# Query Processing and Optimization

CSCI 220: Database Management and Systems Design

Slides adapted from
Simon Miner
Gordon College

# Practice Quiz: Indexing

- With a neighbor, discuss the benefits and drawbacks of:
  - Hashed indexes
  - Ordered indexes (e.g., B+ Tree)
  - Clustering indexes

# Today you will learn…

- How databases execute queries efficiently
- Why relational algebra is useful!

# Library Database Schema

**book**

| | | | |
|---|---|---|---|
| call_number | copy_number | accession_number | title |

**book_author**

| | |
|---|---|
| call_number | author |

**checked_out**

| | | | |
|---|---|---|---|
| call_number | copy_number | borrower_id | date_due |

**borrower**

| | |
|---|---|
| borrower_id | name |

# Example Query

- Find the titles of all books written by "Bruce Schneier"
- SELECT title
  FROM book NATURAL JOIN book_author
  WHERE author = "Bruce Schneier"
- Many possible execution plans. For example:
  A. $\pi_{title} (\boldsymbol{\sigma}_{author = \text{'Bruce Schneier'}} (Book \bowtie BookAuthor))$
  B. $\pi_{title} (Book \bowtie (\boldsymbol{\sigma}_{author = \text{'Bruce Schneier'}} BookAuthor))$

# Evaluating Execution Plans

- Compare:
  A. $\pi_{title}$ ($\boldsymbol{\sigma}_{author\ =\ \text{'Bruce Schneier'}}$ (Book $\bowtie$ BookAuthor))
  B. $\pi_{title}$ (Book $\bowtie$ ($\boldsymbol{\sigma}_{author\ =\ \text{'Bruce Schneier'}}$ BookAuthor))
- Relevant information:
  - How many records are in each table?
  - What indexes do we have?
  - How many books did Bruce Schneier write?

# Evaluating Execution Plans

- Compare:
  A. $\pi_{title}$ ($\boldsymbol{\sigma}_{author\ =\ \text{'Bruce Schneier'}}$ (Book ⋈ BookAuthor))
  B. $\pi_{title}$ (Book ⋈ ($\boldsymbol{\sigma}_{author\ =\ \text{'Bruce Schneier'}}$ BookAuthor))
- Suppose:
  - BookAuthor has 20K tuples
  - Book has 10K tuples (an average of two authors per book)
  - Only 2 BookAuthor tuples contain "Bruce Schneier"
  - Relevant indexes exist
- What's the performance difference?
  A. Processes all 10K book tuples and 20K bookAuthor tuples to create a temporary relation with 20K tuples. Processes at least 50K tuples.
  B. Uses indexes to locate 2 BookAuthor tuples and 2 corresponding book tuples. Processes just 4 tuples!

# Outline

- Selection Strategies
- Join Strategies
- Join Size Estimation
- Rules of Equivalence

# Selection Strategies

- How to perform selection (σ)?
- Linear search is always an option
  - Full table scan
  - Potentially requires accessing every disk block in the table
- Alternatively, use an index
  - Binary search, tree search, or hash table lookup
  - Indexes themselves require disk accesses, but it's usually worth it
  - Indexes may be partly or entirely stored in memory

# Query Type vs Index Type

| Condition | Example | Clustering / Primary Index | Ordered Index | Hashed Index |
|---|---|---|---|---|
| Exact match on candidate key | id = 12345 | **Easy to locate.** | **Easy to locate.** | **Easy to locate.** |
| Exact match on non-key | status = 'Active' | N/A | **Find first match (+ potential scan)** | **Find first match (+ potential scan)** |
| Range query | age between 21 and 65 | **Find first match + sequential scan** | **Find first match + scan, but slower** | Not useful |
| Complex query | color = 'blue' or status = 'Inactive' | Not useful | Not useful, unless **multiple or multi-column indexes** | Not useful, unless **multiple or multi-column indexes** |

# Join Strategies

- Joins are most expensive part of query processing
  - Number of tuples examined can approach the product of the number of records in tables being joined
- Example
  - $\sigma_{\text{Borrower.name = BookAuthor.author}} \text{Borrower} \times \text{BookAuthor}$
    - Where BookAuthor has 10K tuples and Borrower has 2K tuples
    - Cartesian join yields 20 million tuples to process

# Nested Loop Join

```
for (int i = 0; i < 2000; i++) {
  retrieve Borrower[i];
  for (int j = 0; j < 10000; j++) {
    retrieve BookAuthor[j];
    if (Borrower[i].name == BookAuthor[j].author) {
      construct tuple from Borrower[i] & BookAuthor[j];
    }
  }
}
```

# Nested Loop Join

- Simplest and least efficient approach. If each retrieval requires a separate disk access:
  - 2K accesses for Borrower tuples (outer loop)
  - 20 million accesses for BookAuthor tuples (inner loop)
  - 20,002,000 disk accesses total
- If each disk access takes 10ms, this takes:
  > 200K seconds ≈ 55 hours
- Doesn’t count time needed to write the temporary join relation (it might not fit in memory)

# Nested Block Join

```
for (int i = 0; i < 2000; i += 20 ) {
  retrieve block containing Borrower[i]..Borrower[i+19];
  for (int j = 0; j < 10000; j += 20) {
    retrieve block containing BookAuthor[j]..
                              BookAuthor[j+19];
    for (int k = 0; k < 19; k++)
      for(int l = 0; l < 20; l++)
        if (Borrower[i+k].name == BookAuthor[j+l].author)
          construct tuple from Borrower[i+k] &
                               BookAuthor[j+1];
  }
}
```

# Nested Block Join

- Since tables are stored in blocks, we processes data by block. If each block contains 20 tuples:
  - 100 accesses for Borrower tuples (outer loop)
  - 500 accesses for BookAuthor tuples (inner loop) executed 100 times = 50K accesses
  - 50,100 disk accesses total
- This requires 50,100 * 10 ms ≈ 8.5 minutes
- 400x faster than nested loop join!

# Buffering an Entire Relation

```
for (int i = 0; i < 2000; i += 20)
  retrieve and buffer block containing
    Borrower[i]..Borrower[i+19];

for (int j = 0; j < 10000; j += 20) {
  retrieve block containing BookAuthor[j]..BookAuthor[j+19];
  for (int k = 0; k < 2000; k++)
    for (int l = 0; l < 20; l++)
      if (Borrower[k].name == BookAuthor[j+l].author)
        construct tuple from Borrower[k] & BookAuthor[j+l];
}
```

# Buffering an Entire Relation

- Using memory, improvement is possible. If the entire Borrower relation can be stored memory:
  - 100 accesses for Borrower tuples (first loop)
  - 500 accesses for BookAuthor tuples (second loop)
  - 600 accesses total
- The requires 600 * 10 ms = 6 seconds
- This is the best possible scenario, since every record is only processed once

# Using Indexes to Speed Up Joins

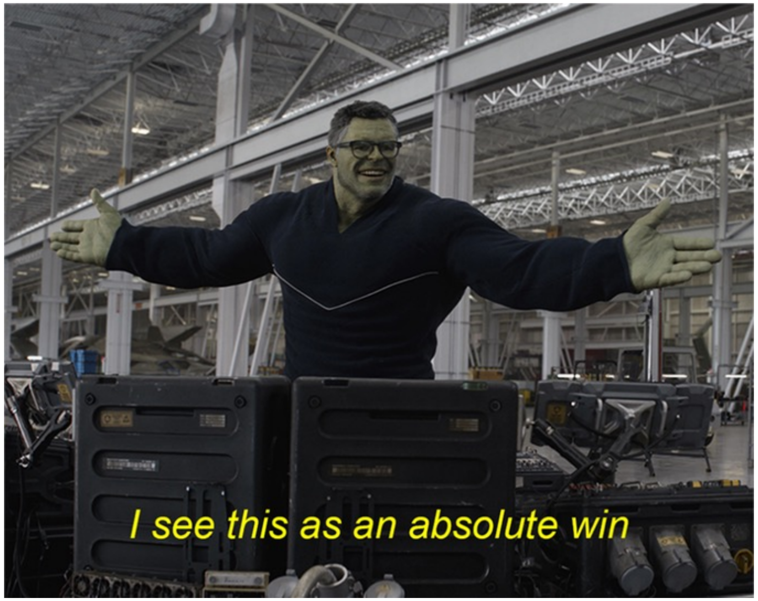
- Example: Borrower ⋈ CheckedOut
- Assume:
  - 2K Borrower tuples, 1K CheckedOut tuples
  - 20 records per block: 100 and 50 blocks for each table, respectively
  - We cannot buffer either table entirely
- Without indexes, a nested block join takes 5050 or 5100 disk accesses
  - Depends on which table is in the outer loop

# Using Indexes to Speed Up Joins

- Example: Borrower ⋈ CheckedOut
- Suppose we have index on Borrower.borrowerID
  - We scan all 1000 CheckedOut records (50 blocks)
  - Then, we use the index to match each with a Borrower record
- We only process 1000 CheckedOut records and 1000 Borrower records

# Using Indexes to Speed Up Joins

- Limitations:
  - Each borrower may require a separate disk access
    - 50 accesses for CheckedOut
    - 1000 accesses for Borrower
  - If the index doesn't fit in memory, traversing the index requires disk accesses
    - B+ Tree Indexes require more accesses than Hashed Indexes




- Nevertheless, a major improvement!

# Temporary Indexes

- Indexes created and buffered for the purpose of a single query and then discarded
- Suppose neither Borrower nor CheckedOut is indexed
  - Borrower ⋈ CheckedOut might cause a temporary index to be built on Borrower.borrowerID
  - If an index entry takes ~10 bytes, entire index will be ~20K
  - Index construction requires reading all 2K borrowers = 100 disk accesses
  - Join itself costs up to 1050 disk accesses (see previous slide)
  - Total of 1150 disk accesses

# Merge Join

- Suppose both tables in a joined are stored in ascending order by the join key
- Using a merge join, we can fetch each tuple once: 50 + 100 = 150 total disk accesses

# Merge Join

```
get first tuple from Borrower;
get first tuple from CheckedOut;
while (we still have valid tuples from both relations) {
  if (Borrower.borrowerID == CheckedOut.borrowerID) {
    output one tuple to the result;
    get next tuple from CheckedOut;
    // We might have more checkouts for this borrower,
    // so keep current borrower tuple
  }
  else if (Borrower.borrowerID < CheckedOut.borrowerID)
    get next tuple from Borrower;
  else
    get next tuple from CheckedOut;
}
```

# Order of Joins

- For multiple joins, performance can be greatly impacted by the order of the joins
- Example: $\pi_{\text{last, first, authorName}}$ Borrower ⋈ BookAuthor ⋈ CheckedOut
- Assume:
  - 2K Borrower, 1K CheckedOut, and 10K Author tuples
  - Each book has an average of 2 authors
- Three ways to do the join operations:
  A. ( Borrower ⋈ BookAuthor ) ⋈ CheckedOut
  B. ( BookAuthor ⋈ CheckedOut ) ⋈ Borrower
  C. ( Borrower ⋈ CheckedOut ) ⋈ BookAuthor
- Final number of tuples is the same, but intermediate joins create temporary tables. Which order is most efficient?

# Order of Joins

- Assume:
  - 2K Borrower, 1K CheckedOut, and 10K Author tuples
  - Each book has an average of 2 authors
- Three ways to do the (binary commutative) join operations:
  A. ( Borrower ⋈ BookAuthor ) ⋈ CheckedOut
  B. ( BookAuthor ⋈ CheckedOut ) ⋈ Borrower
  C. ( Borrower ⋈ CheckedOut ) ⋈ BookAuthor
- Example:
  A. Borrower and BookAuthor have no attributes in common, so a cartesian product is formed. This results in a temporary table with 20 million tuples!

# Statistics and Query Optimization

- Using statistics about database objects can help speed up queries
- Maintaining statistics as the data in the database changes is a manageable process
- Types of statistics
  - Table statistics
  - Column statistics

# Table Statistics

- On a relation r:
  - $n_r$ = number of tuples in the relation
  - $l_r$ = size (in bytes) of a tuple in the relation
  - $f_r$ = blocking factor, number of tuples per block
  - $b_r$ = number of blocks used by the relation
- Thus:
  - $f_r$ = floor( block size / $l_r$ ) if tuples do not span blocks
  - $b_r$ = ceiling( $n_r$ / $f_r$ ) if tuples in r reside in a single file and are not clustered with other relations

# Table Statistics

| Block 1 | | Block 2 | | Block 3 | |
|---|---|---|---|---|---|
| Tuple 1 | Tuple 2 | Tuple 3 | Tuple 4 | Tuple 5 | Tuple 6 |

- **The relation contains 6 tuples ($n_r$=6)**
- **Each tuple occupies 200 bytes ($l_r$=200)**
- Each block holds 2 tuples ($f_r$=2)
- The relation occupies 3 blocks ($b_r$=3)

# Column Statistics

- On a column A, in relation r:
- V( A, r ) = number of distinct values in the column
  - If A is a superkey, then V( A, r ) = $n_r$
  - If column A is indexed, V( A, r ) s relatively easy to maintain
    - Keep track of the count of entries in the index
  - May also be useful to store a histogram of the relative frequency of column values in different ranges
  - May or may not have statistics on other columns
- The number of times each column value occurs can be estimated by $n_r$ / V( A, r )

# Example Statistics

**book_author**

| call_number | author |
|---|---|

**borrower**

| borrower_id | name |
|---|---|

**checked_out**

| call_number | copy_number | borrower_id | date_due |
|---|---|---|---|

| Table | $n_r$ | $l_r$ |
|---|---|---|
| borrower | 2000 | 58 bytes |
| checked_out | 1000 | 74 bytes |
| book_author | 10,000 | 100 bytes |

| V( A, r ) |
|---|
| V( borrower_id, Borrower ) = 2000 |
| V( borrower_id, CheckedOut ) = 100 |
| V( callNo, CheckedOut ) = 500 |
| V( callNo, BookAuthor ) = 5000 |

# Calculating the Size of a Cartesian Product

- Cartesian product: $r \times s$
  - Number of tuples in join: $n_{r \times s} = n_r * n_s$
  - Size of each tuple in join: $l_{r \times s} = l_r + l_s$
- Example: borrower × checked_out
  - $n_{borrower \times checked\_out}$
  - $l_{borrower \times checked\_out}$

# Estimating the Size of a Join

- Natural join: $r \bowtie s$, where r and s have A in common
  - Estimated number of tuples in join:
    $n_{r \bowtie s} = n_s * n_r / \max( V(A, r), V(A, s) )$
  - Number of unique values: $V(A, r \bowtie s) = \min( V(A, r), V(A, s) )$
    - Some tuples in the relation with the larger number of column values do not join with any tuples in the other relation

- If r and s have no attributes in common, then a cartesian product is performed

# Example Join Estimation

- $\pi_{\text{name, author}}$ Borrower ⋈ BookAuthor ⋈ CheckedOut
- Which evaluation plan generates the fewest tuples in the intermediate table?
  A. ( Borrower ⋈ BookAuthor ) ⋈ CheckedOut
  B. ( BookAuthor ⋈ CheckedOut ) ⋈ Borrower
  C. ( Borrower ⋈ CheckedOut ) ⋈ BookAuthor

# Rules of Equivalence

- Reordering the joins improved performance, without changing the results!
- More generally, two formulations of a query are "equivalent" if they produce the same set of results
  - Tuples aren't necessarily in the same order
- The "rules of equivalence" describe when reordering is allowed
- For a given query, a good DBMS will create several "equivalent" evaluation plans and choose the most efficient one

# Rules of Equivalence

- Example: find the titles of all books written by "Bruce Schneier"
- SELECT title
  FROM book NATURAL JOIN book_author
  WHERE author = "Bruce Schneier"
- "Equivalent" execution plans:
  A. $\pi_{title}$ ($\boldsymbol{\sigma}_{author = \text{'Bruce Schneier'}}$ (Book ⋈ BookAuthor))
  B. $\pi_{title}$ (Book ⋈ ($\boldsymbol{\sigma}_{author = \text{'Bruce Schneier'}}$ BookAuthor))
- "Equivalent" in terms of results, not performance!

# Math Review

- Commutativity:
  - A binary operation * is commutative if for all $x, y$: $x * y = y * x$
- Associativity
  - A binary operation * is associative if for all $x, y, z$: $(x * y) * z = x * (y * z)$

# Rules of Equivalence

1. **Cascade of σ.** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:

   $\sigma_{c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$

2. **Commutativity of σ.** The σ operation is commutative:

   $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$

3. **Cascade of π.** In a cascade (sequence) of π operations, all but the last one can be ignored:

   $\pi_{\text{List}_1}(\pi_{\text{List}_2}(\ldots(\pi_{\text{List}_n}(R))\ldots)) \equiv \pi_{\text{List}_1}(R)$

4. **Commuting σ with π.** If the selection condition $c$ involves only those attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:

   $\pi_{A_1, A_2, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \ldots, A_n}(R))$

# Rules of Equivalence

5. **Commutativity of $\bowtie$ (and $\times$).** The join operation is commutative, as is the $\times$ operation:

   $R \bowtie_c S \equiv S \bowtie_c R$

   $R \times S \equiv S \times R$

6. **Commuting $\sigma$ with $\bowtie$ (or $\times$).** If all the attributes in the selection condition $c$ involve only the attributes of one of the relations being joined—say, $R$—the two operations can be commuted as follows:

   $\sigma_c (R \bowtie S) \equiv (\sigma_c (R)) \bowtie S$

7. **Commuting $\pi$ with $\bowtie$ (or $\times$).** Suppose that the projection list is $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$, where $A_1, \dots, A_n$ are attributes of $R$ and $B_1, \dots, B_m$ are attributes of $S$. If the join condition $c$ involves only attributes in $L$, the two operations can be commuted as follows:

   $\pi_L (R \bowtie_c S) \equiv (\pi_{A_1, \dots, A_n} (R)) \bowtie_c (\pi_{B_1, \dots, B_m} (S))$

# Rules of Equivalence

8. **Commutativity of set operations.** The set operations $\cup$ and $\cap$ are commutative, but $-$ is not.
9. **Associativity of $\bowtie$, $\times$, $\cup$, and $\cap$.** These four operations are individually associative; that is, if both occurrences of $\theta$ stand for the same operation that is any one of these four operations (throughout the expression), we have:

   $(R \,\theta\, S)\,\theta\, T \equiv R \,\theta\, (S \,\theta\, T)$

10. **Commuting $\sigma$ with set operations.** The $\sigma$ operation commutes with $\cup$, $\cap$, and $-$. If $\theta$ stands for any one of these three operations (throughout the expression), we have:

    $\sigma_c\,(R \,\theta\, S) \equiv (\sigma_c\,(R))\,\theta\,(\sigma_c\,(S))$

11. **The $\pi$ operation commutes with $\cup$.**

    $\pi_{\mathrm{L}}\,(R \cup S) \equiv (\pi_L\,(R)) \cup (\pi_L\,(S))$

# Rules of Equivalence

12. **Converting a (σ, ×) sequence into ⋈.** If the condition $c$ of a σ that follows a × corresponds to a join condition, convert the (σ, ×) sequence into a ⋈ as follows:

    $(\sigma_c\ (R \times S)) \equiv (R \bowtie_c S)$

13. **Pushing σ in conjunction with set difference.**

    $\sigma_c\ (R - S) = \sigma_c\ (R) - \sigma_c\ (S)$

    However, **σ** may be applied to only one relation:

    $\sigma_c\ (R - S) = \sigma_c\ (R) - S$

14. **Pushing σ to only one argument in ∩.**

    If in the condition $\sigma_c$ all attributes are from relation R, then:

    $\sigma_c\ (R \cap S) = \sigma_c\ (R) \cap S$

15. **Some trivial transformations.**

    If S is empty, then $R \cup S = R$

    If the condition c in $\sigma_c$ is true for the entire $R$, then $\sigma_c\ (R) = R$.

# Push Selections Inward

- Do selections as early as possible
  - Reduces ("flattens") the number of records in the relation(s) being joined
- Example:
  - $\pi_{title}$ ($\boldsymbol{\sigma}_{author = \text{'Bruce Schneier'}}$ (Book ⋈ BookAuthor))
  - $\pi_{title}$ (Book ⋈ ($\boldsymbol{\sigma}_{author = \text{'Bruce Schneier'}}$ BookAuthor))
- Sometimes this is not feasible:
  - $\sigma_{Borrower.name = BookAuthor.author}$ Borrower × BookAuthor
- Alter the structure of the selection itself
  - Find late checked out books that cost more than $20.00.
  - $\sigma_{purchasePrice > 20 \wedge dateDue < today}$ Book ⋈ CheckedOut
  - $\sigma_{purchasePrice > 20}$ Book ⋈ $\sigma_{dateDue < today}$ CheckedOut

# Push Projections Inward

- Do projections as early as possible
  - Reduces ("narrows") the number of columns in the relation(s) being joined
- Example:
  - $\pi_{\text{name, title, dateDue}}$ Borrower ⋈ CheckedOut ⋈ Book
  - $\pi_{\text{name, title, dateDue}}$ Borrower ⋈
    ($\pi_{\text{borrowerID, title, dateDue}}$ CheckedOut ⋈ Book )
  - Reduces the number of columns in the temporary table from the intermediate join