 ×r3 ×r4 ×r5 may be costlier, but may provide an output sorted in an interesting order.
- Not sufficient to find the best join order for each subset of the set of n given relations; must find the best join order for each subset, for each interesting sort order
- Simple extension of earlier dynamic programming algorithms
- Usually, number of interesting orders is quite small and doesn’t affect time/space complexity significantly

Steps in Typical Heuristic Optimization

1. Deconstruct conjunctive selections into a sequence of single selection operations (Equiv., rule 1).
2. Move selection operations down the query tree for the earliest possible execution (Equiv., rules 2, 7a, 7b, 11).
3. Execute first those selection and join operations that will produce the smallest relations (Equiv., rule 8).
4. Replace Cartesian product operations that are followed by a selection condition by join operations (Equiv., rule 10).
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (Equiv., rules 3, 8a, 8b, 12).
6. Identify those subtrees whose operations can be pipelined and execute them using pipelining.

Structure of Query Optimizers (Cont.)

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
- System R and Starburst use a hierarchical procedure based on the nested-block concept of SQL: heuristic rewriting followed by cost-based join-order optimization.
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
- This expense is usually more than offset by savings at query-execution time, particularly by reducing the number of slow disk accesses.
Optimizing Nested Subqueries**

- SQL conceptually treats nested subqueries in the where clause as functions that take parameters from outer level query and return a single value or set of values.
  - Parameters are variables from outer level query that are used in the nested subquery; such variables are called correlation variables.

- E.g.: 
  - `select customer-name` 
  - from borrower 
  - where exists (select * 
  - from depositor where depositor.customer-name = borrower.customer-name)

- Conceptually, nested subquery is executed once for each tuple in the cross-product generated by the outer level from clause.
  - Such evaluation is called correlated evaluation.

  - Note: other conditions in where clause may be used to compute a join (instead of a cross-product) before executing the nested subquery.

Optimizing Nested Subqueries (Cont.)

In general, SQL queries of the form below can be rewritten as shown:

- Rewrite: `select ... from L, where P, and exists (select * from L') where P')`

  - To: `create table U as select distinct V from L, where P, select ... from L', U, where P'` 

  - `P'` contains predicates in `P` that do not involve any correlation variables.
  - `P''` reintroduces predicates involving correlation variables, with relations renamed appropriately.
  - `V` contains all attributes used in predicates with correlation variables.

Optimizing Nested Subqueries (Cont.)

- In our example, the original nested query would be transformed to:
  - `create table t as select distinct customer-name` 
  - from depositor` 
  - select customer-name` 
  - from borrower, t` 

  - The process of replacing a nested query by a query with a join (possibly with a temporary relation) is called **decorrelation**.

  - Decorrelation is more complicated when:
    - the nested subquery uses aggregation, or
    - the result of the nested subquery is used to test for equality, or
    - when the condition linking the nested subquery to the outer query is not exists, and so on.

Materialized Views**

- A materialized view is a view whose contents are computed and stored.

  - Consider the view:
    - `create view branch-total-loan(branch-name, total-loan) as select branch-name, sum(amount)` 
    - from loan 
    - groupby branch-name

  - Materializing the above view would be very useful if the total loan amount is required frequently.
  - `Saves the effort of finding multiple tuples and adding up their amounts`

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as materialized view maintenance.

  - Materialized views can be maintained by recomputation on every update.

  - A better option is to use **incremental view maintenance**.

  - Changes to database relations are used to compute changes to materialized view, which is then updated.

  - View maintenance can be done by:
    - Manually defining triggers on insert, delete, and update of each relation in the view definition.
    - Manually written code to update the view whenever database relations are updated.
    - Supported directly by the database.

Incremental View Maintenance

- The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**.

  - Set of tuples inserted to and deleted from r are denoted I_r and D_r.

  - To simplify our description, we only consider inserts and deletes.

  - We replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple.

  - We describe how to compute the change to the result of each relational operation, given changes to its inputs.

  - We then outline how to handle relational algebra expressions.

Join Operation

- Consider the materialized view `v = r \times s` and an update to r.

  - Let `r^{old}` and `r^{new}` denote the old and new states of relation r.

  - Consider the case of an insert to r:
    - We can write `r^{new} = r^{old} \cup \delta_r`.
    - And rewrite the above to `(r^{old} \times s) \cup (\delta_r \times s)`.
    - But `(\delta_r \times s)` is simply the old value of the materialized view, so the incremental change to the view is just `\delta_r \times s`.

    - Thus, for inserts `\nu^{new} = \nu^{old} + (\delta_r \times s)`.

    - Similarly for deletes `\nu^{new} = \nu^{old} - (\delta_r \times s)`.
Selection and Projection Operations

- Selection: Consider a view \( v = \sigma_{\theta}(r) \).
- \( \nu^{v} = \nu^{r} \cap \sigma_{\theta}(r) \)
- \( \sigma_{\theta}(r) \) is a more difficult operation

Projection is a more difficult operation

\[ R = (A, B, C) \}
\[ \Pi_{A}(r) \]

If the tuple is deleted from \( r \), we decrement the count of the corresponding tuple in \( \Pi_{A}(r) \)

For each tuple in a projection \( \Pi_{A}(r) \), we will keep a count of how many times it was derived

- On insert of a tuple to \( r \), if the resultant tuple is already in \( \Pi_{A}(r) \) we increment its count, else we add a new tuple with count = 1
- On delete of a tuple from \( r \), we decrement the count of the corresponding tuple in \( \Pi_{A}(r) \)

Aggregate Operations (Cont.)

- min, max: \( v = \sigma_{\theta}(r) \)
- Handling insertions on \( r \) is straightforward.
- Maintaining the aggregate values min and max on deletions may be more expensive. We have to look at the other tuples of \( r \) that are in the same group to find the new minimum

Handling Expressions

- To handle an entire expression, we derive expressions for computing the incremental change to the result of each sub-expressions, starting from the smallest sub-expressions.
- E.g. consider \( E_{1} \times E_{2} \) where each of \( E_{1} \) and \( E_{2} \) may be a complex expression
  - Suppose the set of tuples to be inserted into \( E_{1} \) is given by \( D_{1} \)
    - Computed earlier, since smaller sub-expressions are handled first
  - Then the set of tuples to be inserted into \( E_{1} \times E_{2} \) is given by \( D_{1} \times E_{2} \)
    - This is just the usual way of maintaining joins

Materialized View Selection

- Materialized view selection: “What is the best set of views to materialize?”,
  - This decision must be made on the basis of the system workload
- Indices are just like materialized views, problem of index selection is closely related, to that of materialized view selection, although it is simpler.
- Some database systems, provide tools to help the database administrator with index and materialized view selection.

Aggregation Operations

- count: \( v = \sigma_{\theta}(r) \)
  - When a set of tuples \( r \) is inserted
    - For each tuple in \( r \), if the corresponding group is already present in \( v \), we increment its count, else we add a new tuple with count = 1
  - When a set of tuples \( s \) is deleted
    - For each tuple in \( s \), we look for the group \( t.A \) in \( v \), and subtract 1 from the count of the group for the tuple
      - If the count becomes 0, we delete from \( v \) the tuple for the group \( t.A \)
  - sum: \( v = \sigma_{\theta}(r) \)
    - We maintain the sum in a manner similar to count, except we add/subtract the B value instead of adding/subtracting 1 for the count
    - Additionally we maintain the count in order to detect groups with no tuples. Such groups are deleted from \( v \)
  - Cannot simply test for sum = 0 (why?)
    - To handle the case of avg, we maintain the sum and count aggregate values separately, and divide at the end

Other Operations

- Set intersection: \( v = r \cap s \)
  - When a tuple is inserted in \( r \), if it is present in \( s \), and if so we add it to \( v \)
  - If the tuple is deleted from \( r \), we delete it from the intersection if it is present.
  - Updates to \( s \) are symmetric.
  - The other set operations, union and set difference are handled in a similar fashion.
  - Outer joins are handled in much the same way as joins but with some extra work
    - We leave details to you.

Query Optimization and Materialized Views

- Rewriting queries to use materialized views:
  - A materialized view \( v = r \times s \) is available
  - A user submits a query \( r \times s \times t \)
  - We can rewrite the query as \( v \times t \)
    - Whether to do so depends on cost estimates for the two alternative
  - Replacing a use of a materialized view by the view definition:
    - A materialized view \( v = r \times s \) is available, but without any index on it
    - User submits a query \( \sigma_{\theta}(v) \)
    - Suppose also that \( s \) has an index on the common attribute \( B \), and \( r \) has an index on attribute \( A \)
    - The best plan for this query may be to replace \( v \) by \( r \times s \), which can lead to the query plan \( \sigma_{\theta}(r) \times s \)
  - Query optimizer should be extended to consider all above alternatives and choose the best overall plan

End of Chapter

(Extra slides with details of selection cost estimation follow)
Cost Estimate Example (Indices)

Consider the query is \( \sigma_{\text{branch-name} = \text{Perryridge}}(\text{account}) \), with the primary index on \( \text{branch-name} \):

- Since \( V(\text{branch-name}, \text{account}) = 50 \), we expect that 10,000/50 = 200 tuples of the \( \text{account} \) relation pertain to the Perryridge branch.
- Since the index is a clustering index, 200/20 = 10 block reads are required to read the \( \text{account} \) tuples.
- Several index blocks must also be read. If \( B^+ \)-tree index stores 20 pointers per node, then the \( B^+ \)-tree index must have between 3 and 5 leaf nodes and the entire tree has a depth of 2. Therefore, 2 index blocks must be read.
- This strategy requires 12 total block reads.

Example of Cost Estimate for Complex Selection

Consider a selection on \( \text{account} \) with the following condition:

where \( \text{branch-name} = \text{Perryridge} \) and \( \text{balance} = 1200 \)

Consider using algorithm A8:

- The \( \text{branch-name} \) index is clustering, and if we use it the cost estimate is 12 block reads (as we saw before).
- The balance index is non-clustering, and \( V(\text{balance}, \text{account}) = 500 \), so the selection would retrieve 10,000/500 = 20 accounts. Adding the index block reads, gives a cost estimate of 22 block reads.
- Thus using \( \text{branch-name} \) index is preferable, even though its condition is less selective.
- If both indices were non-clustering, it would be preferable to use the balance index.

Selections Using Indices

- **Index scan** – search algorithms that use an index; condition is on search-key of index.
- **A3** (primary index on candidate key, equality). Retrieve a single record that satisfies the corresponding equality condition \( E_{A3} = HT_{i} + 1 \)
- **A4** (primary index on nonkey, equality) Retrieve multiple records. Let the search-key attribute be \( A \).
  
  \[
  E_{A4} = HT_{i} + \frac{\sigma(SQ(A))}{l}
  \]
- **A5** (equality on search-key of secondary index).
  - Retrieve a single record if the search-key is a candidate key \( E_{A5} = HT_{i} + 1 \)
  - Retrieve multiple records (each may be on a different block) if the search-key is not a candidate key. \( E_{A5} = HT_{i} + SQ(A) \)

Selections Involving Comparisons

Selections of the form \( \sigma_{A_{l} = C}(r) \) or \( \sigma_{A_{l}}(r) \) by using a linear file scan or binary search, or by using indices in the following ways:

- **A6** (primary index, comparison). The cost estimate is:
  
  \[
  E_{A6} = HT_{i} + \frac{c}{l}
  \]
  
  where \( c \) is the estimated number of tuples satisfying the condition. In absence of statistical information \( c \) is assumed to be \( n/l \).
- **A7** (secondary index, comparison). The cost estimate:
  
  \[
  E_{A7} = HT_{i} + LB \cdot \frac{c + c}{n}
  \]
  
  where \( c \) is defined as before. (Linear file scan may be cheaper if \( c \) is larger).